OVERVIEW

.NET Aspire overview

Explore various aspects of .NET Aspire, including getting started, storage, database, messaging, caching, frameworks, deployment, and troubleshooting.

- .NET Aspire setup and tooling
- .NET Aspire service discovery
- .NET Aspire and launch profiles
- .NET Aspire service defaults
- .NET Aspire dashboard
- .NET Aspire local networking

- Persist volume mount sample

- Azure Cosmos DB with EF Core
- SQL Database
- SQL Database with EF Core
- Entity Framework Core migrations
- MySqlConnector Database
- MongoDB Database

Messaging integrations

- Implement Messaging with .NET Aspire
- Azure Event Hubs
- Azure Service Bus
- Azure Web PubSub
- RabbitMQ
- Apache Kafka
- NATs

Caching integrations

- Improve app caching with .NET Aspire
- Stack Exchange Redis caching overview
- Redis caching
- Redis output caching
- Redis distributed caching

Framework integrations

- Use Orleans with .NET Aspire
- Orleans voting sample
- Use Dapr with .NET Aspire
- Dapr integration sample

Deployment

- Overview
- Deploy to Azure Container Apps
- Deploy using the Azure Developer CLI
- Integrate with Application Insights
- .NET Aspire deployment manifest format

Troubleshooting

- Allow unsecure transport
- Unable to install workload
- Untrusted localhost certificate
- The specified name is already in use
- Container runtime appears to be unhealthy
- The connection string is missing
- Ask questions on Discord
- Stack Overflow — .NET Aspire

Training

- Introduction to .NET Aspire
- Create a .NET Aspire project
- Use telemetry in a .NET Aspire project
- Use databases in a .NET Aspire project
- Improve performance with a cache in a .NET Aspire project
- Send messages with RabbitMQ in a .NET Aspire project

.NET extensions

Learn about .NET extensions, including logging, dependency injection, configuration, and more. All of which are fundamental in .NET Aspire.

Fundamentals

- Logging
- Dependency injection
- Configuration
- Make HTTP requests

Telemetry

- .NET observability with OpenTelemetry
- Networking telemetry
- .NET SDK telemetry

Resiliency

- Introduction to resilient app dev
- Build resilient HTTP apps
- Implement resiliency in a cloud-native ASP.NET Core microservice

Observability

- .NET app health checks in C#
- App health checks in ASP.NET Core
- Diagnostic tools in .NET
- Diagnostic resource monitoring in .NET

.NET community resources

Find community resources for .NET, including webcasts, shows, open-source projects, and more.

.NET

- .NET documentation
- .NET Aspire samples browser
- ASP.NET documentation
- Azure documentation
- C# documentation
- .NET Discord
- Official .NET Aspire Collection

Webcasts and shows

- Azure Friday
- The Cloud Native Show
- On .NET
- On .NET Live
- .NET Community Standup
- Welcome to .NET Aspire

Open source

- .NET Aspire
- .NET Aspire samples
- .NET samples
- .NET Platform
- .NET Runtime
- ASP.NET Core

Community

Follow @dotnet on 𝕏

Follow @dotnet on Mastodon

.NET Foundation

We Are .NET

.NET Aspire YouTube playlist

.NET Aspire on YouTube

API and language reference

Search the .NET API and language reference documentation.

.NET Aspire API reference

API reference documentation for .NET Aspire

.NET API reference

API reference documentation for .NET

ASP.NET Core API reference

API reference documentation for ASP.NET Core

Are you interested in contributing to the .NET Aspire docs? For more information, see our contributor guide. Interested in the official support policy, see .NET Aspire Support Policy.

.NET Aspire overview

Article • 01/26/2025

.NET Aspire is a set of tools, templates, and packages for building observable, production ready apps. .NET Aspire is delivered through a collection of NuGet packages that bootstrap or improve specific challenges with modern app development. Today's apps generally consume a large number of services, such as databases, messaging, and caching, many of which are supported via .NET Aspire Integrations. For information on support, see the .NET Aspire Support Policy.

Why .NET Aspire?

.NET Aspire improves the experience of building apps that have a variety of projects and resources. With dev-time productivity enhancements that emulate deployed scenarios, you can quickly develop interconnected apps. Designed for flexibility, .NET Aspire allows you to replace or extend parts with your preferred tools and workflows. Key features include:

- **Dev-Time Orchestration**: .NET Aspire provides features for running and connecting multi-project applications, container resources, and other dependencies for local development environments.
- **Integrations**: .NET Aspire integrations are NuGet packages for commonly used services, such as Redis or Postgres, with standardized interfaces ensuring they connect consistently and seamlessly with your app.
- **Tooling**: .NET Aspire comes with project templates and tooling experiences for Visual Studio, Visual Studio Code, and the .NET CLI to help you create and interact with .NET Aspire projects.

Dev-time orchestration

In .NET Aspire, "orchestration" primarily focuses on enhancing the *local development* experience by simplifying the management of your app's configuration and interconnections. It's important to note that .NET Aspire's orchestration isn't intended to

replace the robust systems used in production environments, such as Kubernetes. Instead, it's a set of abstractions that streamline the setup of service discovery, environment variables, and container configurations, eliminating the need to deal with low-level implementation details. With .NET Aspire, your code has a consistent bootstrapping experience on any dev machine without the need for complex manual steps, making it easier to manage during the development phase.

.NET Aspire orchestration assists with the following concerns:

- **App composition**: Specify the .NET projects, containers, executables, and cloud resources that make up the application.
- **Service discovery and connection string management**: The app host injects the right connection strings, network configurations, and service discovery information to simplify the developer experience.

For example, using .NET Aspire, the following code creates a local Redis container resource, waits for it to become available, and then configures the appropriate connection string in the `"frontend"` project with a few helper method calls:

C#

```
// Create a distributed application builder given the command line
arguments.
var builder = DistributedApplication.CreateBuilder(args);

// Add a Redis server to the application.
var cache = builder.AddRedis("cache");

// Add the frontend project to the application and configure it to use the
// Redis server, defined as a referenced dependency.
builder.AddProject<Projects.MyFrontend>("frontend")
       .WithReference(cache)
       .WaitFor(cache);
```

For more information, see .NET Aspire orchestration overview.

> ⓘ **Important**
>
> The call to AddRedis creates a new Redis container in your local dev environment. If you'd rather use an existing Redis instance, you can use the AddConnectionString method to reference an existing connection string. For more information, see Reference existing resources.

.NET Aspire integrations

.NET Aspire integrations are NuGet packages designed to simplify connections to popular services and platforms, such as Redis or PostgreSQL. .NET Aspire integrations handle cloud resource setup and interaction for you through standardized patterns, such as adding health checks and telemetry. Integrations are two-fold - "hosting" integrations represents the service you're connecting to, and "client" integrations represents the client or consumer of that service. In other words, for many hosting packages there's a corresponding client package that handles the service connection within your code.

Each integration is designed to work with the .NET Aspire app host, and their configurations are injected automatically by referencing named resources. In other words, if *Example.ServiceFoo* references *Example.ServiceBar*, *Example.ServiceFoo* inherits the integration's required configurations to allow them to communicate with each other automatically.

For example, consider the following code using the .NET Aspire Service Bus integration:

```
C#

builder.AddAzureServiceBusClient("servicebus");
```

The AddAzureServiceBusClient method handles the following concerns:

- Registers a ServiceBusClient as a singleton in the DI container for connecting to Azure Service Bus.
- Applies ServiceBusClient configurations either inline through code or through configuration.
- Enables corresponding health checks, logging, and telemetry specific to the Azure Service Bus usage.

A full list of available integrations is detailed on the .NET Aspire integrations overview page.

Project templates and tooling

.NET Aspire provides a set of project templates and tooling experiences for Visual Studio, Visual Studio Code, and the .NET CLI. These templates are designed to help you create and interact with .NET Aspire projects, or add .NET Aspire into your existing codebase. The templates include a set of opinionated defaults to help you get started quickly - for example, it has boilerplate code for turning on health checks and logging in .NET apps. These defaults are fully customizable, so you can edit and adapt them to suit your needs.

.NET Aspire templates also include boilerplate extension methods that handle common service configurations for you:

```C#
builder.AddServiceDefaults();
```

For more information on what `AddServiceDefaults` does, see .NET Aspire service defaults.

When added to your *Program.cs* file, the preceding code handles the following concerns:

- **OpenTelemetry**: Sets up formatted logging, runtime metrics, built-in meters, and tracing for ASP.NET Core, gRPC, and HTTP. For more information, see .NET Aspire telemetry.
- **Default health checks**: Adds default health check endpoints that tools can query to monitor your app. For more information, see .NET app health checks in C#.
- **Service discovery**: Enables service discovery for the app and configures HttpClient accordingly.

Next steps

Quickstart: Build your first .NET Aspire project

Quickstart: Build your first .NET Aspire solution

Article • 11/07/2024

Cloud-native apps often require connections to various services such as databases, storage and caching solutions, messaging providers, or other web services. .NET Aspire is designed to streamline connections and configurations between these types of services. This quickstart shows how to create a .NET Aspire Starter Application template solution.

In this quickstart, you explore the following tasks:

- ✓ Create a basic .NET app that is set up to use .NET Aspire.
- ✓ Add and configure a .NET Aspire integration to implement caching at project creation time.
- ✓ Create an API and use service discovery to connect to it.
- ✓ Orchestrate communication between a front end UI, a back end API, and a local Redis cache.

Prerequisites

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0
- An OCI compliant container runtime, such as:
  - Docker Desktop or Podman. For more information, see Container runtime.
- An Integrated Developer Environment (IDE) or code editor, such as:
  - Visual Studio 2022 version 17.9 or higher (Optional)
  - Visual Studio Code (Optional)
    - C# Dev Kit: Extension (Optional)
  - JetBrains Rider with .NET Aspire plugin (Optional)

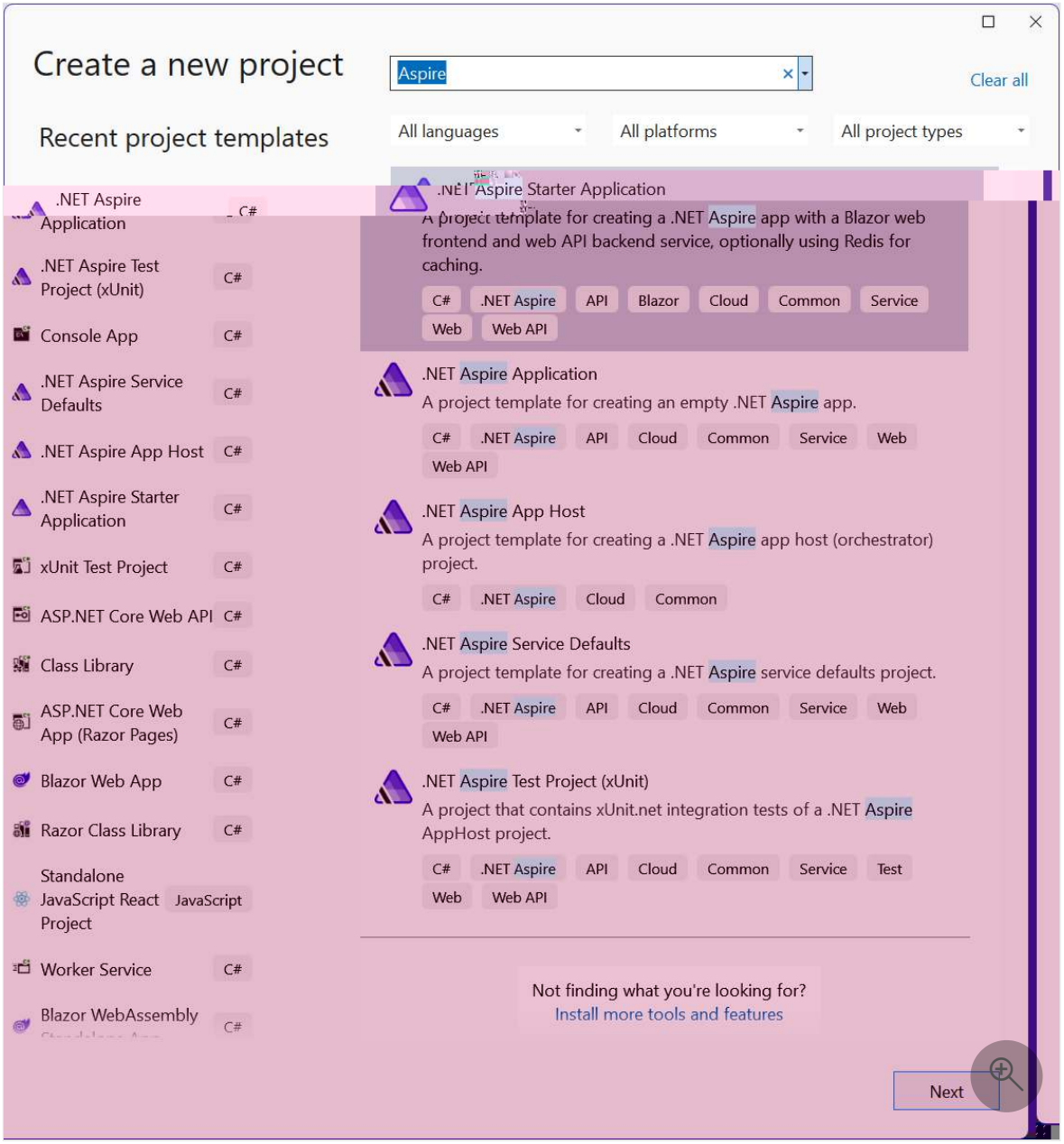
For more information, see .NET Aspire setup and tooling, and .NET Aspire SDK.

Create the .NET Aspire template

To create a new .NET Aspire Starter Application, you can use either Visual Studio, Visual Studio Code, or the .NET CLI.

Visual Studio provides .NET Aspire templates that handle some initial setup configurations for you. Complete the following steps to create a project for this quickstart:

1. At the top of Visual Studio, navigate to **File** > **New** > **Project**.
2. In the dialog window, search for *Aspire* and select **.NET Aspire Starter App**. Select **Next**.



3. On the **Configure your new project** screen:
    - Enter a **Project Name** of *AspireSample*.
    - Leave the rest of the values at their defaults and select **Next**.
4. On the **Additional information** screen:
    - Make sure **.NET 9.0 (Standard Term Support)** is selected.

- Ensure that **Use Redis for caching (requires a supported container runtime)** is checked and select **Create**.
- Optionally, you can select **Create a tests project**. For more information, see Write your first .NET Aspire test.

Visual Studio creates a new solution that is structured to use .NET Aspire.

For more information on the available templates, see .NET Aspire templates.

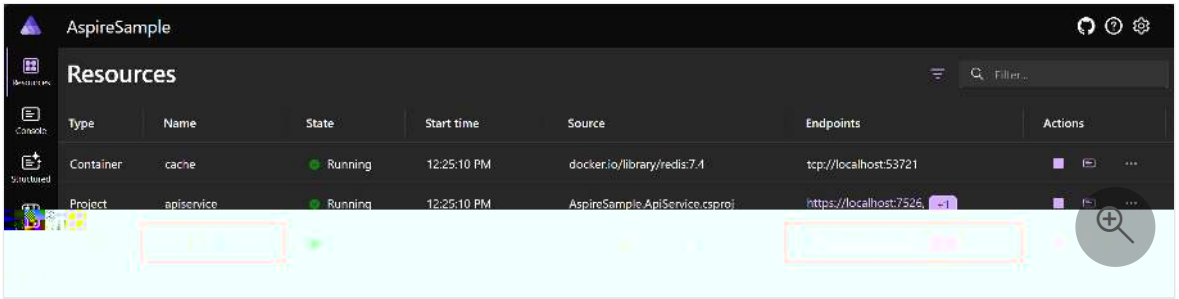
Test the app locally

The sample app includes a frontend Blazor app that communicates with a Minimal API project. The API project is used to provide *fake* weather data to the frontend. The frontend app is configured to use service discovery to connect to the API project. The API project is configured to use output caching with Redis. The sample app is now ready for testing. You want to verify the following conditions:

- Weather data is retrieved from the API project using service discovery and displayed on the weather page.
- Subsequent requests are handled via the output caching configured by the .NET Aspire Redis integration.

In Visual Studio, set the **AspireSample.AppHost** project as the startup project by right-clicking on the project in the **Solution Explorer** and selecting **Set as Startup Project**. It might already have been automatically set as the startup project. Once set, press `F5` or (`Ctrl` + `F5` to run without debugging) to run the app.

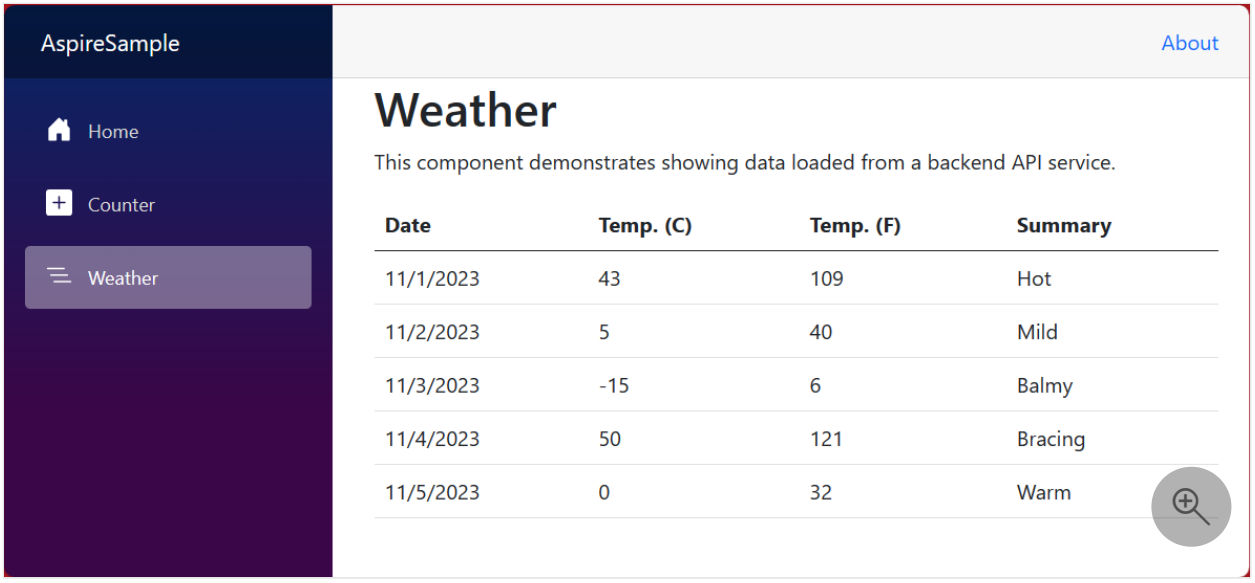
1. The app displays the .NET Aspire dashboard in the browser. You look at the dashboard in more detail later. For now, find the **webfrontend** project in the list of resources and select the project's **localhost** endpoint.



   The home page of the **webfrontend** app displays "Hello, world!"

2. Navigate from the home page to the weather page in the using the left side navigation. The weather page displays weather data. Make a mental note of some of the values represented in the forecast table.

3. Continue occasionally refreshing the page for 10 seconds. Within 10 seconds, the cached data is returned. Eventually, a different set of weather data appears, since the data is randomly generated and the cache is updated.



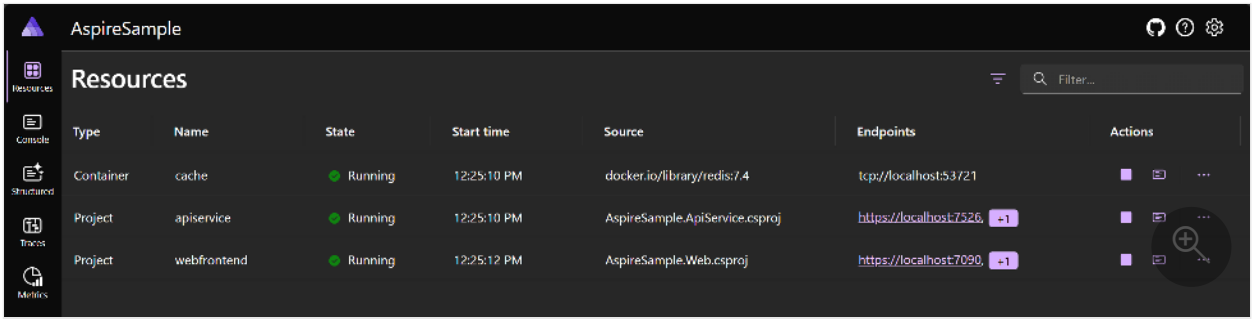
🤓 Congratulations! You created and ran your first .NET Aspire solution! To stop the app, close the browser window.

To stop the app in Visual Studio, select the **Stop Debugging** from the **Debug** menu.

Next, investigate the structure and other features of your new .NET Aspire solution.

Explore the .NET Aspire dashboard

When you run a .NET Aspire project, a dashboard launches that you use to monitor various parts of your app. The dashboard resembles the following screenshot:

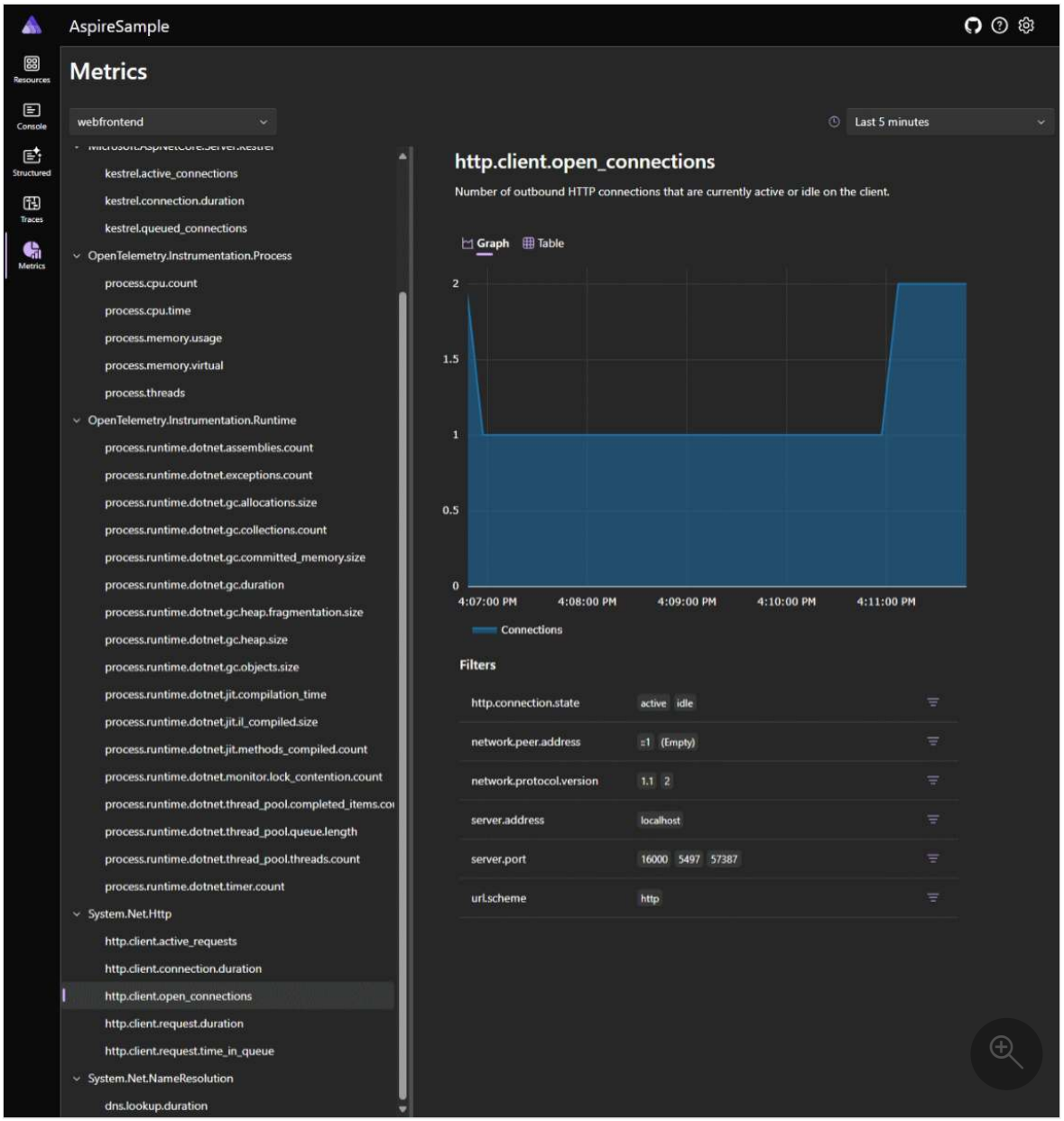


Visit each page using the left navigation to view different information about the .NET Aspire resources:

- **Resources**: Lists basic information for all of the individual .NET projects in your .NET Aspire project, such as the app state, endpoint addresses, and the environment variables that were loaded in.

- **Console**: Displays the console output from each of the projects in your app.
- **Structured**: Displays structured logs in table format. These logs support basic filtering, free-form search, and log level filtering as well. You should see logs from the `apiservice` and the `webfrontend`. You can expand the details of each log entry by selecting the **View** button on the right end of the row.
- **Traces**: Displays the traces for your application, which can track request paths through your apps. Locate a request for **/weather** and select **View** on the right side of the page. The dashboard should display the request in stages as it travels through the different parts of your app.

- **Metrics**: Displays various instruments and meters that are exposed and their corresponding dimensions for your app. Metrics conditionally expose filters based on their available dimensions.



For more information, see .NET Aspire dashboard overview.

Understand the .NET Aspire solution structure

The solution consists of the following projects:

- **AspireSample.ApiService**: An ASP.NET Core Minimal API project is used to provide data to the front end. This project depends on the shared **AspireSample.ServiceDefaults** project.
- **AspireSample.AppHost**: An orchestrator project designed to connect and configure the different projects and services of your app. The orchestrator should be set as the *Startup project*, and it depends on the **AspireSample.ApiService** and **AspireSample.Web** projects.
- **AspireSample.ServiceDefaults**: A .NET Aspire shared project to manage configurations that are reused across the projects in your solution related to resilience, service discovery, and telemetry.

- **AspireSample.Web**: An ASP.NET Core Blazor App project with default .NET Aspire service configurations, this project depends on the **AspireSample.ServiceDefaults** project. For more information, see .NET Aspire service defaults.

Your *AspireSample* directory should resemble the following structure:

Directory

```
└──📂 AspireSample
    ├──📂 AspireSample.ApiService
    │   ├──📂 Properties
    │   │   └── launchSettings.json
    │   ├── appsettings.Development.json
    │   ├── appsettings.json
    │   ├── AspireSample.ApiService.csproj
    │   └── Program.cs
    ├──📂 AspireSample.AppHost
    │   ├──📂 Properties
    │   │   └── launchSettings.json
    │   ├── appsettings.Development.json
    │   ├── appsettings.json
    │   ├── AspireSample.AppHost.csproj
    │   └── Program.cs
    ├──📂 AspireSample.ServiceDefaults
    │   ├── AspireSample.ServiceDefaults.csproj
    │   └── Extensions.cs
    ├──📂 AspireSample.Web
    │   ├──📂 Components
    │   │   ├──📂 Layout
    │   │   │   ├── MainLayout.razor
    │   │   │   ├── MainLayout.razor.css
    │   │   │   ├── NavMenu.razor
    │   │   │   └── NavMenu.razor.css
    │   │   ├──📂 Pages
    │   │   │   ├── Counter.razor
    │   │   │   ├── Error.razor
    │   │   │   ├── Home.razor
    │   │   │   └── Weather.razor
    │   │   ├── _Imports.razor
    │   │   ├── App.razor
    │   │   └── Routes.razor
    │   ├──📂 Properties
    │   │   └── launchSettings.json
    │   ├──📂 wwwroot
    │   │   ├──📂 bootstrap
    │   │   │   ├── bootstrap.min.css
    │   │   │   └── bootstrap.min.css.map
    │   │   ├── app.css
    │   │   └── favicon.png
    │   ├── appsettings.Development.json
    │   ├── appsettings.json
    │   ├── AspireSample.Web.csproj
    │   ├── Program.cs
```

```
│   └── WeatherApiClient.cs
└── AspireSample.sln
```

Explore the starter projects

Each project in an .NET Aspire solution plays a role in the composition of your app. The **.Web* project is a standard ASP.NET Core Blazor App that provides a front end UI. For more information, see What's new in ASP.NET Core 9.0: Blazor. The **.ApiService* project is a standard ASP.NET Core Minimal API template project. Both of these projects depend on the **.ServiceDefaults* project, which is a shared project that's used to manage configurations that are reused across projects in your solution.

The two projects of interest in this quickstart are the **.AppHost* and **.ServiceDefaults* projects detailed in the following sections.

.NET Aspire host project

The **.AppHost* project is responsible for acting as the orchestrator, and sets the `IsAspireHost` property of the project file to `true`:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <Sdk Name="Aspire.AppHost.Sdk" Version="9.1.0" />

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net9.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <IsAspireHost>true</IsAspireHost>
    <UserSecretsId>2aa31fdb-0078-4b71-b953-d23432af8a36</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference
Include="..\AspireSample.ApiService\AspireSample.ApiService.csproj" />
    <ProjectReference Include="..\AspireSample.Web\AspireSample.Web.csproj"
/>
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Aspire.Hosting.AppHost" Version="9.1.0" />
    <PackageReference Include="Aspire.Hosting.Redis" Version="9.1.0" />
  </ItemGroup>
```

```
</Project>
```

For more information, see .NET Aspire orchestration overview and .NET Aspire SDK.

Consider the *Program.cs* file of the *AspireSample.AppHost* project:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache");

var apiService = builder.AddProject<Projects.AspireSample_ApiService>
("apiservice");

builder.AddProject<Projects.AspireSample_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(cache)
    .WaitFor(cache)
    .WithReference(apiService)
    .WaitFor(apiService);

builder.Build().Run();
```

If you've used either the .NET Generic Host or the ASP.NET Core Web Host before, the app host programming model and builder pattern should be familiar to you. The preceding code:

- Creates an IDistributedApplicationBuilder instance from calling DistributedApplication.CreateBuilder().
- Calls AddRedis with the name `"cache"` to add a Redis server to the app, assigning the returned value to a variable named `cache`, which is of type `IResourceBuilder<RedisResource>`.
- Calls AddProject given the generic-type parameter with the project's details, adding the `AspireSample.ApiService` project to the application model. This is one of the fundamental building blocks of .NET Aspire, and it's used to configure service discovery and communication between the projects in your app. The name argument `"apiservice"` is used to identify the project in the application model, and used later by projects that want to communicate with it.
- Calls `AddProject` again, this time adding the `AspireSample.Web` project to the application model. It also chains multiple calls to WithReference passing the `cache` and `apiService` variables. The `WithReference` API is another fundamental API of .NET Aspire, which injects either service discovery information or connection string configuration into the project being added to the application model. Additionally,

calls to the `WaitFor` API are used to ensure that the `cache` and `apiService` resources are available before the `AspireSample.Web` project is started. For more information, see .NET Aspire orchestration: Waiting for resources.

Finally, the app is built and run. The DistributedApplication.Run() method is responsible for starting the app and all of its dependencies. For more information, see .NET Aspire orchestration overview.

> **Tip**
>
> The call to **AddRedis** creates a local Redis container for the app to use. If you'd rather simply point to an existing Redis instance, you can use the `AddConnectionString` method to reference an existing connection string. For more information, see **Reference existing resources**.

.NET Aspire service defaults project

The **.ServiceDefaults* project is a shared project that's used to manage configurations that are reused across the projects in your solution. This project ensures that all dependent services share the same resilience, service discovery, and OpenTelemetry configuration. A shared .NET Aspire project file contains the `IsAspireSharedProject` property set as `true`:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <IsAspireSharedProject>true</IsAspireSharedProject>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />

    <PackageReference Include="Microsoft.Extensions.Http.Resilience"
Version="9.3.0" />
    <PackageReference Include="Microsoft.Extensions.ServiceDiscovery"
Version="9.1.0" />
    <PackageReference Include="OpenTelemetry.Exporter.OpenTelemetryProtocol"
Version="1.11.2" />
    <PackageReference Include="OpenTelemetry.Extensions.Hosting"
Version="1.11.2" />
    <PackageReference Include="OpenTelemetry.Instrumentation.AspNetCore"
```

```
Version="1.11.1" />
    <PackageReference Include="OpenTelemetry.Instrumentation.Http"
Version="1.11.1" />
    <PackageReference Include="OpenTelemetry.Instrumentation.Runtime"
Version="1.11.1" />
  </ItemGroup>

</Project>
```

The service defaults project exposes an extension method on the IHostApplicationBuilder type, named `AddServiceDefaults`. The service defaults project from the template is a starting point, and you can customize it to meet your needs. For more information, see .NET Aspire service defaults.

Orchestrate service communication

.NET Aspire provides orchestration features to assist with configuring connections and communication between the different parts of your app. The *AspireSample.AppHost* project added the *AspireSample.ApiService* and *AspireSample.Web* projects to the application model. It also declared their names as `"webfrontend"` for Blazor front end, `"apiservice"` for the API project reference. Additionally, a Redis server resource labeled `"cache"` was added. These names are used to configure service discovery and communication between the projects in your app.

The front end app defines a typed HttpClient that's used to communicate with the API project.

C#

```
namespace AspireSample.Web;

public class WeatherApiClient(HttpClient httpClient)
{
    public async Task<WeatherForecast[]> GetWeatherAsync(
        int maxItems = 10,
        CancellationToken cancellationToken = default)
    {
        List<WeatherForecast>? forecasts = null;

        await foreach (var forecast in
            httpClient.GetFromJsonAsAsyncEnumerable<WeatherForecast>(
                "/weatherforecast", cancellationToken))
        {
            if (forecasts?.Count >= maxItems)
            {
                break;
            }
```

```
            if (forecast is not null)
            {
                forecasts ??= [];
                forecasts.Add(forecast);
            }
        }

        return forecasts?.ToArray() ?? [];
    }
}

public record WeatherForecast(DateOnly Date, int TemperatureC, string?
Summary)
{
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
}
```

The `HttpClient` is configured to use service discovery. Consider the following code from the *Program.cs* file of the *AspireSample.Web* project:

C#

```
using AspireSample.Web;
using AspireSample.Web.Components;

var builder = WebApplication.CreateBuilder(args);

// Add service defaults & Aspire client integrations.
builder.AddServiceDefaults();
builder.AddRedisOutputCache("cache");

// Add services to the container.
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();

builder.Services.AddHttpClient<WeatherApiClient>(client =>
    {
        // This URL uses "https+http://" to indicate HTTPS is preferred over
HTTP.
        // Learn more about service discovery scheme resolution at
https://aka.ms/dotnet/sdschemes.
        client.BaseAddress = new("https+http://apiservice");
    });

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}
```

```
app.UseHttpsRedirection();

app.UseAntiforgery();

app.UseOutputCache();

app.MapStaticAssets();

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();

app.MapDefaultEndpoints();

app.Run();
```

The preceding code:

- Calls `AddServiceDefaults`, configuring the shared defaults for the app.
- Calls AddRedisOutputCache with the same `connectionName` that was used when adding the Redis container `"cache"` to the application model. This configures the app to use Redis for output caching.
- Calls AddHttpClient and configures the HttpClient.BaseAddress to be `"https+http://apiservice"`. This is the name that was used when adding the API project to the application model, and with service discovery configured, it automatically resolves to the correct address to the API project.

For more information, see Make HTTP requests with the HttpClient class.

See also

- .NET Aspire integrations overview
- Service discovery in .NET Aspire
- .NET Aspire service defaults
- Health checks in .NET Aspire
- .NET Aspire telemetry
- Troubleshoot untrusted localhost certificate in .NET Aspire

Next steps

Tutorial: Add .NET Aspire to an existing .NET app

Tutorial: Add .NET Aspire to an existing .NET app

Article • 03/03/2025

If you have existing microservices and .NET web app, you can add .NET Aspire to it and get all the included features and benefits. In this article, you add .NET Aspire orchestration to a simple, preexisting .NET 9 project. You learn how to:

- ✓ Understand the structure of the existing microservices app.
- ✓ Enroll existing projects in .NET Aspire orchestration.
- ✓ Understand the changes enrollment makes in the projects.
- ✓ Start the .NET Aspire project.

Prerequisites

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0
- An OCI compliant container runtime, such as:
  - Docker Desktop or Podman. For more information, see Container runtime.
- An Integrated Developer Environment (IDE) or code editor, such as:
  - Visual Studio 2022 version 17.9 or higher (Optional)
  - Visual Studio Code (Optional)
    - C# Dev Kit: Extension (Optional)
  - JetBrains Rider with .NET Aspire plugin (Optional)

For more information, see .NET Aspire setup and tooling, and .NET Aspire SDK.

Get started

Let's start by obtaining the code for the solution:

1. Open a command prompt and change directories to where you want to store the code.

2. To clone to .NET 9 example solution, use the following `git clone` command:

   Bash

   ```
   git clone https://github.com/MicrosoftDocs/mslearn-dotnet-cloudnative-
   ```

```
devops.git eShopLite
```

Explore the sample app

This article uses a .NET 9 solution with three projects:

- **Data Entities**: This project is an example class library. It defines the `Product` class used in the Web App and Web API.
- **Products**: This example Web API returns a list of products in the catalog and their properties.
- **Store**: This example Blazor Web App displays the product catalog to website visitors.

Open and start debugging the project to examine its default behavior:

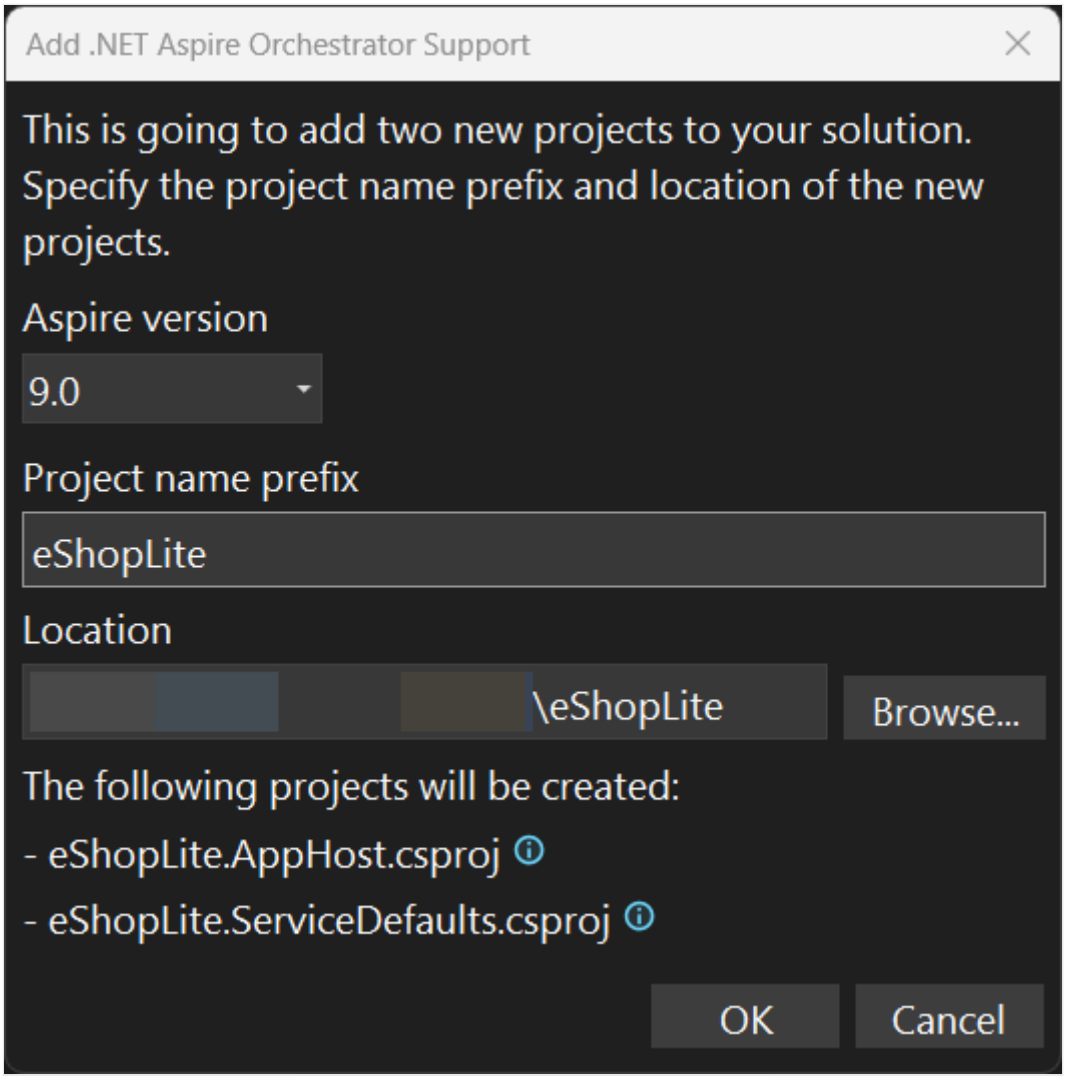
1. Start Visual Studio and then select **File** > **Open** > **Project/Solution**.
2. Navigate to the top level folder of the solution you cloned, select **eShopLite.sln**, and then select **Open**.
3. In the **Solution Explorer**, right-click the **eShopLite** solution, and then select **Configure Startup Projects**.
4. Select **Multiple startup projects**.
5. In the **Action** column, select **Start** for both the **Products** and **Store** projects.
6. Select **OK**.
7. To start debugging the solution, press `F5` or select **Start**.
8. Two pages open in the browser:
    - A page displays products in JSON format from a call to the Products Web API.
    - A page displays the homepage of the website. In the menu on the left, select **Products** to see the catalog obtained from the Web API.
9. To stop debugging, close the browser.

Add .NET Aspire to the Store web app

Now, let's enroll the **Store** project, which implements the web user interface, in .NET Aspire orchestration:

1. In Visual Studio, in the **Solution Explorer**, right-click the **Store** project, select **Add**, and then select **.NET Aspire Orchestrator Support**.

2. In the **Add .NET Aspire Orchestrator Support** dialog, select **OK**.



You should now have two new projects, both added to the solution:

- **eShopLite.AppHost**: An orchestrator project designed to connect and configure the different projects and services of your app. The orchestrator is set as the *Startup project*, and it depends on the **eShopLite.Store** project.
- **eShopLite.ServiceDefaults**: A .NET Aspire shared project to manage configurations that are reused across the projects in your solution related to resilience, service discovery, and telemetry.

In the **eShopLite.AppHost** project, open the *Program.cs* file. Notice this line of code, which registers the **Store** project in the .NET Aspire orchestration:

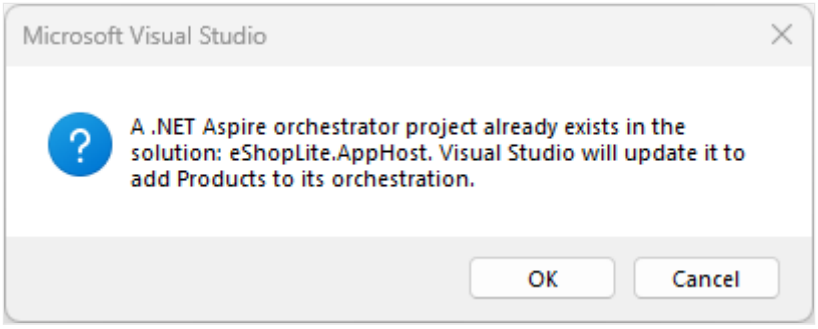
```
C#
builder.AddProject<Projects.Store>("store");
```

For more information, see AddProject.

To add the **Products** project to .NET Aspire:

1. In Visual Studio, in the **Solution Explorer**, right-click the **Products** project, select **Add**, and then select **.NET Aspire Orchestrator Support**.

2. A dialog indicating that .NET Aspire Orchestrator project already exists, select **OK**.



In the **eShopLite.AppHost** project, open the *Program.cs* file. Notice this line of code, which registers the **Products** project in the .NET Aspire orchestration:

C#

```
builder.AddProject<Projects.Products>("products");
```

Also notice that the **eShopLite.AppHost** project, now depends on both the **Store** and **Products** projects.

Service Discovery

At this point, both projects are part of .NET Aspire orchestration, but the *Store* project needs to rely on the **Products** backend address through .NET Aspire's service discovery. To enable service discovery, open the *Program.cs* file in **eShopLite.AppHost** project and update the code so that the `builder` adds a reference to the *Products* project:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var products = builder.AddProject<Projects.Products>("products");

builder.AddProject<Projects.Store>("store")
       .WithExternalHttpEndpoints()
       .WithReference(products);

builder.Build().Run();
```

The preceding code expresses that the *Store* project depends on the *Products* project. For more information, see .NET Aspire app host: Reference resources. This reference is used to discover the address of the *Products* project at run time. Additionally, the *Store* project is configured to use external HTTP endpoints. If you later choose to deploy this app, you'd need the call to WithExternalHttpEndpoints to ensure that it's public to the outside world.

Next, update the *appsettings.json* in the *Store* project with the following JSON:

JSON

```
{
  "DetailedErrors": true,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ProductEndpoint": "http://products",
  "ProductEndpointHttps": "https://products"
}
```

The addresses for both the endpoints now uses the "products" name that was added to the orchestrator in the *app host*. These names are used to discover the address of the *Products* project.

Explore the enrolled app

Let's start the solution and examine the new behavior that .NET Aspire provides.

> ⓘ **Note**
>
> Notice that the **eShopLite.AppHost** project is the new startup project.

1. In Visual Studio, to start debugging, press F5 Visual Studio builds the projects.
2. If the **Start Docker Desktop** dialog appears, select **Yes**. Visual Studio starts the Docker engine and creates the necessary containers. When the deployment is complete, the .NET Aspire dashboard is displayed.
3. In the dashboard, select the endpoint for the **products** project. A new browser tab appears and displays the product catalog in JSON format.
4. In the dashboard, select the endpoint for the **store** project. A new browser tab appears and displays the home page for the web app.

5. In the menu on the left, select **Products**. The product catalog is displayed.
6. To stop debugging, close the browser.

Congratulations, you added .NET Aspire orchestration to your pre-existing web app. You can now add .NET Aspire integrations and use the .NET Aspire tooling to streamline your cloud-native web app development.

.NET Aspire setup and tooling

Article • 03/15/2025

.NET Aspire includes tooling to help you create and configure cloud-native apps. The tooling includes useful starter project templates and other features to streamline getting started with .NET Aspire for Visual Studio, Visual Studio Code, and CLI workflows. In the sections ahead, you learn how to work with .NET Aspire tooling and explore the following tasks:

- ✓ Install .NET Aspire and its dependencies
- ✓ Create starter project templates using Visual Studio, Visual Studio Code, or the .NET CLI
- ✓ Install .NET Aspire integrations
- ✓ Work with the .NET Aspire dashboard

Install .NET Aspire prerequisites

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0.
- An OCI compliant container runtime, such as:
  - Docker Desktop or Podman. For more information, see Container runtime.
- An Integrated Developer Environment (IDE) or code editor, such as:
  - Visual Studio 2022 version 17.9 or higher (Optional)
  - Visual Studio Code (Optional)
    - C# Dev Kit: Extension (Optional)
  - JetBrains Rider with .NET Aspire plugin (Optional)

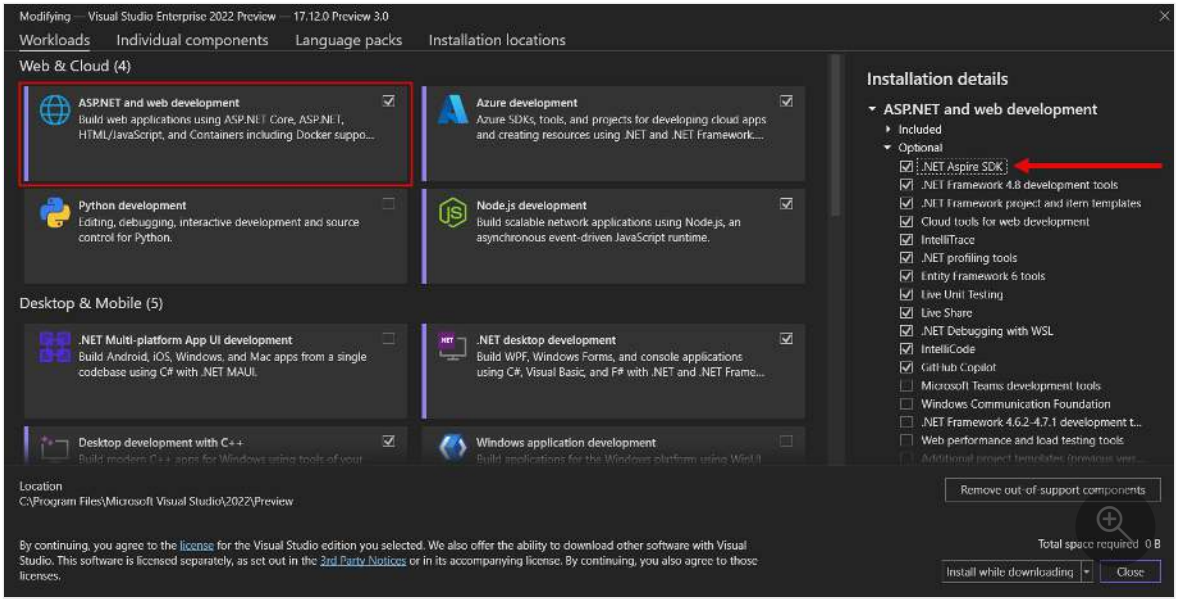
> **Tip**
>
> Alternatively, you can develop .NET Aspire solutions using **GitHub Codespaces** or **Dev Containers**.

Visual Studio 2022 17.9 or higher includes the latest .NET Aspire SDK by default when you install the Web & Cloud workload. If you have an earlier version of Visual Studio 2022, you can either upgrade to Visual Studio 2022 17.9 or you can install the .NET Aspire SDK using the following steps:

To install the .NET Aspire workload in Visual Studio 2022, use the Visual Studio installer.

1. Open the Visual Studio Installer.

2. Select **Modify** next to Visual Studio 2022.
3. Select the **ASP.NET and web development** workload.
4. On the **Installation details** panel, select **.NET Aspire SDK**.
5. Select **Modify** to install the .NET Aspire integration.



.NET Aspire templates

.NET Aspire provides a set of solution and project templates. These templates are available in your favorite .NET developer integrated environment. You can use these templates to create full .NET Aspire solutions, or add individual projects to existing .NET Aspire solutions.

Install the .NET Aspire templates

To install the .NET Aspire templates in Visual Studio, you need to manually install them unless you're using Visual Studio 17.12 or later. For Visual Studio 17.9 to 17.11, follow these steps:

1. Open Visual Studio.
2. Go to **Tools** > **NuGet Package Manager** > **Package Manager Console**.
3. Run the following command to install the templates:

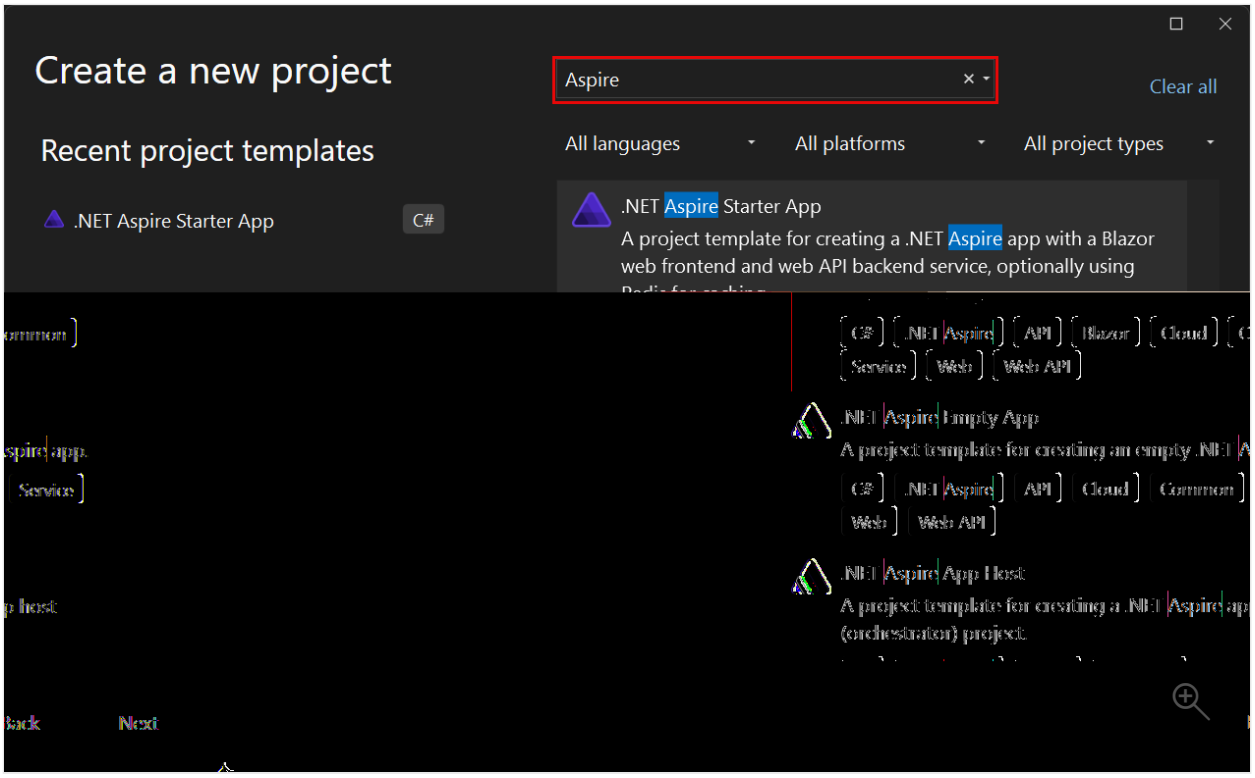
.NET CLI

```
dotnet new install Aspire.ProjectTemplates
```

For Visual Studio 17.12 or later, the .NET Aspire templates are installed automatically.

List the .NET Aspire templates

The .NET Aspire templates are installed automatically when you install Visual Studio 17.9 or later. To see what .NET Aspire templates are available, select **File** > **New** > **Project** in Visual Studio, and search for "Aspire" in the search bar (`Alt` + `S`). You'll see a list of available .NET Aspire project templates:



For more information, see .NET Aspire templates.

Container runtime

.NET Aspire projects are designed to run in containers. You can use either Docker Desktop or Podman as your container runtime. Docker Desktop is the most common container runtime. Podman is an open-source daemonless alternative to Docker, that can build and run Open Container Initiative (OCI) containers. If your host environment has both Docker and Podman installed, .NET Aspire defaults to using Docker. You can instruct .NET Aspire to use Podman instead, by setting the `DOTNET_ASPIRE_CONTAINER_RUNTIME` environment variable to `podman`:

Windows

PowerShell

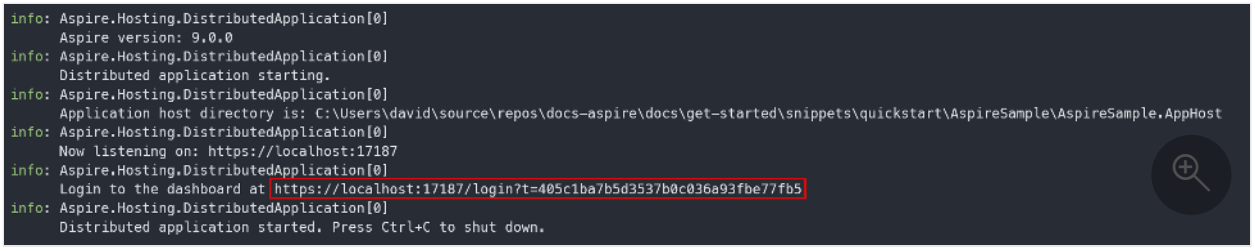
```
[System.Environment]::SetEnvironmentVariable("DOTNET_ASPIRE_CONTAINER_RUNTIME", "podman", "User")
```

For more information, see Install Podman on Windows.

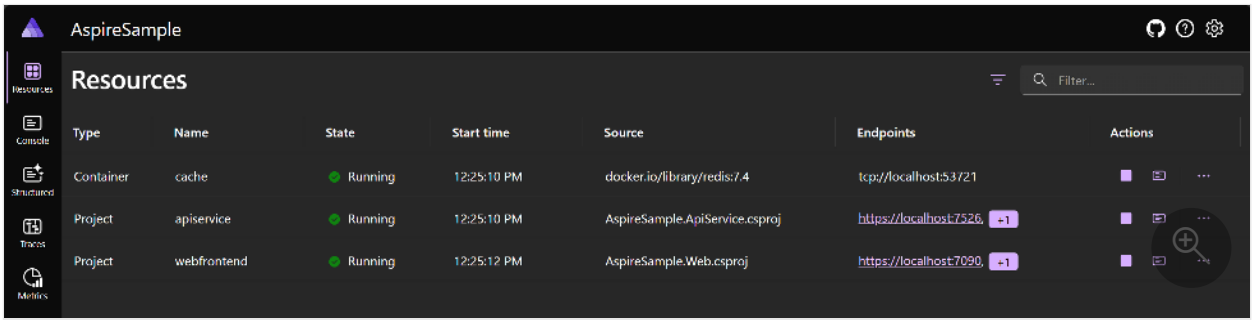
.NET Aspire dashboard

.NET Aspire templates that expose the app host project also include a useful developer dashboard that's used to monitor and inspect various aspects of your app, such as logs, traces, and environment configurations. This dashboard is designed to improve the local development experience and provides an overview of the overall state and structure of your app.

The .NET Aspire dashboard is only visible while the app is running and starts automatically when you start the *.*AppHost* project. Visual Studio and Visual Studio Code launch both your app and the .NET Aspire dashboard for you automatically in your browser. If you start the app using the .NET CLI, copy and paste the dashboard URL from the output into your browser, or hold `Ctrl` and select the link (if your terminal supports hyperlinks).



The left navigation provides links to the different parts of the dashboard, each of which you explore in the following sections.



The .NET Aspire dashboard is also available in a standalone mode. For more information, see Standalone .NET Aspire dashboard.

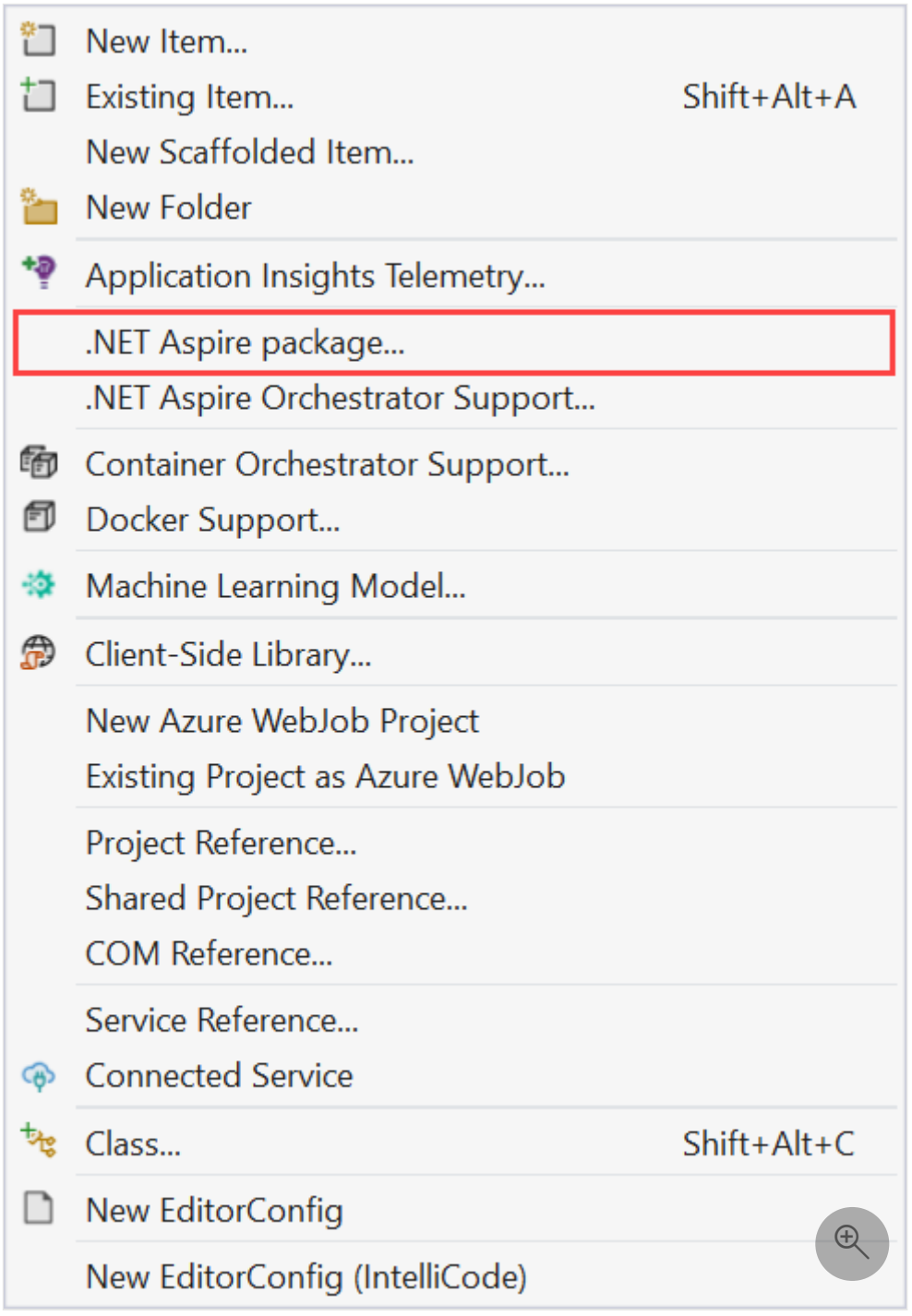
Visual Studio tooling

Visual Studio provides additional features for working with .NET Aspire integrations and the App Host orchestrator project. Not all of these features are currently available in Visual Studio Code or through the CLI.

Add an integration package

You add .NET Aspire integrations to your app like any other NuGet package using Visual Studio. However, Visual Studio also provides UI options to add .NET Aspire integrations directly.

1. In Visual Studio, right select on the project you want to add an .NET Aspire integration to and select **Add** > **.NET Aspire package...**.



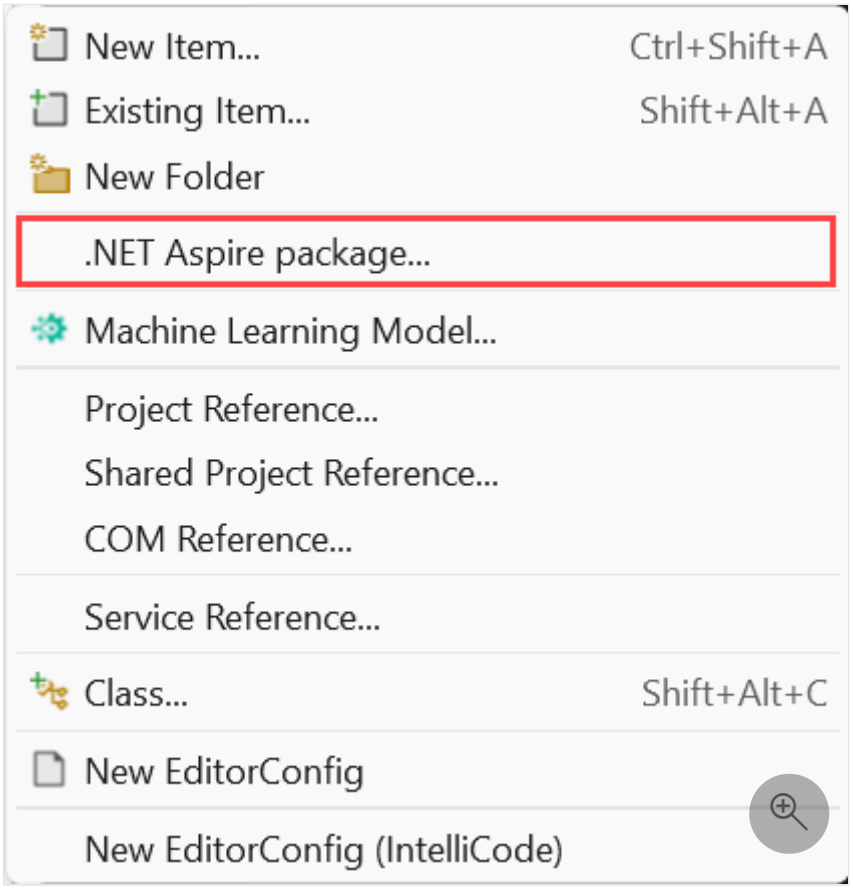
2. The package manager opens with search results preconfigured (populating filter criteria) for .NET Aspire integrations, allowing you to easily browse and select the desired integration.

For more information on .NET Aspire integrations, see .NET Aspire integrations overview.

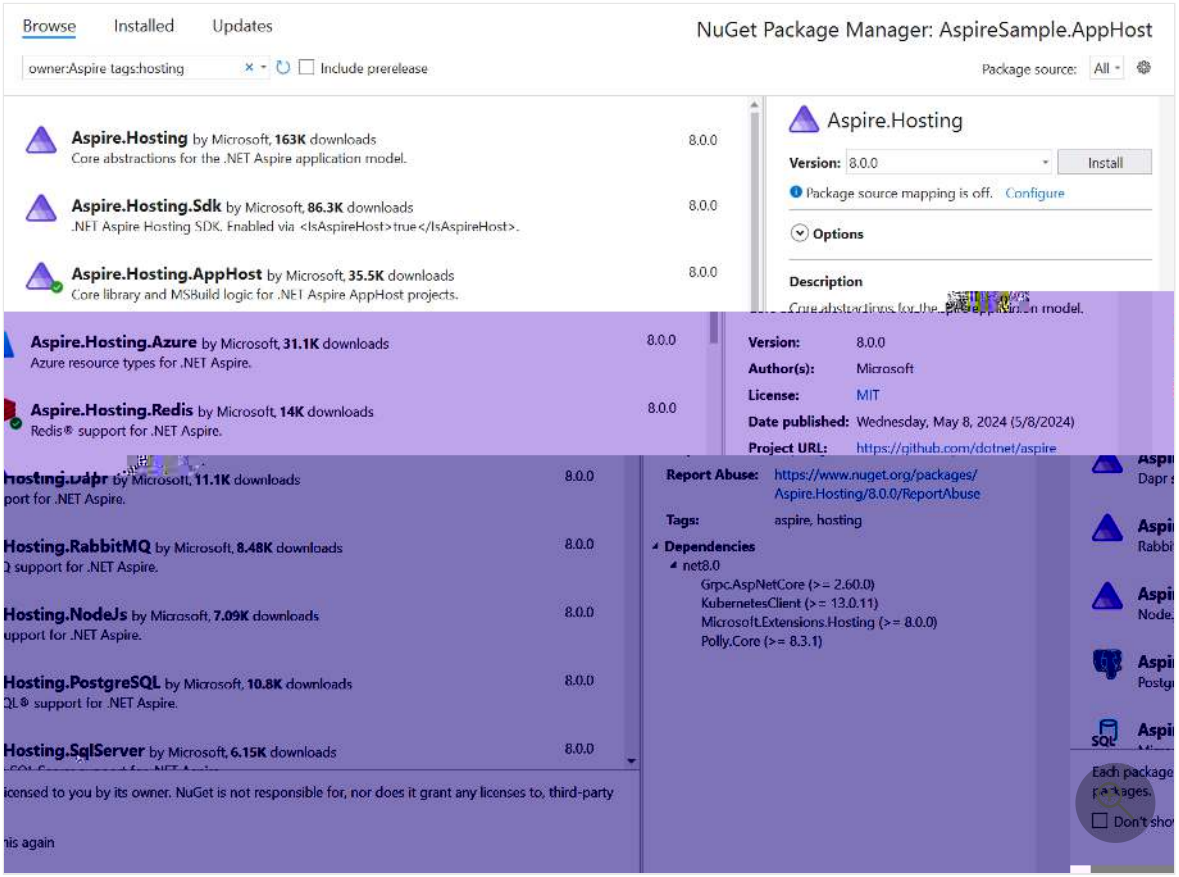
Add hosting packages

.NET Aspire hosting packages are used to configure various resources and dependencies an app may depend on or consume. Hosting packages are differentiated from other integration packages in that they're added to the *.*AppHost* project. To add a hosting package to your app, follow these steps:

1. In Visual Studio, right select on the *.*AppHost* project and select **Add** > **.NET Aspire package...**.



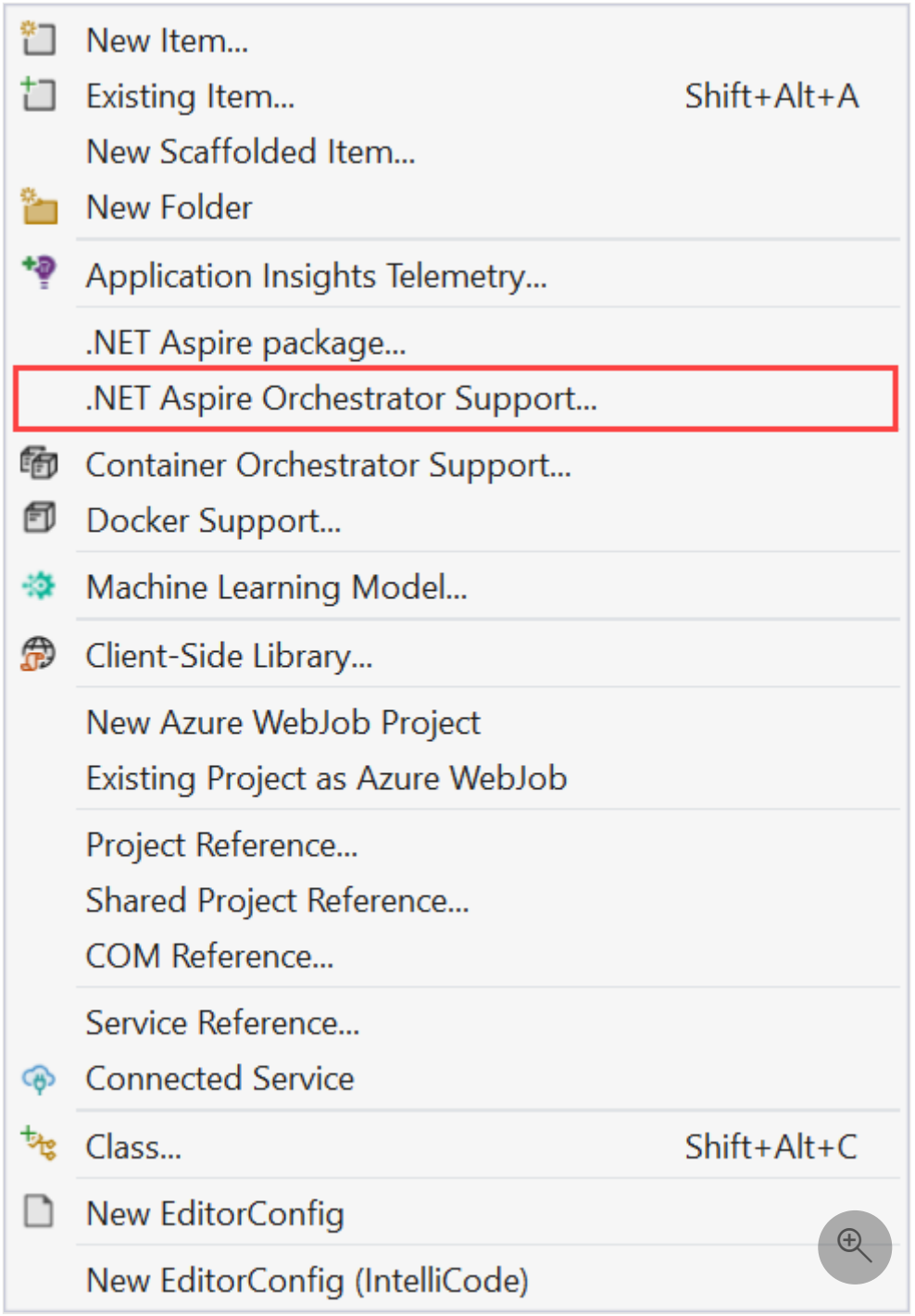
2. The package manager opens with search results preconfigured (populating filter criteria) for .NET Aspire hosting packages, allowing you to easily browse and select the desired package.



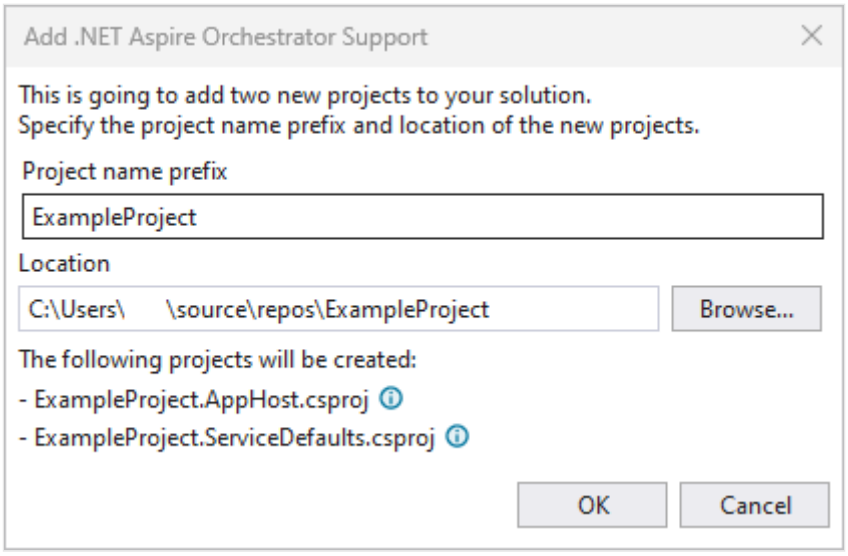
Add orchestration projects

You can add .NET Aspire orchestration projects to an existing app using the following steps:

1. In Visual Studio, right select on an existing project and select **Add** > **.NET Aspire Orchestrator Support...**



2. A dialog window opens with a summary of the **.AppHost* and **.ServiceDefaults* projects that are added to your solution.



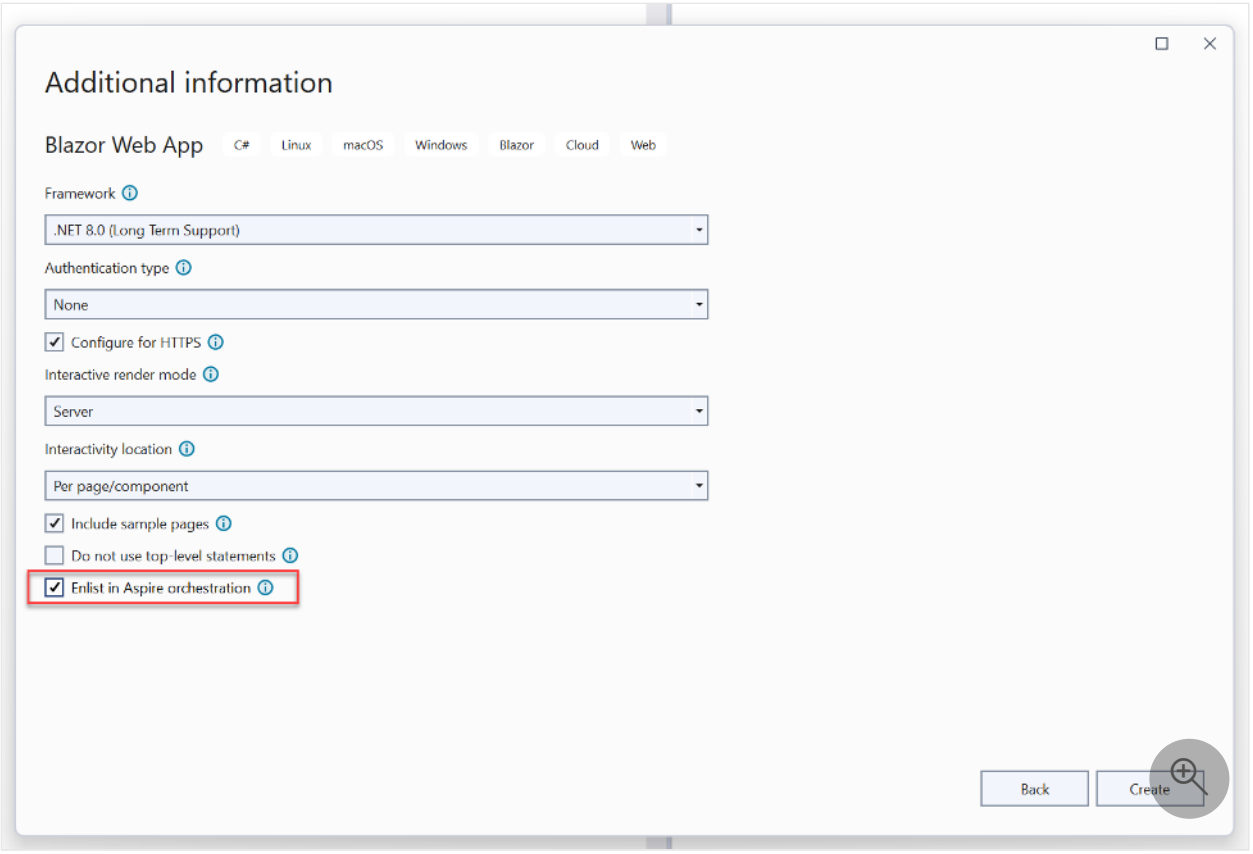
3. Select **OK** and the following changes are applied:

    - The **.AppHost* and **.ServiceDefaults* orchestration projects are added to your solution.
    - A call to `builder.AddServiceDefaults` will be added to the *Program.cs* file of your original project.
    - A reference to your original project will be added to the *Program.cs* file of the **.AppHost* project.

For more information on .NET Aspire orchestration, see .NET Aspire orchestration overview.

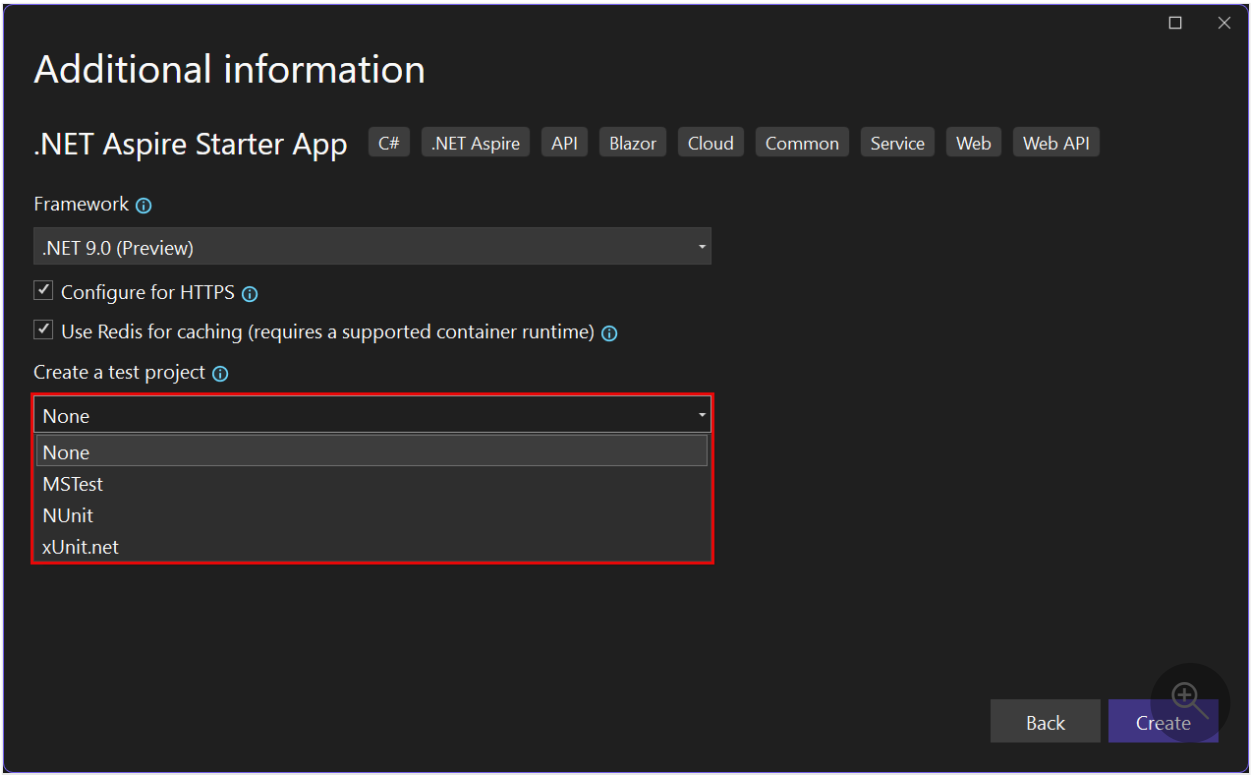
Enlist in orchestration

Visual Studio provides the option to **Enlist in Aspire orchestration** during the new project workflow. Select this option to have Visual Studio create **.AppHost* and **.ServiceDefaults* projects alongside your selected project template.



Create test project

When you're using Visual Studio, and you select the **.NET Aspire Start Application** template, you have the option to include a test project. This test project is an xUnit project that includes a sample test that you can use as a starting point for your tests.



For more information, see Write your first .NET Aspire test.

See also

- Unable to install .NET Aspire workload
- Use Dev Proxy with .NET Aspire project

.NET Aspire SDK

Article • 02/25/2025

The .NET Aspire SDK is intended for **.AppHost* projects, which serve as the .NET Aspire orchestrator. These projects are designated using the `<IsAspireHost>true</IsAspireHost>` property, as well as specifying the `Aspire.AppHost.Sdk` in the project file. The SDK provides a set of features that simplify the development of .NET Aspire apps.

Overview

The  Aspire.AppHost.Sdk is an additive MSBuild project SDK for building .NET Aspire apps. The `Aspire.AppHost.Sdk` is defined with a top-level `Project/Sdk`:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

    <Sdk Name="Aspire.AppHost.Sdk" Version="9.1.0" />

    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net9.0</TargetFramework>
        <IsAspireHost>true</IsAspireHost>
        <!-- Omitted for brevity -->
    </PropertyGroup>

    <ItemGroup>
        <PackageReference Include="Aspire.Hosting.AppHost" Version="9.1.0"
/>
    </ItemGroup>

    <!-- Omitted for brevity -->
</Project>
```

The preceding example project defines the top-level SDK as `Microsoft.NET.Sdk` and the `Aspire.AppHost.Sdk` as an additive SDK. The `IsAspireHost` property is set to `true` to indicate that this project is an .NET Aspire app host. The project also references the `Aspire.Hosting.AppHost` package which brings in a number of Aspire-related dependencies.

SDK Features

The .NET Aspire SDK provides several key features.

Project references

Each `ProjectReference` in the .NET Aspire app host project isn't treated as standard project references. Instead, they enable the *app host* to execute these projects as part of its orchestration. Each project reference triggers a generator to create a `class` that represents the project as an IProjectMetadata. This metadata is used to populate the named projects in the generated `Projects` namespace. When you call the Aspire.Hosting.ProjectResourceBuilderExtensions.AddProject API, the `Projects` namespace is used to reference the project—passing the generated class as a generic-type parameter.

> **Tip**
>
> If you need to reference a project in the tranditional way within the app host, set the `IsAspireProjectResource` attribute on the `ProjectReference` element to `false`, as shown in the following example:

XML

```xml
<ProjectReference Include="..\MyProject\MyProject.csproj"
IsAspireProjectResource="false" />
```

Orchestrator dependencies

The .NET Aspire SDK dynamically adds references to the .NET Aspire dashboard and other app host dependencies, such as the developer control plane (DCP) packages. These dependencies are specific to the platform that the app host is built on.

When the app host project runs, the orchestrator relies on these dependencies to provide the necessary functionality to the app host. For more information, see .NET Aspire orchestration overview.

.NET Aspire templates

Article • 03/15/2025

There are a number of .NET Aspire project templates available to you. You can use these templates to create full .NET Aspire solutions, or add individual projects to existing .NET Aspire solutions.

The .NET Aspire templates are available in the Aspire.ProjectTemplates NuGet package.

Available templates

The .NET Aspire templates allow you to create new apps pre-configured with the .NET Aspire solutions structure and default settings. These projects also provide a unified debugging experience across the different resources of your app.

.NET Aspire templates are available in two categories: solution templates and project templates. Solution templates create a new .NET Aspire solution with multiple projects, while project templates create individual projects that can be added to an existing .NET Aspire solution.

Solution templates

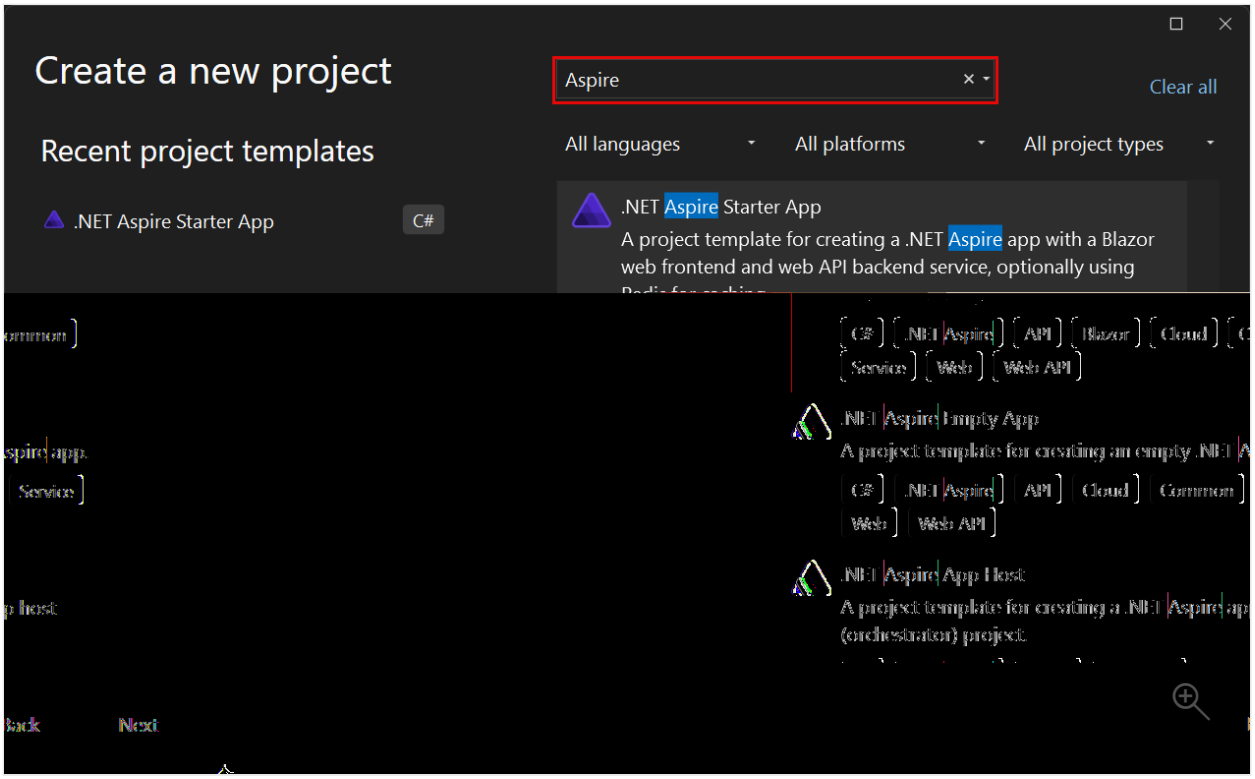
The following .NET Aspire solution templates are available, assume the solution is named *AspireSample*:

- **.NET Aspire Empty App**: A minimal .NET Aspire project that includes the following:
  - **AspireSample.AppHost**: An orchestrator project designed to connect and configure the different projects and services of your app.
  - **AspireSample.ServiceDefaults**: A .NET Aspire shared project to manage configurations that are reused across the projects in your solution related to resilience, service discovery, and telemetry.
- **.NET Aspire Starter App**: In addition to the **.AppHost** and **.ServiceDefaults** projects, the .NET Aspire Starter App also includes the following:
  - **AspireSample.ApiService**: An ASP.NET Core Minimal API project is used to provide data to the frontend. This project depends on the shared **AspireSample.ServiceDefaults** project.
  - **AspireSample.Web**: An ASP.NET Core Blazor App project with default .NET Aspire service configurations, this project depends on the **AspireSample.ServiceDefaults** project.

    - **AspireSample.Test**: Either an MSTest, NUnit, or xUnit test project with project references to the **AspireSample.AppHost** and an example *WebTests.cs* file demonstrating an integration test.

The following .NET Aspire project templates are available:

- **.NET Aspire App Host**: A standalone **.AppHost** project that can be used to orchestrate and manage the different projects and services of your app.
- **.NET Aspire Test projects**: These project templates are used to create test projects for your .NET Aspire app, and they're intended to represent functional and integration tests. The test projects include the following templates:
    - **MSTest**: A project that contains MSTest integration of a .NET Aspire AppHost project.
    - **NUnit**: A project that contains NUnit integration of a .NET Aspire AppHost project.



Follow the prompts to configure your project or solution from the template, and then select **Create**.

See also

- .NET Aspire SDK
- .NET Aspire setup and tooling
- Testing in .NET Aspire

.NET Aspire and GitHub Codespaces

Article • 02/25/2025

GitHub Codespaces offers a cloud-hosted development environment based on Visual Studio Code. It can be accessed directly from a web browser or through Visual Studio Code locally, where Visual Studio Code acts as a client connecting to a cloud-hosted backend. With .NET Aspire 9.1, comes logic to better support GitHub Codespaces including:

- Automatically configure port forwarding with the correct protocol.
- Automatically translate URLs in the .NET Aspire dashboard.

Before .NET Aspire 9.1 it was still possible to use .NET Aspire within a GitHub Codespace, however more manual configuration was required.

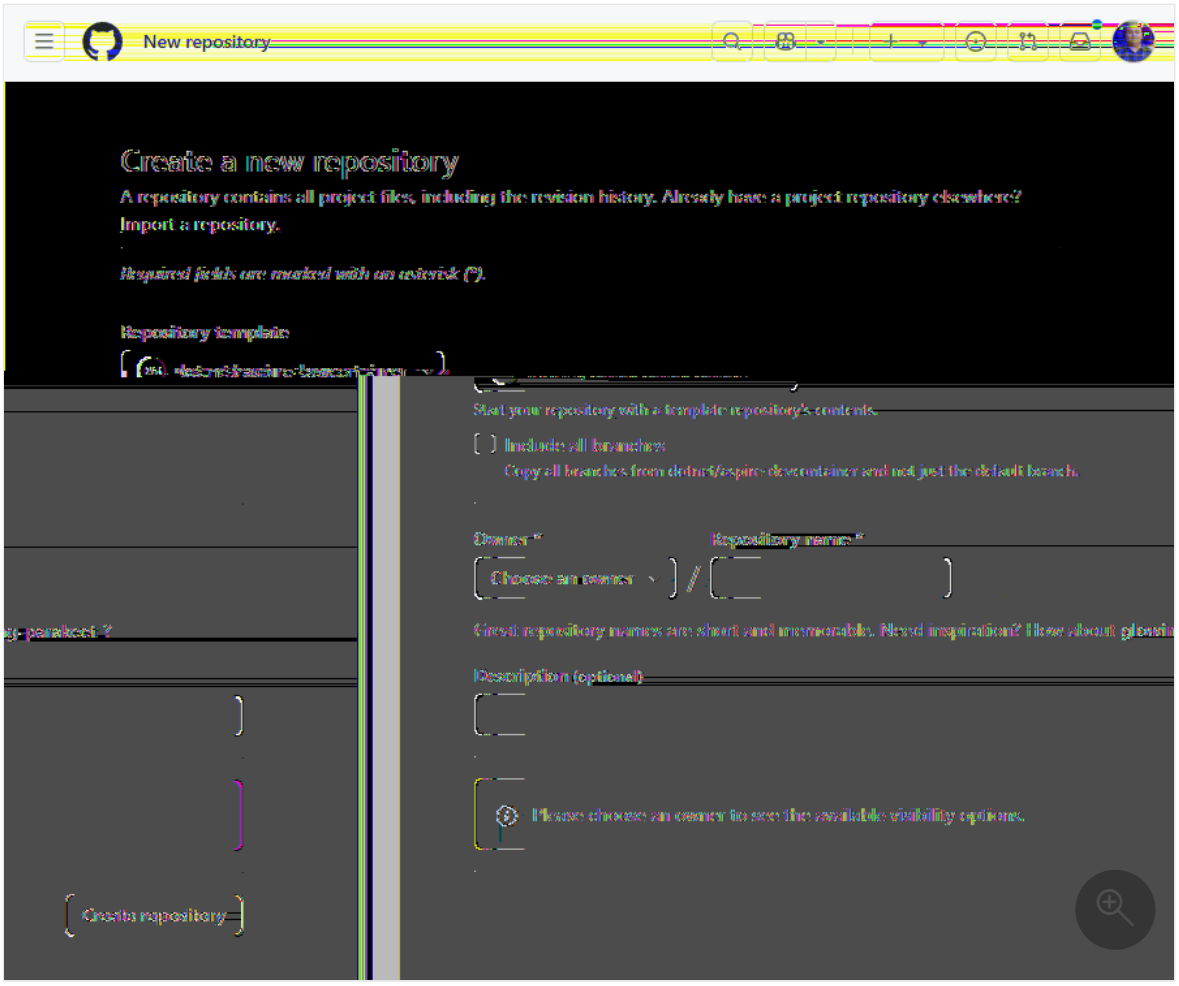
GitHub Codespaces vs. Dev Containers

GitHub Codespaces builds upon Visual Studio Code and the Dev Containers specification. In addition to supporting GitHub Codespaces, .NET Aspire 9.1 enhances support for using Visual Studio Code and locally hosted Dev Containers. While the experiences are similar, there are some differences. For more information, see .NET Aspire and Visual Studio Code Dev Containers.

Quick start using template repository

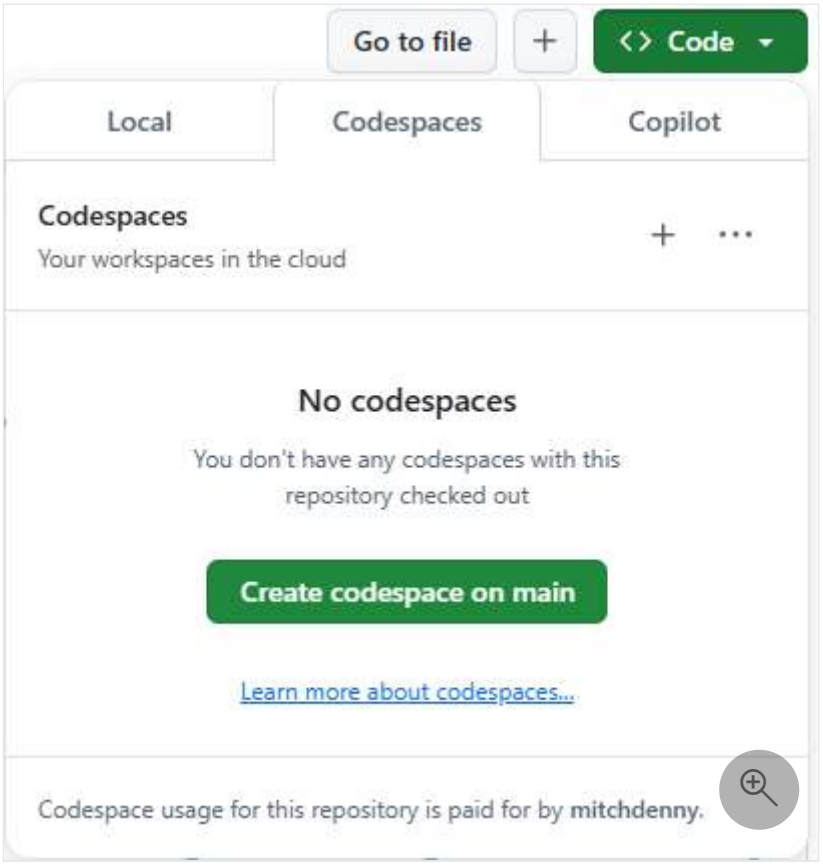
To configure GitHub Codespaces for .NET Aspire, use the *.devcontainer/devcontainer.json* file in your repository. The simplest way to get started is by creating a new repository from our template repository. Consider the following steps:

1. Create a new repository using our template.

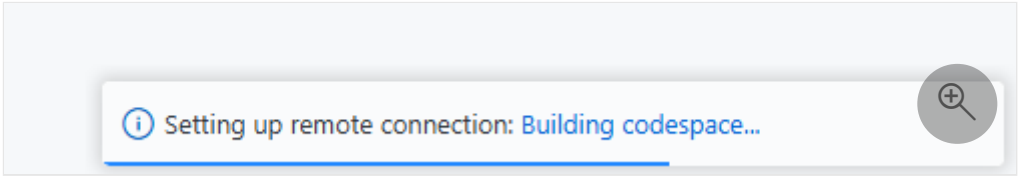


Once you provide the details and select **Create repository**, the repository is created and shown in GitHub.

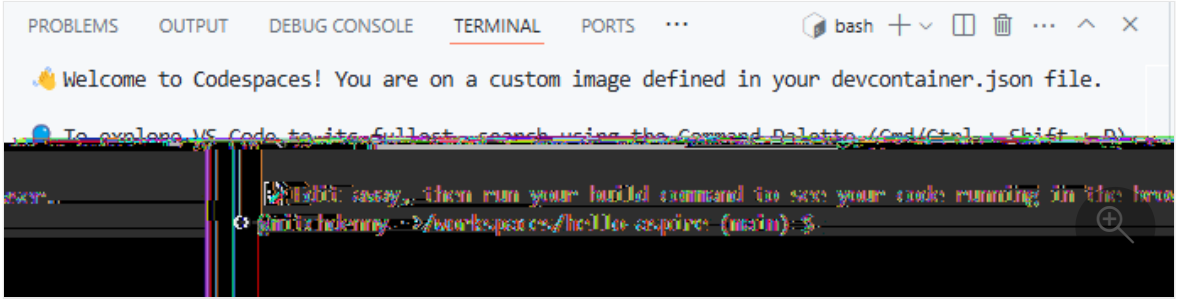
2. From the new repository, select on the Code button and select the Codespaces tab and then select **Create codespace on main**.



After you select **Create codespace on main**, you navigate to a web-based version of Visual Studio Code. Before you use the Codespace, the containerized development environment needs to be prepared. This process happens automatically on the server and you can review progress by selecting the **Building codespace** link on the notification in the bottom right of the browser window.



When the container image has finished being built the **Terminal** prompt appears which signals that the environment is ready to be interacted with.



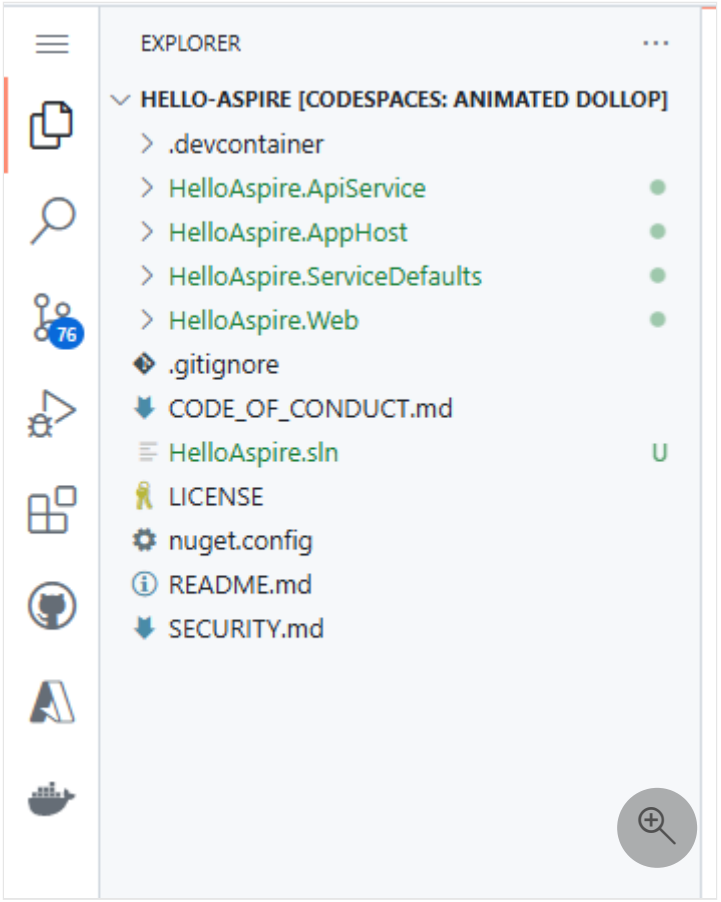
At this point, the .NET Aspire templates have been installed and the ASP.NET Core developer certificate has been added and accepted.

3. Create a new .NET Aspire project using the starter template.

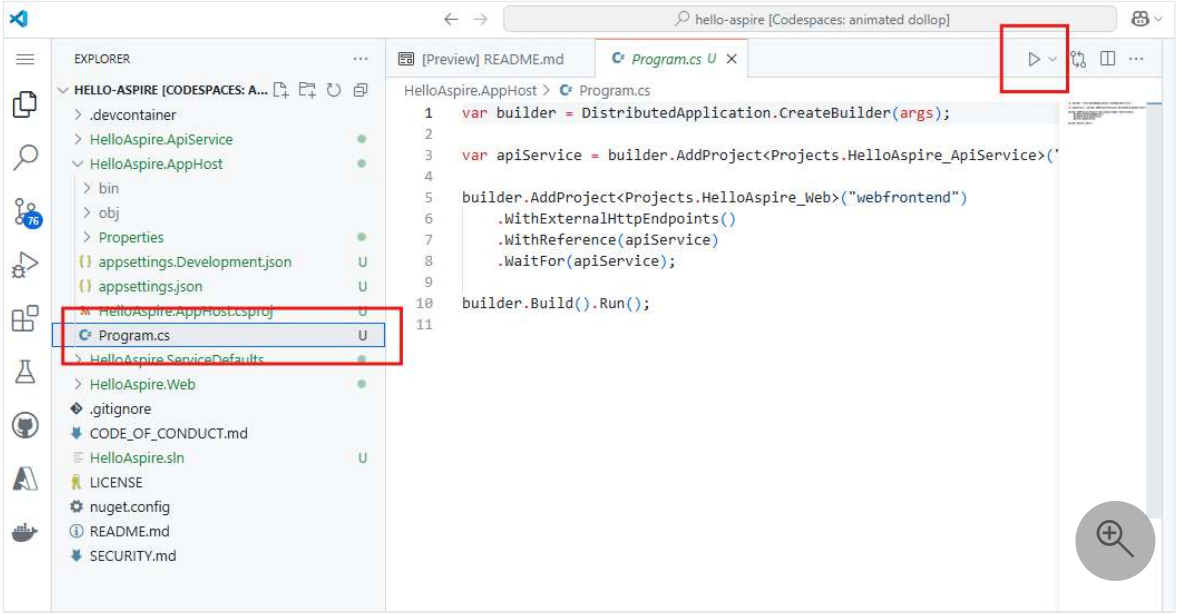
.NET CLI

```
dotnet new aspire-starter --name HelloAspire
```

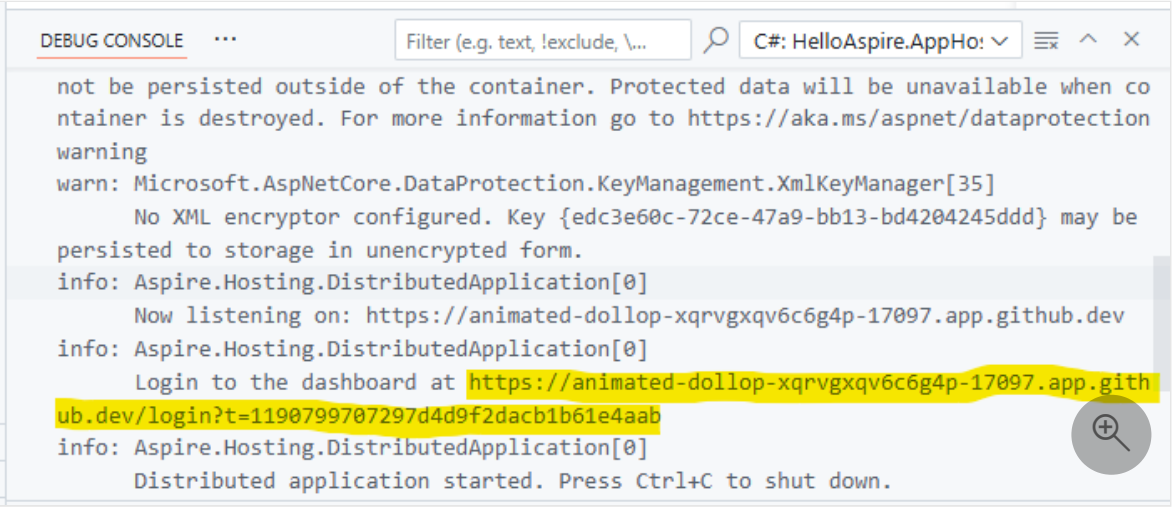
This results in many files and folders being created in the repository, which are visible in the **Explorer** panel on the left side of the window.



4. Launch the app host via the *HelloAspire.AppHost/Program.cs* file, by selecting the **Run project** button near the top-right corner of the **Tab bar**.

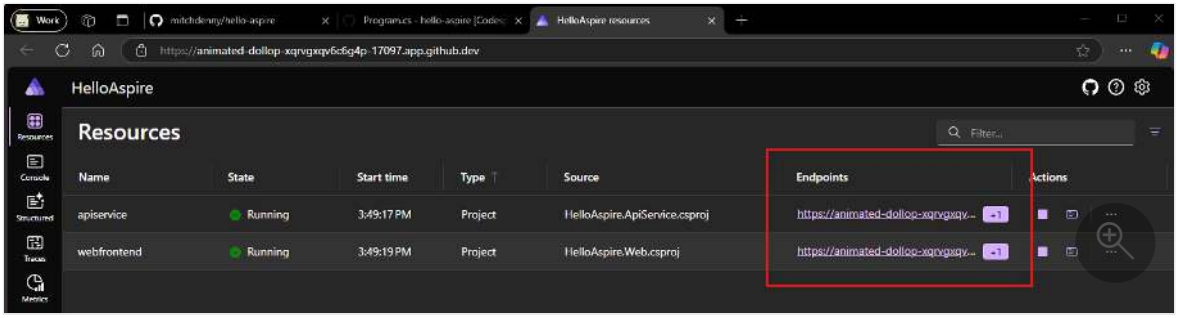


After a few moments the **Debug Console** panel is displayed, and it includes a link to the .NET Aspire dashboard exposed on a GitHub Codespaces endpoint with the authentication token.



5. Open the .NET Aspire dashboard by selecting the dashboard URL in the **Debug Console**. This opens the .NET Aspire dashboard in a separate tab within your browser.

   You notice on the dashboard that all HTTP/HTTPS endpoints defined on resources have had their typical `localhost` address translated to a unique fully qualified subdomain on the `app.github.dev` domain.



Traffic to each of these endpoints is automatically forwarded to the underlying process or container running within the Codespace. This includes development time tools such as PgAdmin and Redis Insight.

> **Note**
>
> In addition to the authentication token embedded within the URL of the dashboard link of the **Debug Console**, endpoints also require authentication via your GitHub identity to avoid port forwarded endpoints being accessible to everyone. For more information on port forwarding in GitHub Codespaces, see Forwarding ports in your codespace .

6. Commit changes to the GitHub repository.

   GitHub Codespaces doesn't automatically commit your changes to the branch you're working on in GitHub. You have to use the **Source Control** panel to stage and commit the changes and push them back to the repository.

   Working in a GitHub Codespace is similar to working with Visual Studio Code on your own machine. You can checkout different branches and push changes just like you normally would. In addition, you can easily spin up multiple Codespaces simultaneously if you want to quickly work on another branch without disrupting your existing debug session. For more information, see Developing in a codespace.

7. Clean up your Codespace.

   GitHub Codespaces are temporary development environments and while you might use one for an extended period of time, they should be considered a disposable resource that you recreate as needed (with all of the customization/setup contained within the *devcontainer.json* and associated configuration files).

   To delete your GitHub Codespace, visit the GitHub Codespaces page. This shows you a list of all of your Codespaces. From here you can perform management operations on each Codespace, including deleting them.

   GitHub charges for the use of Codespaces. For more information, see Managing the cost of GitHub Codespaces in your organization.

   > **Note**
   >
   > .NET Aspire supports the use of Dev Containers in Visual Studio Code independent of GitHub Codespaces. For more information on how to use Dev Containers locally, see .NET Aspire and Dev Containers in Visual Studio Code.

Manually configuring *devcontainer.json*

The preceding walkthrough demonstrates the streamlined process of creating a GitHub Codespace using the .NET Aspire Devcontainer template. If you already have an existing repository and wish to utilize Devcontainer functionality with .NET Aspire, add a *devcontainer.json* file to the *.devcontainer* folder within your repository:

Directory

```
└──📂 .devcontainer
    └── devcontainer.json
```

The template repository contains a copy of the *devcontainer.json* file that you can use as a starting point, which should be sufficient for .NET Aspire. The following JSON represents the latest version of the *.devcontainer/devcontainer.json* file from the template:

JSON

```
// For format details, see https://aka.ms/devcontainer.json. For config
options, see the
// README at:
https://github.com/devcontainers/templates/tree/main/src/dotnet
{
    "name": ".NET Aspire",
    // Or use a Dockerfile or Docker Compose file. More info:
https://containers.dev/guide/dockerfile
    "image": "mcr.microsoft.com/devcontainers/dotnet:9.0-bookworm",
    "features": {
        "ghcr.io/devcontainers/features/docker-in-docker:2": {},
        "ghcr.io/devcontainers/features/powershell:1": {},
    },

    "hostRequirements": {
        "cpus": 8,
        "memory": "32gb",
        "storage": "64gb"
    },

    // Use 'forwardPorts' to make a list of ports inside the container
available locally.
    // "forwardPorts": [5000, 5001],
    // "portsAttributes": {
    //      "5001": {
    //          "protocol": "https"
    //      }
    // }

    // Use 'postCreateCommand' to run commands after the container is
created.
    // "postCreateCommand": "dotnet restore",
    "onCreateCommand": "dotnet new install Aspire.ProjectTemplates::9.1.0 --
force",
    "postStartCommand": "dotnet dev-certs https --trust",
    "customizations": {
        "vscode": {
            "extensions": [
                "ms-dotnettools.csdevkit",
                "GitHub.copilot-chat",
```

```
                "GitHub.copilot"
            ]
        }
    }
    // Configure tool-specific properties.
    // "customizations": {},

    // Uncomment to connect as root instead. More info: https://aka.ms/dev-
containers-non-root.
    // "remoteUser": "root"
}
```

Speed up Codespace creation

Creating a GitHub Codespace can take some time as it prepares the underlying container image. To expedite this process, you can utilize *prebuilds* to significantly reduce the creation time to approximately 30-60 seconds (exact timing might vary). For more information on GitHub Codespaces prebuilds, see GitHub Codespaces prebuilds.

.NET Aspire and Visual Studio Code Dev Containers

Article • 02/25/2025

The Dev Containers Visual Studio Code extension provides a way for development teams to develop within a containerized environment where all dependencies are preconfigured. With .NET Aspire 9.1, there's added logic to better support working with .NET Aspire within a Dev Container environment by automatically configuring port forwarding.

Before .NET Aspire 9.1, it possible to use .NET Aspire within a Dev Container, however more manual configuration was required.

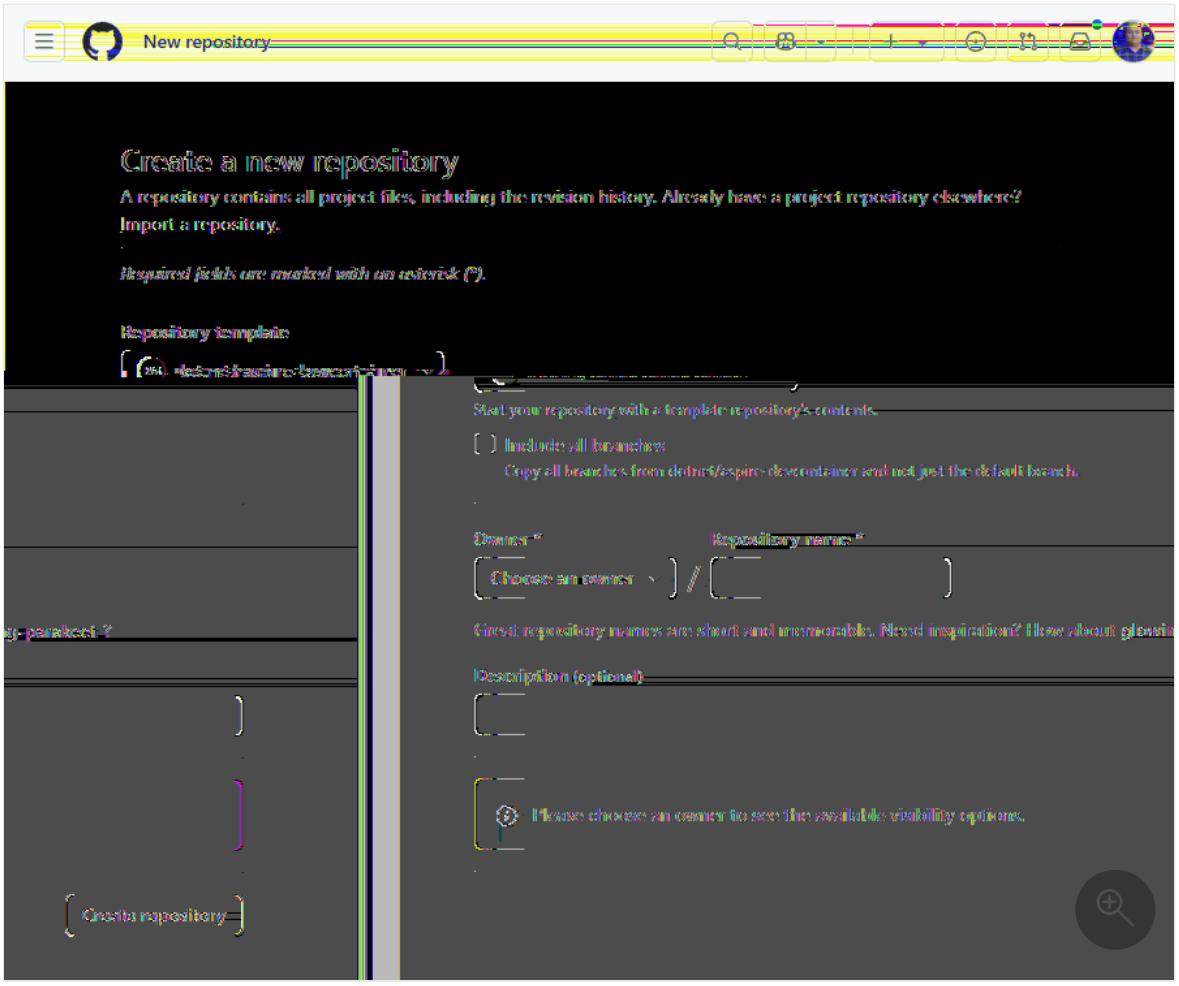
Dev Containers vs. GitHub Codespaces

Using Dev Containers in Visual Studio Code is similar to using GitHub Codespaces. With the release of .NET Aspire 9.1, support for both Dev Containers in Visual Studio Code and GitHub Codespaces was enhanced. Although the experiences are similar, there are some differences. For more information on using .NET Aspire with GitHub Codespaces, see .NET Aspire and GitHub Codespaces.

Quick start using template repository

To configure Dev Containers in Visual Studio Code, use the _.devcontainer/devcontainer.json file in your repository. The simplest way to get started is by creating a new repository from our template repository. Consider the following steps:

1. Create a new repository using our template.



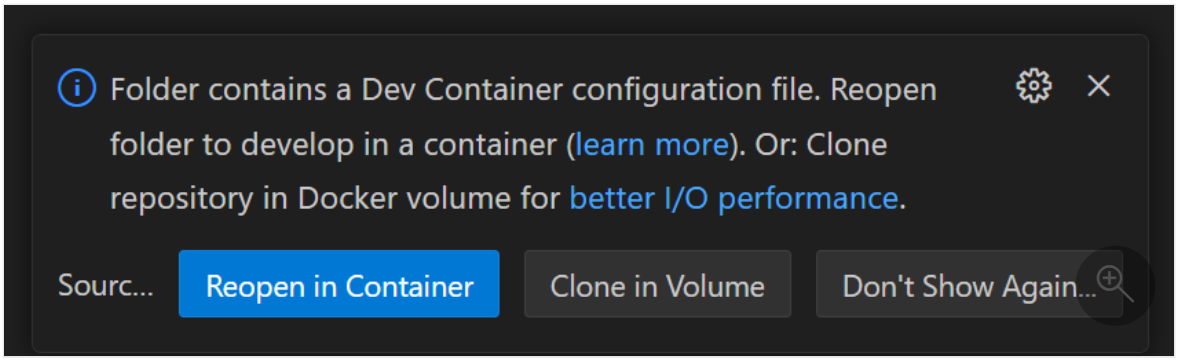
Once you provide the details and select **Create repository**, the repository is created and shown in GitHub.

2. Clone the repository to your local developer workstation using the following command:

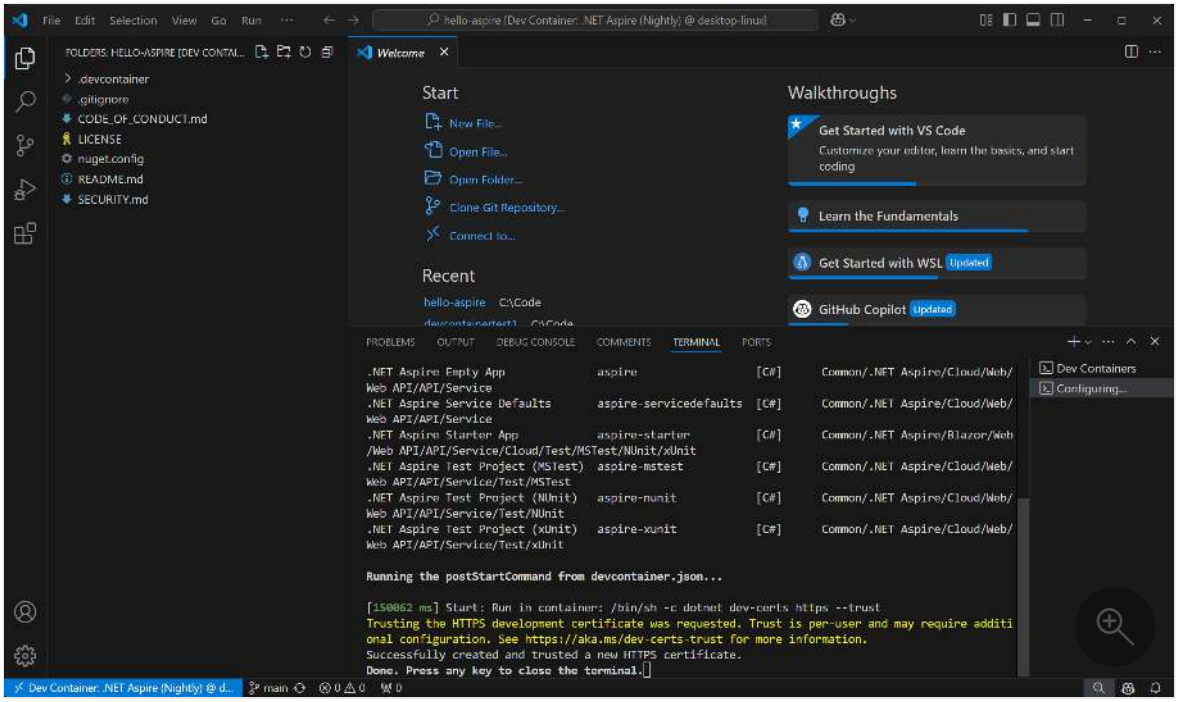
.NET CLI

```
git clone https://github.com/<org>/<username>/<repository>
```

3. Open the repository in Visual Studio Code. After a few moments Visual Studio Code detects the *.devcontainer/devcontainer.json* file and prompt to open the repository inside a container. Select whichever option is most appropriate for your workflow.



After a few moments, the list of files become visible and the local build of the dev container will be completed.



4. Open a new terminal window in Visual Studio Code (`Ctrl` + `Shift` + `` ` ``) and create a new .NET Aspire project using the `dotnet` command-line.

.NET CLI

```
dotnet new aspire-starter -n HelloAspire
```

After a few moments, the project will be created and initial dependencies restored.

5. Open the *ProjectName.AppHost/Program.cs* file in the editor and select the run button on the top right corner of the editor window.

Visual Studio Code builds and starts the .NET Aspire app host and automatically opens the .NET Aspire Dashboard. Because the endpoints hosted in the container

are using a self-signed certificate the first time, you access an endpoint for a specific Dev Container you're presented with a certificate error.

The certificate error is expected. Once you've confirmed that the URL being requested corresponds to the dashboard in the Dev Container you can ignore this warning.

.NET Aspire automatically configures forwarded ports so that when you select on the endpoints in the .NET Aspire dashboard they're tunneled to processes and nested containers within the Dev Container.

6. Commit changes to the GitHub repository

   After successfully creating the .NET Aspire project and verifying that it launches and you can access the dashboard, it's a good idea to commit the changes to the repository.

The preceding walkthrough demonstrates the streamlined process of creating a Dev Container using the .NET Aspire Dev Container template. If you already have an existing repository and wish to utilize Dev Container functionality with .NET Aspire, add a *devcontainer.json* file to the *.devcontainer* folder within your repository:

Directory

```
└──📂 .devcontainer
    └── devcontainer.json
```

The template repository contains a copy of the *devcontainer.json* file that you can use as a starting point, which should be sufficient for .NET Aspire. The following JSON represents the latest version of the *.devcontainer/devcontainer.json* file from the template:

JSON

```
// For format details, see https://aka.ms/devcontainer.json. For config
options, see the
// README at:
https://github.com/devcontainers/templates/tree/main/src/dotnet
{
    "name": ".NET Aspire",
    // Or use a Dockerfile or Docker Compose file. More info:
https://containers.dev/guide/dockerfile
    "image": "mcr.microsoft.com/devcontainers/dotnet:9.0-bookworm",
    "features": {
        "ghcr.io/devcontainers/features/docker-in-docker:2": {},
        "ghcr.io/devcontainers/features/powershell:1": {},
    },

    "hostRequirements": {
        "cpus": 8,
        "memory": "32gb",
        "storage": "64gb"
    },

    // Use 'forwardPorts' to make a list of ports inside the container
available locally.
    // "forwardPorts": [5000, 5001],
    // "portsAttributes": {
    //      "5001": {
    //          "protocol": "https"
    //      }
    // }

    // Use 'postCreateCommand' to run commands after the container is
created.
    // "postCreateCommand": "dotnet restore",
    "onCreateCommand": "dotnet new install Aspire.ProjectTemplates::9.1.0 --
```

```
force",
    "postStartCommand": "dotnet dev-certs https --trust",
    "customizations": {
        "vscode": {
            "extensions": [
                "ms-dotnettools.csdevkit",
                "GitHub.copilot-chat",
                "GitHub.copilot"
            ]
        }
    }
    // Configure tool-specific properties.
    // "customizations": {},

    // Uncomment to connect as root instead. More info: https://aka.ms/dev-
containers-non-root.
    // "remoteUser": "root"
}
```

What's new in .NET Aspire 9.1

Article • 02/25/2025

📢 .NET Aspire 9.1 is the next minor version release of .NET Aspire; it supports *both*:

- .NET 8.0 Long Term Support (LTS) *or*
- .NET 9.0 Standard Term Support (STS).

> **Note**
>
> You're able to use .NET Aspire 9.1 with either .NET 8 or .NET 9!

As always, we focused on highly requested features and pain points from the community. Our theme for 9.1 was "polish, polish, polish"—so you see quality of life fixes throughout the whole platform. Some highlights from this release are resource relationships in the dashboard, support for working in GitHub Codespaces, and publishing resources as a Dockerfile.

If you have feedback, questions, or want to contribute to .NET Aspire, collaborate with us on GitHub or join us on Discord to chat with team members.

Whether you're new to .NET Aspire or have been with us since the preview, it's important to note that .NET Aspire releases out-of-band from .NET releases. While major versions of .NET Aspire align with .NET major versions, minor versions are released more frequently. For more details on .NET and .NET Aspire version support, see:

- .NET support policy: Definitions for LTS and STS.
- .NET Aspire support policy: Important unique product life cycle details.

⬆️ Upgrade to .NET Aspire 9.1

Moving between minor releases of .NET Aspire is simple:

1. In your app host project file (that is, *MyApp.AppHost.csproj*), update the 📦 Aspire.AppHost.Sdk NuGet package to version `9.1.0`:

   XML

   ```xml
   <Project Sdk="Microsoft.NET.Sdk">

       <Sdk Name="Aspire.AppHost.Sdk" Version="9.1.0" />

       <!-- Omitted for brevity -->
   ```

```
</Project>
```

For more information, see .NET Aspire SDK.

2. Check for any NuGet package updates, either using the NuGet Package Manager in Visual Studio or the **Update NuGet Package** command in VS Code.

3. Update to the latest .NET Aspire templates by running the following .NET command line:

```
.NET CLI
```

```
dotnet new update
```

> **Note**
>
> The `dotnet new update` command updates all of your templates to the latest version.

If your app host project file doesn't have the `Aspire.AppHost.Sdk` reference, you might still be using .NET Aspire 8. To upgrade to 9.0, you can follow the documentation from last release.

🌱 Improved onboarding experience

The onboarding experience for .NET Aspire is improved with 9.1. The team worked on creating a GitHub Codespaces template that installs all the necessary dependencies for .NET Aspire, making it easier to get started, including the templates and the ASP.NET Core developer certificate. Additionally, there's support for Dev Containers. For more information, see:

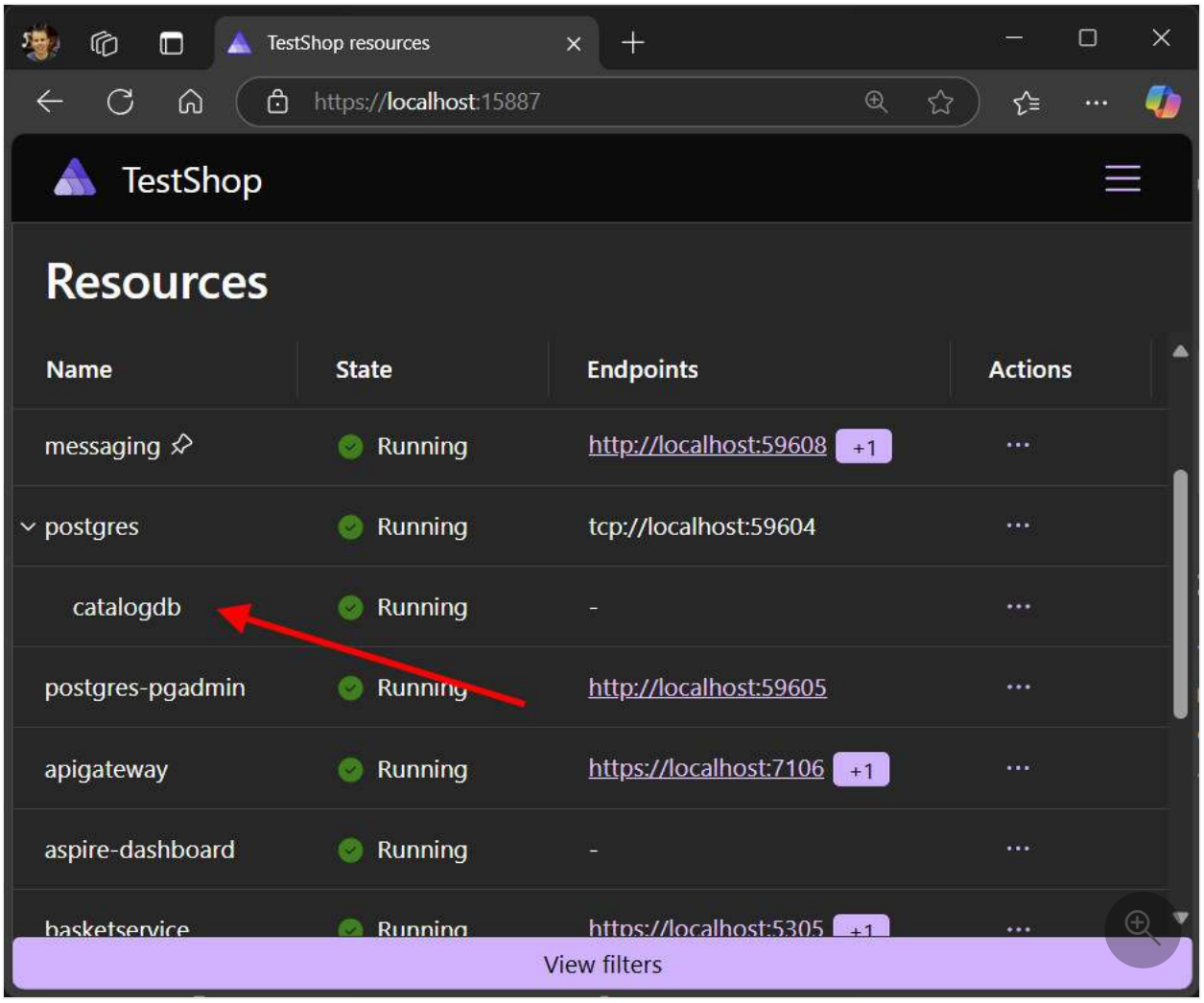
- .NET Aspire and GitHub Codespaces
- .NET Aspire and Visual Studio Code Dev Containers

🔧 Dashboard UX and customization

With every release of .NET Aspire, the dashboard gets more powerful and customizable, this release is no exception. The following features were added to the dashboard in .NET Aspire 9.1:

Resource relationships

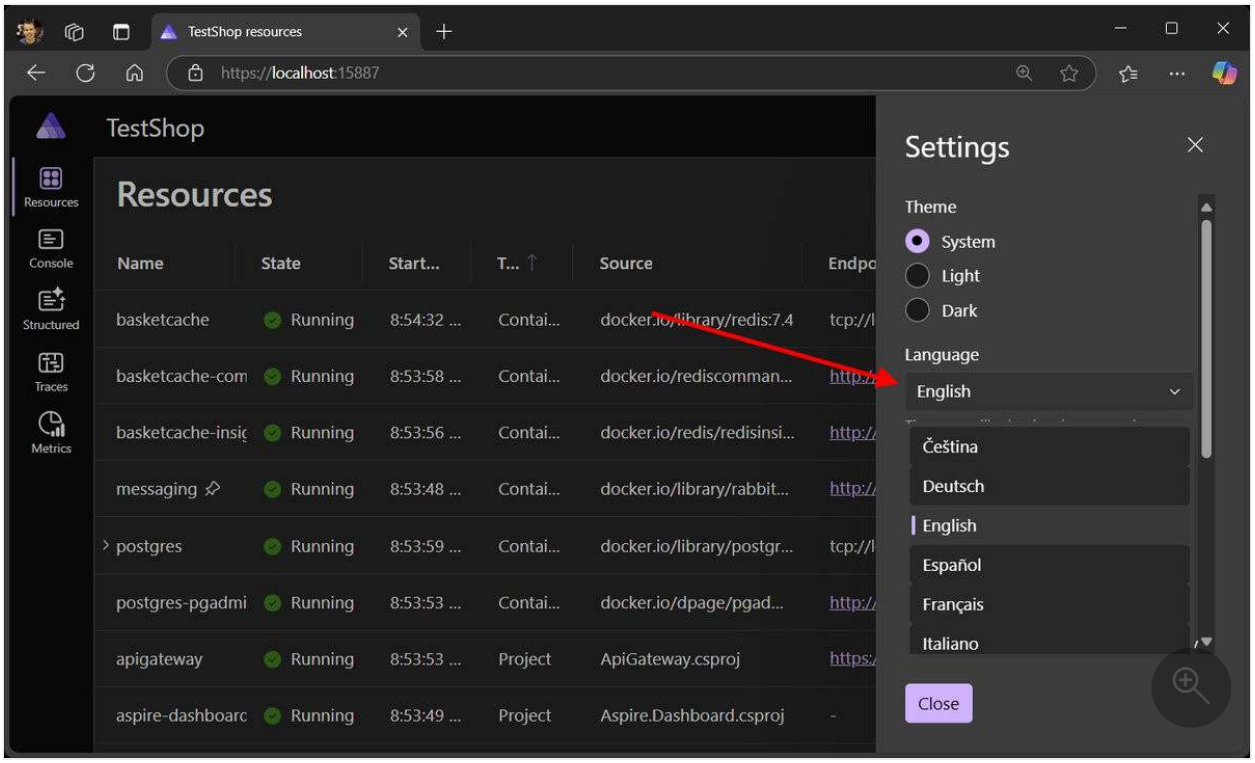
The dashboard now supports "parent" and "child" resource relationships. For instance, when you create a Postgres instance with multiple databases, these databases are nested under the same instance on the **Resource** page.



For more information, see Explore the .NET Aspire dashboard.

Localization overrides

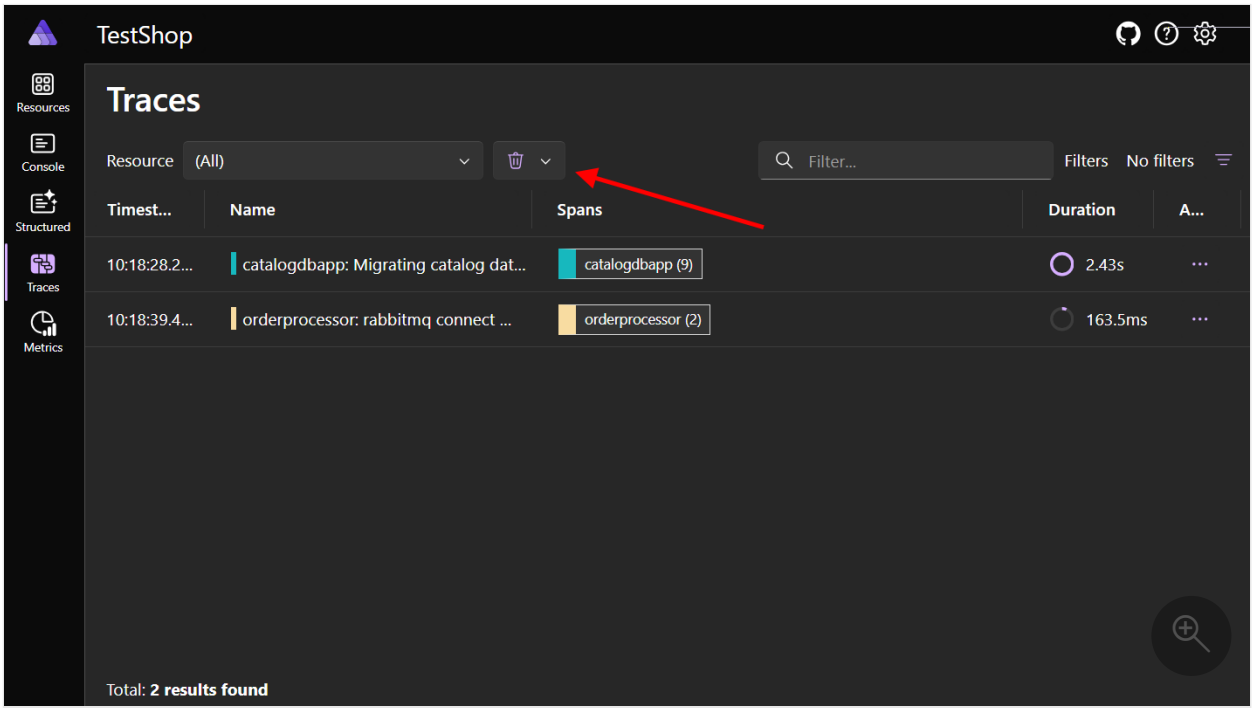
The dashboard defaults to the language set in your browser. This release introduces the ability to override this setting and change the dashboard language independently from the browser language. Consider the following screen capture that demonstrates the addition of the language dropdown in the dashboard:



🗑️ Clear logs and telemetry from the dashboard

New buttons were added to the **Console logs**, **Structured logs**, **Traces** and **Metrics** pages to clear data. There's also a "Remove all" button in the settings popup to remove everything with one action.

Now you use this feature to reset the dashboard to a blank slate, test your app, view only the relevant logs and telemetry, and repeat.



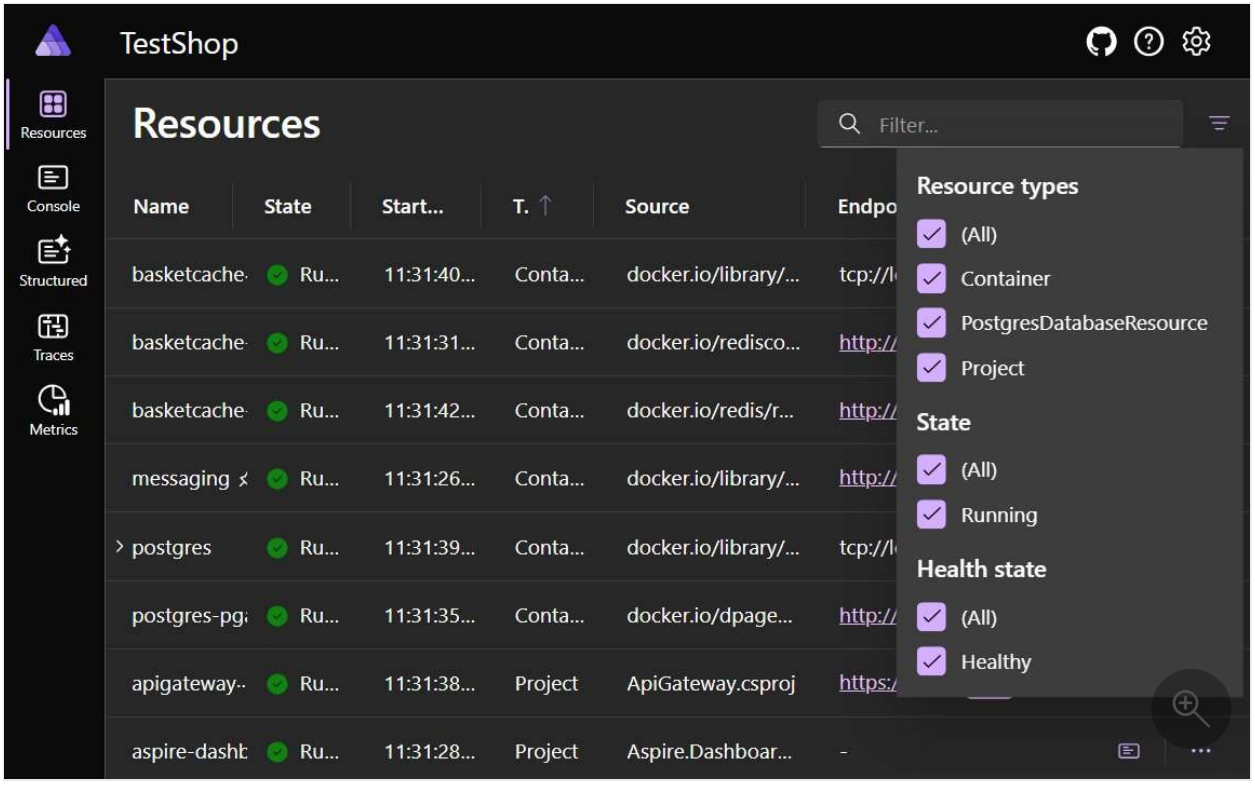
We 💜 love the developer community and thrive on its feedback, collaboration, and contributions. This feature is a community contribution from @Daluur. Join us in

celebrating their contribution by using the feature!

> **Tip**
>
> If you're interested in contributing to .NET Aspire, look for issues labeled with **good first issue** and follow the **contributor guide**.

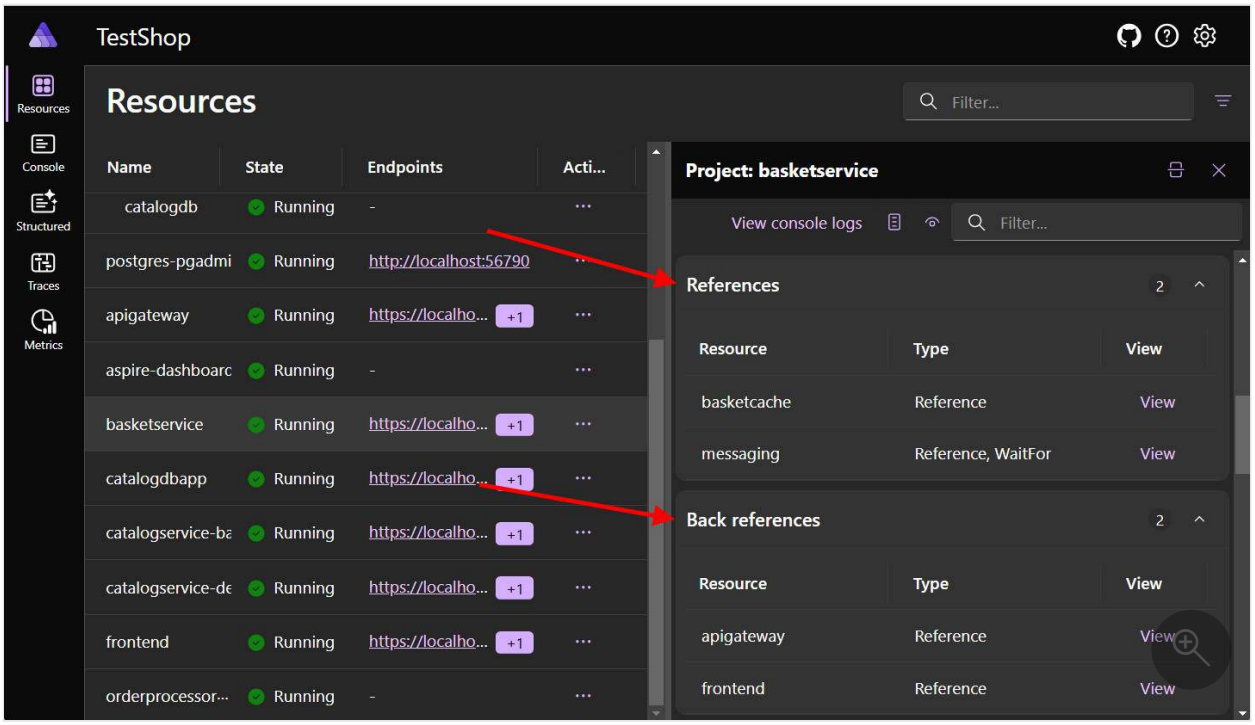
New filtering

You can now filter what you see in the **Resource** page by **Resource type**, **State**, and **Health state**. Consider the following screen capture, which demonstrates the addition of the filter options in the dashboard:



More resource details

When you select a resource in the dashboard, the details pane now displays new data points, including **References**, **Back references**, and **Volumes** with their mount types. This enhancement provides a clearer and more comprehensive view of your resources, improving the overall user experience by making relevant details more accessible.



For more information, see .NET Aspire dashboard: Resources page.

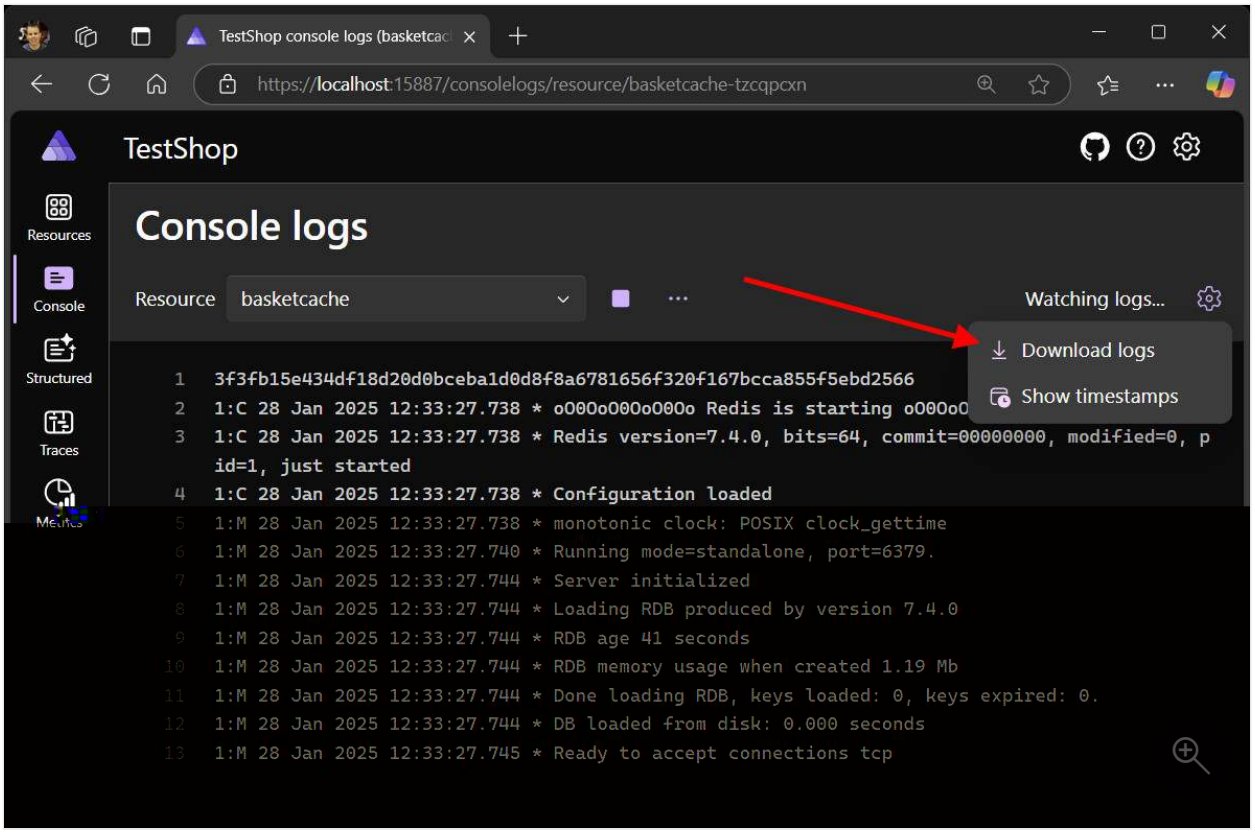
CORS support for custom local domains

You can now set the `DOTNET_DASHBOARD_CORS_ALLOWED_ORIGINS` environment variable to allow the dashboard to receive telemetry from other browser apps, such as if you have resources running on custom localhost domains.

For more information, see .NET Aspire app host: Dashboard configuration.

Flexibility with console logs

The console log page has two new options. You're now able to download your logs so you can view them in your own diagnostics tools. Plus, you can turn timestamps on or off to reduce visual clutter when needed.



For more information, see .NET Aspire dashboard: Console logs page.

🎨 Various UX improvements

Several new features in .NET Aspire 9.1 enhance and streamline the following popular tasks:

- ▶️ Resource commands, such as **Start** and **Stop** buttons, are now available on the **Console logs** page.
- 🔍 Single selection to open in the *text visualizer*.
- 🔗 URLs within logs are now automatically clickable, with commas removed from endpoints.

Additionally, the 🖱️ scroll position resets when switching between different resources—this helps to visually reset the current resource view.

For more details on the latest dashboard enhancements, check out James Newton-King on Bluesky, where he's been sharing new features daily.

⚙️ Local development enhancements

In .NET Aspire 9.1, several improvements to streamline your local development experience were an emphasis. These enhancements are designed to provide greater

flexibility, better integration with Docker, and more efficient resource management. Here are some of the key updates:

▶️ Start resources on demand

You can now tell resources not to start with the rest of your app by using WithExplicitStart on the resource in your app host. Then, you can start it whenever you're ready from inside the dashboard.

For more information, see Configure explicit resource start.

Better Docker integration

The `PublishAsDockerfile()` feature was introduced for all projects and executable resources. This enhancement allows for complete customization of the Docker container and Dockerfile used during the publish process.

While this API was available in previous versions, it couldn't be used with ProjectResource or ExecutableResource types.

🧹 Cleaning up Docker networks

In 9.1, we addressed a persistent issue where Docker networks created by .NET Aspire would remain active even after the application was stopped. This bug, tracked in .NET Aspire GitHub issue #6504, is resolved. Now, Docker networks are properly cleaned up, ensuring a more efficient and tidy development environment.

✅ Socket address issues fixed

Several users reported issues (#6693, #6704, #7095) with restarting the .NET Aspire app host, including reconciliation errors and "address already in use" messages.

This release introduces a more robust approach to managing socket addresses, ensuring only one instance of each address is used at a time. Additionally, improvements were made to ensure proper project restarts and resource releases, preventing hanging issues. These changes enhance the stability and reliability of the app host, especially during development and testing.

🔌 Integration updates

.NET Aspire continues to excel through its integrations with various platforms. This release includes numerous updates to existing integrations and details about ownership migrations, enhancing the overall functionality and user experience.

Azure updates

This release also focused on improving various Azure integrations:

New emulators

We're excited to bring new emulators for making local development easier. The following integrations got new emulators in this release:

- Azure Service Bus
- Azure Cosmos DB Linux-based (preview)
- Azure SignalR

```C#
var serviceBus = builder.AddAzureServiceBus("servicebus")
                        .RunAsEmulator();

#pragma warning disable ASPIRECOSMOSDB001
var cosmosDb = builder.AddAzureCosmosDB("cosmosdb")
                      .RunAsPreviewEmulator();

var signalr = builder.AddAzureSignalR("signalr",
AzureSignalRServiceMode.Serverless)
                     .RunAsEmulator();
```

These new emulators work side-by-side with the existing emulators for:

- Azure Storage
- Azure Event Hubs
- Azure Cosmos DB

Cosmos DB

Along with support for the new emulator, Cosmos DB added the following features.

Support for Entra ID authentication by default

Previously, the Cosmos DB integration used access keys and a Key Vault secret to connect to the service. .NET Aspire 9.1 added support for using more secure

authentication using managed identities by default. If you need to keep using access key authentication, you can get back to the previous behavior by calling WithAccessKeyAuthentication.

Support for modeling Database and Containers in the app host

You can define a Cosmos DB database and containers in the app host and these resources are available when you run the application in both the emulator and in Azure. This allows you to define these resources up front and no longer need to create them from the application, which might not have permission to create them.

For example API usage to add database and containers, see the following related articles:

- .NET Aspire Azure Cosmos DB integration
- .NET Aspire Cosmos DB Entity Framework Core integration

Support for Cosmos DB-based triggers in Azure Functions

The AzureCosmosDBResource was modified to support consumption in Azure Functions applications that uses the Cosmos DB trigger. A Cosmos DB resource can be initialized and added as a reference to an Azure Functions resource with the following code:

C#

```
var cosmosDb = builder.AddAzureCosmosDB("cosmosdb")
                      .RunAsEmulator();
var database = cosmosDb.AddCosmosDatabase("mydatabase");
database.AddContainer("mycontainer", "/id");

var funcApp =
builder.AddAzureFunctionsProject<Projects.AzureFunctionsEndToEnd_Functions>
("funcapp")
    .WithReference(cosmosDb)
    .WaitFor(cosmosDb);
```

The resource can be used in the Azure Functions trigger as follows:

C#

```
public class MyCosmosDbTrigger(ILogger<MyCosmosDbTrigger> logger)
{
    [Function(nameof(MyCosmosDbTrigger))]
    public void Run([CosmosDBTrigger(
        databaseName: "mydatabase",
        containerName: "mycontainer",
```

```
        CreateLeaseContainerIfNotExists = true,
        Connection = "cosmosdb")] IReadOnlyList<Document> input)
    {
        logger.LogInformation(
            "C# cosmosdb trigger function processed: {Count} messages",
            input.Count);
    }
}
```

For more information using Azure Functions with .NET Aspire, see .NET Aspire Azure Functions integration (Preview).

🚚 Service Bus and Event Hubs

Similar to Cosmos DB, the Service Bus and Event Hubs integrations now allow you to define Azure Service Bus queues, topics, subscriptions, and Azure Event Hubs instances and consumer groups directly in your app host code. This enhancement simplifies your application logic by enabling the creation and management of these resources outside the application itself.

For more information, see the following updated articles:

- .NET Aspire Azure Service Bus integration
- .NET Aspire Azure Event Hubs integration

♻️ Working with existing resources

There's consistent feedback about making it easier to connect to existing Azure resources in .NET Aspire. With 9.1, you can now easily connect to an existing Azure resource either directly by `string` name, or with app model parameters which can be changed at deployment time. For example to connect to an Azure Service Bus account, we can use the following code:

C#

```
var existingServiceBusName = builder.AddParameter("serviceBusName");
var existingServiceBusResourceGroup =
builder.AddParameter("serviceBusResourceGroup");

var serviceBus = builder.AddAzureServiceBus("messaging")
                        .AsExisting(existingServiceBusName,
existingServiceBusResourceGroup);
```

The preceding code reads the name and resource group from the parameters, and connects to the existing resource when the application is run or deployed. For more

information, see use existing Azure resources.

🌎 Azure Container Apps

Experimental support for configuring custom domains in Azure Container Apps (ACA) was added. For example:

C#

```
#pragma warning disable ASPIREACADOMAINS001

var customDomain = builder.AddParameter("customDomain");
var certificateName = builder.AddParameter("certificateName");

builder.AddProject<Projects.AzureContainerApps_ApiService>("api")
       .WithExternalHttpEndpoints()
       .PublishAsAzureContainerApp((infra, app) =>
       {
           app.ConfigureCustomDomain(customDomain, certificateName);
       });
```

For more information, see .NET Aspire diagnostics overview.

➕ Even more integration updates

- OpenAI now supports the Microsoft.Extensions.AI NuGet package.
- RabbitMQ updated to version 7, and MongoDB to version 3. These updates introduced breaking changes, leading to the release of new packages with version-specific suffixes. The original packages continue to use the previous versions, while the new packages are as follows:
  - Aspire.RabbitMQ.Client.v7 NuGet package. For more information, see the .NET Aspire RabbitMQ client integration documentation.
  - Aspire.MongoDB.Driver.v3 NuGet package. For more information, see the .NET Aspire MongoDB client integration documentation.
- Dapr migrated to the CommunityToolkit to facilitate faster innovation.
- Numerous other integrations received updates, fixes, and new features. For detailed information, refer to our GitHub release notes.

The Aspire.Hosting.AWS NuGet package and source code migrated under Amazon Web Services (AWS)) ownership. This migration happened as part of .NET Aspire 9.0, we're just restating that change here.

🧪 Testing in .NET Aspire

.NET Aspire 9.1 simplifies writing cross-functional integration tests with a robust approach. The app host allows you to create, evaluate, and manage containerized environments seamlessly within a test run. This functionality supports popular testing frameworks like xUnit, NUnit, and MSTest, enhancing your testing capabilities and efficiency.

Now, you're able to disable port randomization or enable the dashboard. For more information, see .NET Aspire testing overview. Additionally, you can now Pass arguments to your app host.

Some of these enhancements were introduced as a result of stability issues that were reported, such as .NET Aspire GitHub issue #6678 —where some resources failed to start do to "address in use" errors.

🚀 Deployment

Significant improvements to the Azure Container Apps (ACA) deployment process are included in .NET Aspire 9.1, enhancing both the `azd` CLI and app host options. One of the most requested features—support for deploying `npm` applications to ACA—is now implemented. This new capability allows `npm` apps to be deployed to ACA just like other resources, streamlining the deployment process and providing greater flexibility for developers.

We recognize there's more work to be done in the area of deployment. Future releases will continue to address these opportunities for improvement. For more information on deploying .NET Aspire to ACA, see Deploy a .NET Aspire project to Azure Container Apps.

⚠️ Breaking changes

.NET Aspire is moving quickly, and with that comes breaking changes. Breaking are categorized as either:

- **Binary incompatible**: The assembly version has changed, and you need to recompile your code.
- **Source incompatible**: The source code has changed, and you need to change your code.
- **Behavioral change**: The code behaves differently, and you need to change your code.

Typically APIs are decorated with the ObsoleteAttribute giving you a warning when you compile, and an opportunity to adjust your code. For an overview of breaking changes in .NET Aspire 9.1, see Breaking changes in .NET Aspire 9.1.

🎯 Upgrade today

Follow the directions outlined in the Upgrade to .NET Aspire 9.1 section to make the switch to 9.1 and take advantage of all these new features today! As always, we're listening for your feedback on GitHub -and looking out for what you want to see in 9.2 ☺.

For a complete list of issues addressed in this release, see .NET Aspire GitHub repository —9.1 milestone.

Upgrade to .NET Aspire 9.0

Article • 11/12/2024

.NET Aspire 9.0 is now generally available. In this article, you learn the steps involved in updating your existing .NET Aspire 8.x projects to .NET Aspire 9.0. There are a few ways in which you can update your projects to .NET Aspire 9.0:

- Manually upgrade your projects to .NET Aspire 9.0.
- Use the **Upgrade Assistant** to upgrade your projects to .NET Aspire 9.0.

> **Tip**
>
> If you're new to .NET Aspire, there's no reason to upgrade anything. For more information, see **.NET Aspire setup and tooling**.

Prerequisites

Before you upgrade your projects to .NET Aspire 9.0, ensure that you have the following prerequisites:

- Install the latest tooling.
- Use the .NET Aspire SDK.

> **Note**
>
> Feel free to uninstall the .NET Aspire workload as you'll no longer need it.
>
> .NET CLI
>
> ```
> dotnet workload uninstall aspire
> ```
>
> For more information, see **dotnet workload uninstall**.

If you don't uninstall the .NET Aspire workload, and you're using the new .NET Aspire SDK and templates, you see both .NET Aspire 8.0 and .NET Aspire 9.0 templates.

Manually upgrade to .NET Aspire 9.0

To upgrade your projects to .NET Aspire 9.0, you need to update your project files. The following steps guide you through the process:

- Edit your app host project file to use the new .NET Aspire 9.0 SDK (`Aspire.AppHost.Sdk`).
- Update the NuGet packages in your project files to the latest versions.
- Adjust your *Program.cs* file to use the new APIs and remove any obsolete APIs.

Edit your app host project file

To upgrade your app host project to .NET Aspire 9.0, you need to update your project file to use the new Aspire.AppHost.Sdk:

diff

```
<Project Sdk="Microsoft.NET.Sdk">

+  <Sdk Name="Aspire.AppHost.Sdk" Version="9.0.0" />

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <IsAspireHost>true</IsAspireHost>
    <UserSecretsId>0afc20a6-cd99-4bf7-aae1-1359b0d45189</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Aspire.Hosting.AppHost" Version="8.0.0" />
  </ItemGroup>

</Project>
```

Optionally upgrade the target framework moniker (TFM)

.NET Aspire 9.0 runs on .NET 9.0, but you can also run it on .NET 8.0. In other words, just because you're using the .NET Aspire SDK, and pointing to version 9.0 packages, you can still target .NET 8.0. If you want to run your .NET Aspire 9.0 project on .NET 9.0, you need to update the `TargetFramework` property in your project file:

diff

```
<Project Sdk="Microsoft.NET.Sdk">

  <Sdk Name="Aspire.AppHost.Sdk" Version="9.0.0" />
```

```
  <PropertyGroup>
    <OutputType>Exe</OutputType>
-   <TargetFramework>net8.0</TargetFramework>
+   <TargetFramework>net9.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <IsAspireHost>true</IsAspireHost>
    <UserSecretsId>0afc20a6-cd99-4bf7-aae1-1359b0d45189</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Aspire.Hosting.AppHost" Version="9.1.0" />
  </ItemGroup>

</Project>
```

For more information on TFMs, see Target frameworks in SDK-style projects: Latest versions.

Overall app host project differences

If you followed all of the preceding steps, your app host project file should look like this:

diff

```
<Project Sdk="Microsoft.NET.Sdk">

+  <Sdk Name="Aspire.AppHost.Sdk" Version="9.0.0" />

  <PropertyGroup>
    <OutputType>Exe</OutputType>
-   <TargetFramework>net8.0</TargetFramework>
+   <TargetFramework>net9.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <IsAspireHost>true</IsAspireHost>
    <UserSecretsId>0afc20a6-cd99-4bf7-aae1-1359b0d45189</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
-   <PackageReference Include="Aspire.Hosting.AppHost" Version="8.0.0" />
+   <PackageReference Include="Aspire.Hosting.AppHost" Version="9.1.0" />
  </ItemGroup>

</Project>
```

The changes include the addition of the `Aspire.AppHost.Sdk`, the update of the `TargetFramework` property to `net9.0`, and the update of the `Aspire.Hosting.AppHost`

package to version `9.0.0`.

Adjust your *Program.cs* file

With the introduction of .NET Aspire 9.0, there are some *breaking changes*. Some APIs were originally marked as experimental (with the ExperimentalAttribute) and are now removed, while other APIs are now attributed as ObsoleteAttribute with details on new replacement APIs. You need to adjust your *Program.cs* file (and potentially other affected APIs) to use the new APIs. If you're using the Upgrade Assistant to upgrade your projects, it automatically adjusts your *Program.cs* file in most cases.

For the complete list of breaking changes in .NET Aspire 9.0, see Breaking changes in .NET Aspire 9.0.

Use the Upgrade Assistant

The Upgrade Assistant is a tool that helps upgrade targeted projects to the latest version. If you're new to the Upgrade Assistant, there's two modalities to choose from:

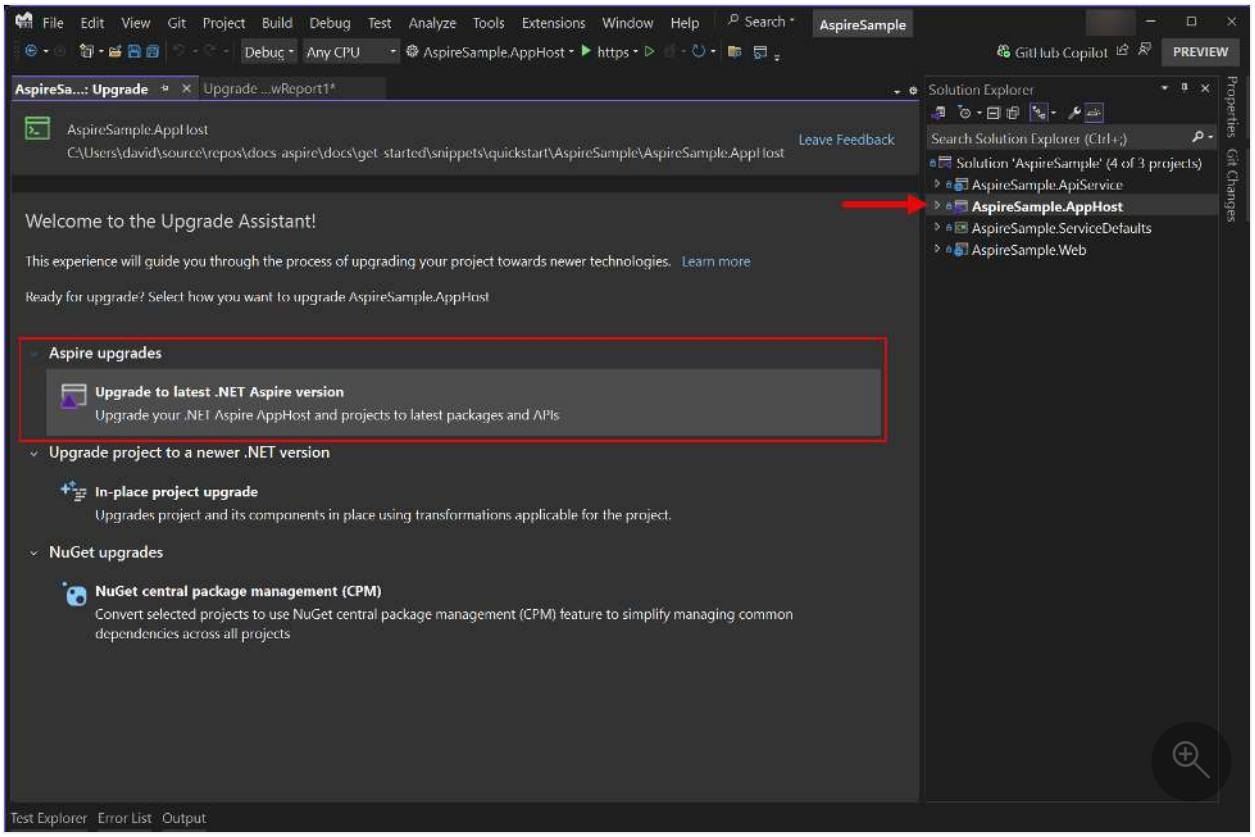
- The Visual Studio extension version.
- The .NET CLI global tool version.

Regardless of how you install the Upgrade Assistant, you can use it to upgrade your .NET Aspire 8.x projects to .NET Aspire 9.0.

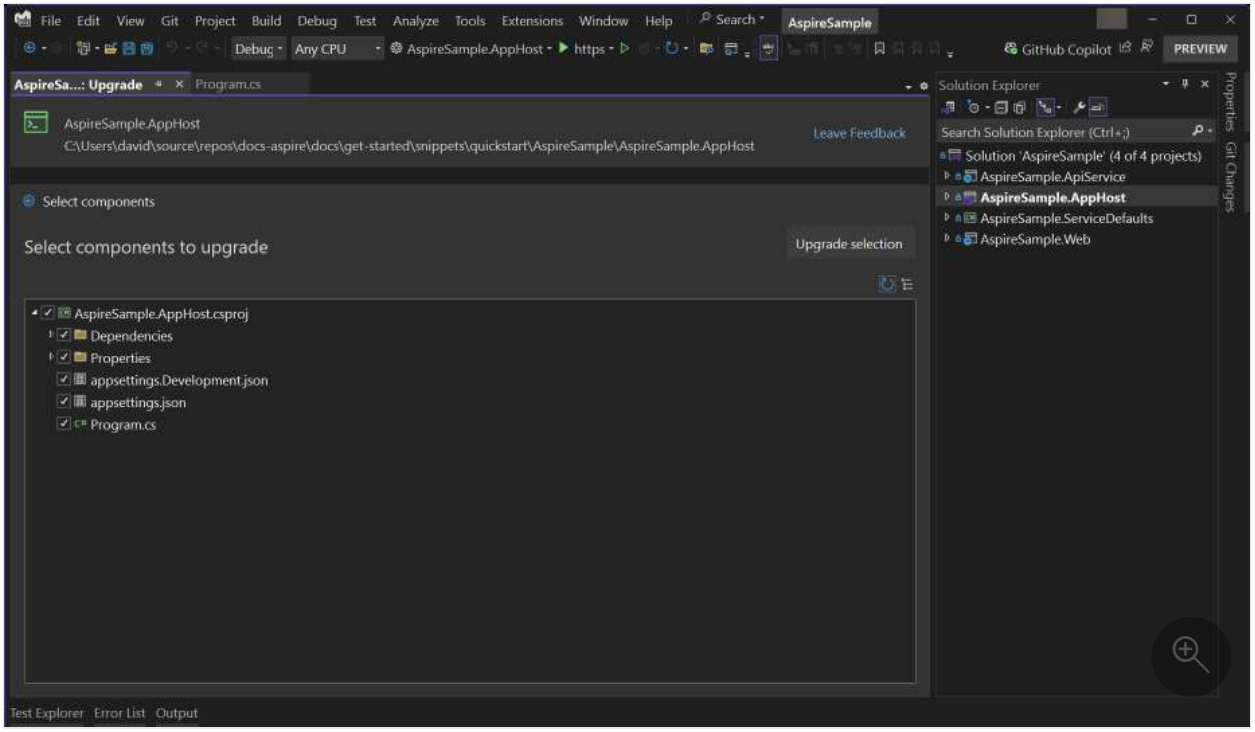
To upgrade the .NET Aspire app host project to .NET Aspire 9.0 with Visual Studio, right-click the project in **Solution Explorer** and select **Upgrade**.

> **ⓘ Important**
>
> If the **Upgrade Assistant** isn't already installed, you'll be prompted to install it.

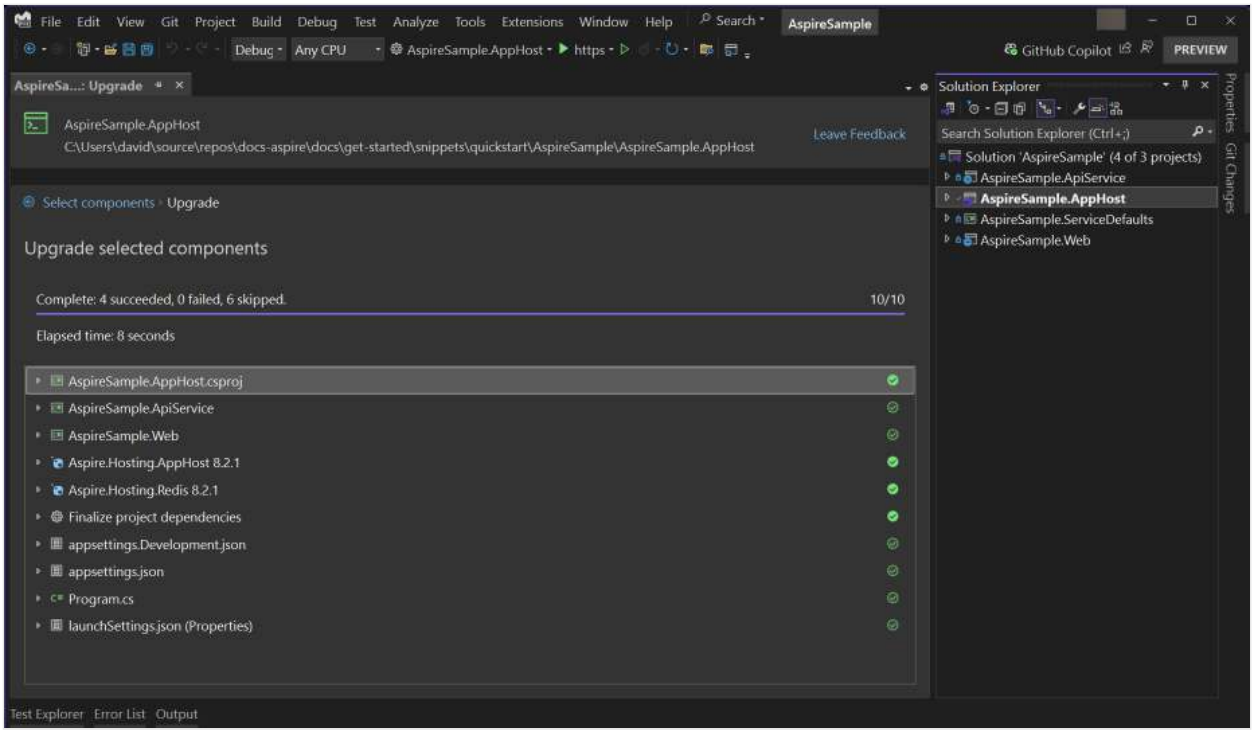
The Upgrade Assistant displays a welcome package. Select the **Aspire upgrades** option:



With the **Aspire upgrades** option selected, the Upgrade Assistant displays the selectable upgrade target components. Leave all the options checked and select **Upgrade selection**:



Finally, after selecting the components to upgrade, the Upgrade Assistant displays the results of the upgrade process. If everything was successful, you see green check marks next to each component:



To take advantage of the latest updates in your .NET Aspire solution, update all NuGet packages to version `9.0.0`.

With the app host project updated, your project file should look like this:

```
diff
```

```
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <IsAspireHost>true</IsAspireHost>
    <UserSecretsId>0afc20a6-cd99-4bf7-aae1-1359b0d45189</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
-   <PackageReference Include="Aspire.Hosting.AppHost" Version="8.0.0" />
+   <PackageReference Include="Aspire.Hosting.AppHost" Version="9.1.0" />
  </ItemGroup>

</Project>
```

> **Tip**
>
> You'll want to also update the NuGet packages in your other projects to the latest versions.

Verify the upgrade

As with any upgrade, ensure that the app runs as expected and that all tests pass. Build the solution and look for suggestions, warnings, or errors in the output window—address anything that wasn't an issue before. If you encounter any issues, let us know by filing a GitHub issue.

.NET Aspire orchestration overview

Article • 03/14/2025

.NET Aspire provides APIs for expressing resources and dependencies within your distributed application. In addition to these APIs, there's tooling that enables several compelling scenarios. The orchestrator is intended for *local development* purposes and isn't supported in production environments.

Before continuing, consider some common terminology used in .NET Aspire:

- **App model**: A collection of resources that make up your distributed application (DistributedApplication), defined within the Aspire.Hosting.ApplicationModel namespace. For a more formal definition, see Define the app model.
- **App host/Orchestrator project**: The .NET project that orchestrates the *app model*, named with the **.AppHost* suffix (by convention).
- **Resource**: A resource is a dependent part of an application, such as a .NET project, container, executable, database, cache, or cloud service. It represents any part of the application that can be managed or referenced.
- **Integration**: An integration is a NuGet package for either the *app host* that models a *resource* or a package that configures a client for use in a consuming app. For more information, see .NET Aspire integrations overview.
- **Reference**: A reference defines a connection between resources, expressed as a dependency using the WithReference API. For more information, see Reference resources or Reference existing resources.

> **Note**
>
> .NET Aspire's orchestration is designed to enhance your *local development* experience by simplifying the management of your cloud-native app's configuration and interconnections. While it's an invaluable tool for development, it's not intended to replace production environment systems like Kubernetes, which are specifically designed to excel in that context.

Define the app model

.NET Aspire empowers you to seamlessly build, provision, deploy, configure, test, run, and observe your distributed applications. All of these capabilities are achieved through the utilization of an *app model* that outlines the resources in your .NET Aspire solution and their relationships. These resources encompass projects, executables, containers,

and external services and cloud resources that your app depends on. Within every .NET Aspire solution, there's a designated App host project, where the app model is precisely defined using methods available on the IDistributedApplicationBuilder. This builder is obtained by invoking DistributedApplication.CreateBuilder.

C#

```
// Create a new app model builder
var builder = DistributedApplication.CreateBuilder(args);

// TODO:
//   Add resources to the app model
//   Express dependencies between resources

builder.Build().Run();
```

App host project

The app host project handles running all of the projects that are part of the .NET Aspire project. In other words, it's responsible for orchestrating all apps within the app model. The project itself is a .NET executable project that references the  Aspire.Hosting.AppHost NuGet package, sets the `IsAspireHost` property to `true`, and references the .NET Aspire SDK:

XML

```
<Project Sdk="Microsoft.NET.Sdk">

    <Sdk Name="Aspire.AppHost.Sdk" Version="9.1.0" />

    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net9.0</TargetFramework>
        <IsAspireHost>true</IsAspireHost>
        <!-- Omitted for brevity -->
    </PropertyGroup>

    <ItemGroup>
        <PackageReference Include="Aspire.Hosting.AppHost" Version="9.1.0"
/>
    </ItemGroup>

    <!-- Omitted for brevity -->

</Project>
```

The following code describes an app host `Program` with two project references and a Redis cache:

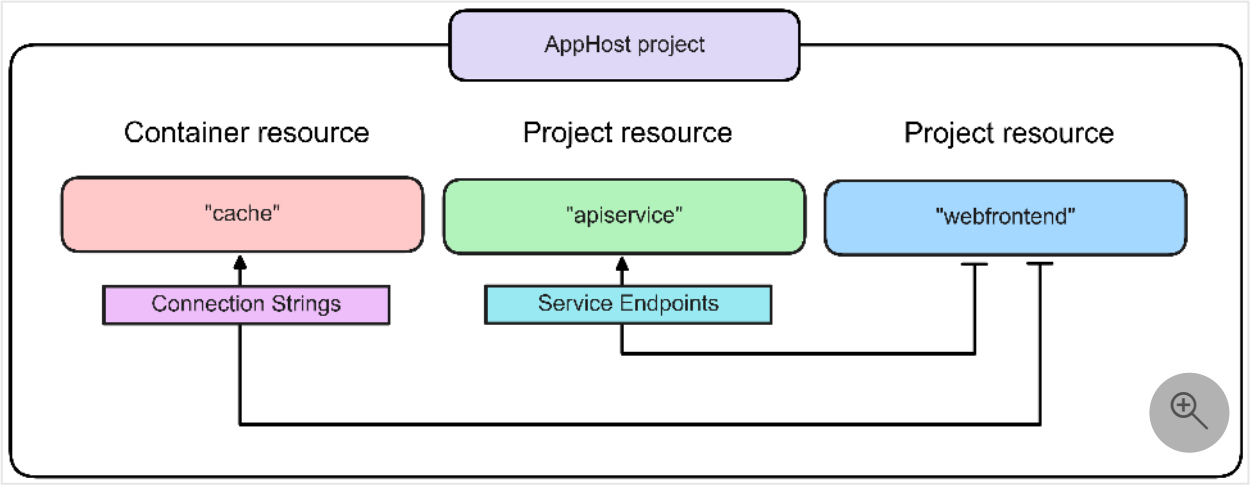
```C#
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache");

var apiservice = builder.AddProject<Projects.AspireApp_ApiService>
("apiservice");

builder.AddProject<Projects.AspireApp_Web>("webfrontend")
       .WithExternalHttpEndpoints()
       .WithReference(cache)
       .WaitFor(cache)
       .WithReference(apiService)
       .WaitFor(apiService);

builder.Build().Run();
```

The preceding code:

- Creates a new app model builder using the CreateBuilder method.
- Adds a Redis `cache` resource named "cache" using the AddRedis method.
- Adds a project resource named "apiservice" using the AddProject method.
- Adds a project resource named "webfrontend" using the AddProject method.
  - Specifies that the project has external HTTP endpoints using the WithExternalHttpEndpoints method.
  - Adds a reference to the `cache` resource and waits for it to be ready using the WithReference and WaitFor methods.
  - Adds a reference to the `apiservice` resource and waits for it to be ready using the WithReference and WaitFor methods.
- Builds and runs the app model using the Build and Run methods.

The example code uses the .NET Aspire Redis hosting integration.

To help visualize the relationship between the app host project and the resources it describes, consider the following diagram:



Each resource must be uniquely named. This diagram shows each resource and the relationships between them. The container resource is named "cache" and the project resources are named "apiservice" and "webfrontend". The web frontend project references the cache and API service projects. When you're expressing references in this way, the web frontend project is saying that it depends on these two resources, the "cache" and "apiservice" respectively.

Built-in resource types

.NET Aspire projects are made up of a set of resources. The primary base resource types in the Aspire.Hosting.AppHost NuGet package are described in the following table:

Expand table

Method	Resource type	Description
AddProject	ProjectResource	A .NET project, for example, an ASP.NET Core web app.
AddContainer	ContainerResource	A container image, such as a Docker image.
AddExecutable	ExecutableResource	An executable file, such as a Node.js app.
AddParameter	ParameterResource	A parameter resource that can be used to express external parameters.

Project resources represent .NET projects that are part of the app model. When you add a project reference to the app host project, the .NET Aspire SDK generates a type in the `Projects` namespace for each referenced project. For more information, see .NET Aspire SDK: Project references.

To add a project to the app model, use the AddProject method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

// Adds the project "apiservice" of type "Projects.AspireApp_ApiService".
var apiservice = builder.AddProject<Projects.AspireApp_ApiService>
("apiservice");
```

Projects can be replicated and scaled out by adding multiple instances of the same project to the app model. To configure replicas, use the WithReplicas method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

// Adds the project "apiservice" of type "Projects.AspireApp_ApiService".
var apiservice = builder.AddProject<Projects.AspireApp_ApiService>
("apiservice")
                        .WithReplicas(3);
```

The preceding code adds three replicas of the "apiservice" project resource to the app model. For more information, see .NET Aspire dashboard: Resource replicas.

Configure explicit resource start

Project, executable and container resources are automatically started with your distributed application by default. A resource can be configured to wait for an explicit startup instruction with the WithExplicitStart method. A resource configured with WithExplicitStart is initialized with KnownResourceStates.NotStarted.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres");
var postgresdb = postgres.AddDatabase("postgresdb");

builder.AddProject<Projects.AspireApp_DbMigration>("dbmigration")
       .WithReference(postgresdb)
       .WithExplicitStart();
```

In the preceeding code the "dbmigration" resource is configured to not automatically start with the distributed application.

Resources with explicit start can be started from the .NET Aspire dashboard by clicking the "Start" command. For more information, see .NET Aspire dashboard: Stop or Start a

resource.

Reference resources

A reference represents a dependency between resources. For example, you can probably imagine a scenario where you a web frontend depends on a Redis cache. Consider the following example app host `Program` C# code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache");

builder.AddProject<Projects.AspireApp_Web>("webfrontend")
       .WithReference(cache);
```

The "webfrontend" project resource uses WithReference to add a dependency on the "cache" container resource. These dependencies can represent connection strings or service discovery information. In the preceding example, an environment variable is *injected* into the "webfrontend" resource with the name `ConnectionStrings__cache`. This environment variable contains a connection string that the `webfrontend` uses to connect to Redis via the .NET Aspire Redis integration, for example, `ConnectionStrings__cache="localhost:62354"`.

Waiting for resources

In some cases, you might want to wait for a resource to be ready before starting another resource. For example, you might want to wait for a database to be ready before starting an API that depends on it. To express this dependency, use the WaitFor method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres");
var postgresdb = postgres.AddDatabase("postgresdb");

builder.AddProject<Projects.AspireApp_ApiService>("apiservice")
       .WithReference(postgresdb)
       .WaitFor(postgresdb);
```

In the preceding code, the "apiservice" project resource waits for the "postgresdb" database resource to enter the KnownResourceStates.Running. The example code shows

the .NET Aspire PostgreSQL integration, but the same pattern can be applied to other resources.

Other cases might warrant waiting for a resource to run to completion, either KnownResourceStates.Exited or KnownResourceStates.Finished before the dependent resource starts. To wait for a resource to run to completion, use the WaitForCompletion method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres");
var postgresdb = postgres.AddDatabase("postgresdb");

var migration = builder.AddProject<Projects.AspireApp_Migration>
("migration")
                       .WithReference(postgresdb)
                       .WaitFor(postgresdb);

builder.AddProject<Projects.AspireApp_ApiService>("apiservice")
       .WithReference(postgresdb)
       .WaitForCompletion(migration);
```

In the preceding code, the "apiservice" project resource waits for the "migration" project resource to run to completion before starting. The "migration" project resource waits for the "postgresdb" database resource to enter the KnownResourceStates.Running. This can be useful in scenarios where you want to run a database migration before starting the API service, for example.

Forcing resource start in the dashboard

Waiting for a resource can be bypassed using the "Start" command in the dashboard. Clicking "Start" on a waiting resource in the dashboard instructs it to start immediately without waiting for the resource to be healthy or completed. This can be useful when you want to test a resource immediately and don't want to wait for the app to be in the right state.

APIs for adding and expressing resources

.NET Aspire hosting integrations and client integrations are both delivered as NuGet packages, but they serve different purposes. While *client integrations* provide client library configuration for consuming apps outside the scope of the app host, *hosting integrations* provide APIs for expressing resources and dependencies within the app

host. For more information, see .NET Aspire integrations overview: Integration responsibilities.

Express container resources

To express a ContainerResource you add it to an IDistributedApplicationBuilder instance by calling the AddContainer method:

Docker

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var ollama = builder.AddContainer("ollama", "ollama/ollama")
    .WithBindMount("ollama", "/root/.ollama")
    .WithBindMount("./ollamaconfig", "/usr/config")
    .WithHttpEndpoint(port: 11434, targetPort: 11434, name: "ollama")
    .WithEntrypoint("/usr/config/entrypoint.sh")
    .WithContainerRuntimeArgs("--gpus=all");
```

For more information, see GPU support in Docker Desktop.

The preceding code adds a container resource named "ollama" with the image `ollama/ollama`. The container resource is configured with multiple bind mounts, a named HTTP endpoint, an entrypoint that resolves to Unix shell script, and container run arguments with the WithContainerRuntimeArgs method.

Customize container resources

All ContainerResource subclasses can be customized to meet your specific requirements. This can be useful when using a hosting integration that models a container resource, but requires modifications. When you have an `IResourceBuilder<ContainerResource>` you can chain calls to any of the available APIs to modify the container resource. .NET Aspire container resources typically point to pinned tags, but you might want to use the `latest` tag instead.

To help exemplify this, imagine a scenario where you're using the .NET Aspire Redis integration. If the Redis integration relies on the `7.4` tag and you want to use the `latest` tag instead, you can chain a call to the WithImageTag API:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
                   .WithImageTag("latest");

// Instead of using the "7.4" tag, the "cache"
// container resource now uses the "latest" tag.
```

For more information and additional APIs available, see ContainerResourceBuilderExtensions.

Container resource lifecycle

When the app host is run, the ContainerResource is used to determine what container image to create and start. Under the hood, .NET Aspire runs the container using the defined container image by delegating calls to the appropriate OCI-compliant container runtime, either Docker or Podman. The following commands are used:

Docker

First, the container is created using the `docker container create` command. Then, the container is started using the `docker container start` command.

- docker container create: Creates a new container from the specified image, without starting it.
- docker container start: Start one or more stopped containers.

These commands are used instead of `docker run` to manage attached container networks, volumes, and ports. Calling these commands in this order allows any IP (network configuration) to already be present at initial startup.

Beyond the base resource types, ProjectResource, ContainerResource, and ExecutableResource, .NET Aspire provides extension methods to add common resources to your app model. For more information, see Hosting integrations.

Container resource lifetime

By default, container resources use the *session* container lifetime. This means that every time the app host process is started, the container is created and started. When the app host stops, the container is stopped and removed. Container resources can opt-in to a *persistent* lifetime to avoid unnecessary restarts and use persisted container state. To

achieve this, chain a call the ContainerResourceBuilderExtensions.WithLifetime API and pass ContainerLifetime.Persistent:

```C#
var builder = DistributedApplication.CreateBuilder(args);

var ollama = builder.AddContainer("ollama", "ollama/ollama")
    .WithLifetime(ContainerLifetime.Persistent);
```

The preceding code adds a container resource named "ollama" with the image "ollama/ollama" and a persistent lifetime.

Connection string and endpoint references

It's common to express dependencies between project resources. Consider the following example code:

```C#
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache");

var apiservice = builder.AddProject<Projects.AspireApp_ApiService>
("apiservice");

builder.AddProject<Projects.AspireApp_Web>("webfrontend")
       .WithReference(cache)
       .WithReference(apiservice);
```

Project-to-project references are handled differently than resources that have well-defined connection strings. Instead of connection string being injected into the "webfrontend" resource, environment variables to support service discovery are injected.

Expand table

Method	Environment variable
`WithReference(cache)`	`ConnectionStrings__cache="localhost:62354"`
`WithReference(apiservice)`	`services__apiservice__http__0="http://localhost:5455"` `services__apiservice__https__0="https://localhost:7356"`

Adding a reference to the "apiservice" project results in service discovery environment variables being added to the frontend. This is because typically, project-to-project

communication occurs over HTTP/gRPC. For more information, see .NET Aspire service discovery.

To get specific endpoints from a ContainerResource or an ExecutableResource, use one of the following endpoint APIs:

- WithEndpoint
- WithHttpEndpoint
- WithHttpsEndpoint

Then call the GetEndpoint API to get the endpoint which can be used to reference the endpoint in the WithReference method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var customContainer = builder.AddContainer("myapp", "mycustomcontainer")
                             .WithHttpEndpoint(port: 9043, name:
"endpoint");

var endpoint = customContainer.GetEndpoint("endpoint");

var apiservice = builder.AddProject<Projects.AspireApp_ApiService>
("apiservice")
                        .WithReference(endpoint);
```

Expand table

Method	Environment variable
`WithReference(endpoint)`	`services__myapp__endpoint__0=https://localhost:9043`

The `port` parameter is the port that the container is listening on. For more information on container ports, see Container ports. For more information on service discovery, see .NET Aspire service discovery.

Service endpoint environment variable format

In the preceding section, the WithReference method is used to express dependencies between resources. When service endpoints result in environment variables being injected into the dependent resource, the format might not be obvious. This section provides details on this format.

When one resource depends on another resource, the app host injects environment variables into the dependent resource. These environment variables configure the dependent resource to connect to the resource it depends on. The format of the environment variables is specific to .NET Aspire and expresses service endpoints in a way that is compatible with Service Discovery.

Service endpoint environment variable names are prefixed with `services__` (double underscore), then the service name, the endpoint name, and finally the index. The index supports multiple endpoints for a single service, starting with `0` for the first endpoint and incrementing for each endpoint.

Consider the following environment variable examples:

Environment

```
services__apiservice__http__0
```

The preceding environment variable expresses the first HTTP endpoint for the `apiservice` service. The value of the environment variable is the URL of the service endpoint. A named endpoint might be expressed as follows:

Environment

```
services__apiservice__myendpoint__0
```

In the preceding example, the `apiservice` service has a named endpoint called `myendpoint`. The value of the environment variable is the URL of the service endpoint.

Reference existing resources

Some situations warrant that you reference an existing resource, perhaps one that is deployed to a cloud provider. For example, you might want to reference an Azure database. In this case, you'd rely on the Execution context to dynamically determine whether the app host is running in "run" mode or "publish" mode. If you're running locally and want to rely on a cloud resource, you can use the `IsRunMode` property to conditionally add the reference. You might choose to instead create the resource in publish mode. Some hosting integrations support providing a connection string directly, which can be used to reference an existing resource.

Likewise, there might be use cases where you want to integrate .NET Aspire into an existing solution. One common approach is to add the .NET Aspire app host project to an existing solution. Within your app host, you express dependencies by adding project

references to the app host and building out the app model. For example, one project might depend on another. These dependencies are expressed using the WithReference method. For more information, see Add .NET Aspire to an existing .NET app.

App host life cycles

The .NET Aspire app host exposes several life cycles that you can hook into by implementing the IDistributedApplicationLifecycleHook interface. The following lifecycle methods are available:

Expand table

Order	Method	Description
1	BeforeStartAsync	Executes before the distributed application starts.
2	AfterEndpointsAllocatedAsync	Executes after the orchestrator allocates endpoints for resources in the application model.
3	AfterResourcesCreatedAsync	Executes after the resource was created by the orchestrator.

While the app host provides life cycle hooks, you might want to register custom events. For more information, see Eventing in .NET Aspire.

Register a life cycle hook

To register a life cycle hook, implement the IDistributedApplicationLifecycleHook interface and register the hook with the app host using the AddLifecycleHook API:

C#

```
using Aspire.Hosting.Lifecycle;
using Microsoft.Extensions.Logging;

var builder = DistributedApplication.CreateBuilder(args);

builder.Services.AddLifecycleHook<LifecycleLogger>();

builder.Build().Run();

internal sealed class LifecycleLogger(ILogger<LifecycleLogger> logger)
    : IDistributedApplicationLifecycleHook
{
    public Task BeforeStartAsync(
        DistributedApplicationModel appModel, CancellationToken
cancellationToken = default)
```

```
    {
        logger.LogInformation("BeforeStartAsync");
        return Task.CompletedTask;
    }

    public Task AfterEndpointsAllocatedAsync(
        DistributedApplicationModel appModel, CancellationToken
cancellationToken = default)
    {
        logger.LogInformation("AfterEndpointsAllocatedAsync");
        return Task.CompletedTask;
    }

    public Task AfterResourcesCreatedAsync(
        DistributedApplicationModel appModel, CancellationToken
cancellationToken = default)
    {
        logger.LogInformation("AfterResourcesCreatedAsync");
        return Task.CompletedTask;
    }
}
```

The preceding code:

- Implements the IDistributedApplicationLifecycleHook interface as a `LifecycleLogger`.
- Registers the life cycle hook with the app host using the AddLifecycleHook API.
- Logs a message for all the events.

When this app host is run, the life cycle hook is executed for each event. The following output is generated:

Output

```
info: LifecycleLogger[0]
      BeforeStartAsync
info: Aspire.Hosting.DistributedApplication[0]
      Aspire version: 9.0.0
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application starting.
info: Aspire.Hosting.DistributedApplication[0]
      Application host directory is: ..\AspireApp\AspireApp.AppHost
info: LifecycleLogger[0]
      AfterEndpointsAllocatedAsync
info: Aspire.Hosting.DistributedApplication[0]
      Now listening on: https://localhost:17043
info: Aspire.Hosting.DistributedApplication[0]
      Login to the dashboard at https://localhost:17043/login?
t=d80f598bc8a64c7ee97328a1cbd55d72
info: LifecycleLogger[0]
      AfterResourcesCreatedAsync
```

```
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application started. Press Ctrl+C to shut down.
```

The preferred way to hook into the app host life cycle is to use the eventing API. For more information, see Eventing in .NET Aspire.

Execution context

The IDistributedApplicationBuilder exposes an execution context (DistributedApplicationExecutionContext), which provides information about the current execution of the app host. This context can be used to evaluate whether or not the app host is executing as "run" mode, or as part of a publish operation. Consider the following properties:

- IsRunMode: Returns `true` if the current operation is running.
- IsPublishMode: Returns `true` if the current operation is publishing.

This information can be useful when you want to conditionally execute code based on the current operation. Consider the following example that demonstrates using the `IsRunMode` property. In this case, an extension method is used to generate a stable node name for RabbitMQ for local development runs.

C#

```
private static IResourceBuilder<RabbitMQServerResource>
RunWithStableNodeName(
    this IResourceBuilder<RabbitMQServerResource> builder)
{
    if (builder.ApplicationBuilder.ExecutionContext.IsRunMode)
    {
        builder.WithEnvironment(context =>
        {
            // Set a stable node name so queue storage is consistent between
sessions
            var nodeName = $"{builder.Resource.Name}@localhost";
            context.EnvironmentVariables["RABBITMQ_NODENAME"] = nodeName;
        });
    }

    return builder;
}
```

The execution context is often used to conditionally add resources or connection strings that point to existing resources. Consider the following example that demonstrates conditionally adding Redis or a connection string based on the execution context:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var redis = builder.ExecutionContext.IsRunMode
    ? builder.AddRedis("redis")
    : builder.AddConnectionString("redis");

builder.AddProject<Projects.WebApplication>("api")
       .WithReference(redis);

builder.Build().Run();
```

In the preceding code:

- If the app host is running in "run" mode, a Redis container resource is added.
- If the app host is running in "publish" mode, a connection string is added.

This logic can easily be inverted to connect to an existing Redis resource when you're running locally, and create a new Redis resource when you're publishing.

> **ⓘ Important**
>
> .NET Aspire provides common APIs to control the modality of resource builders, allowing resources to behave differently based on the execution mode. The fluent APIs are prefixed with `RunAs*` and `PublishAs*`. The `RunAs*` APIs influence the local development (or run mode) behavior, whereas the `PublishAs*` APIs influence the publishing of the resource. For more information on how the Azure resources use these APIs, see Use existing Azure resources.

Resource relationships

Resource relationships link resources together. Relationships are informational and don't impact an app's runtime behavior. Instead, they're used when displaying details about resources in the dashboard. For example, relationships are visible in the dashboard's resource details, and `Parent` relationships control resource nesting on the resources page.

Relationships are automatically created by some app model APIs. For example:

- WithReference adds a relationship to the target resource with the type `Reference`.
- WaitFor adds a relationship to the target resource with the type `WaitFor`.

- Adding a database to a DB container creates a relationship from the database to the container with the type `Parent`.

Relationships can also be explicitly added to the app model using WithRelationship and WithParentRelationship.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var catalogDb = builder.AddPostgres("postgres")
                       .WithDataVolume()
                       .AddDatabase("catalogdb");

builder.AddProject<Projects.AspireApp_CatalogDbMigration>("migration")
       .WithReference(catalogDb)
       .WithParentRelationship(catalogDb);

builder.Build().Run();
```

The preceding example uses WithParentRelationship to configure `catalogdb` database as the `migration` project's parent. The `Parent` relationship is special because it controls resource nesting on the resource page. In this example, `migration` is nested under `catalogdb`.

> **Note**
>
> There's validation for parent relationships to prevent a resource from having multiple parents or creating a circular reference. These configurations can't be rendered in the UI, and the app model will throw an error.

See also

- .NET Aspire integrations overview
- .NET Aspire SDK
- Eventing in .NET Aspire
- Service discovery in .NET Aspire
- .NET Aspire service defaults
- Expressing external parameters
- .NET Aspire inner-loop networking overview

.js apps in .NET Aspire

ode.js and Node Package Manager (`npm`) apps in a
in this article demonstrates Angular , React , and
wing .NET Aspire APIs exist to support these scenarios
osting.NodeJS NuGet package:

.

wo APIs is that the former is used to host Node.js apps,
apps that execute from a *package.json* file's `scripts`
ing `npm run <script-name>` command.

with .NET Aspire, you

NET 8.0 or .NET
An OCI comp
- Docke

- JetBrains Rider with .NET Aspire plugin (Optional)

For more information, see .NET Aspire setup and tooling, and .NET Aspire SDK.

Additionally, you need to install Node.js on your machine. The sample app in this article was built with Node.js version 20.12.2 and npm version 10.5.1. To verify your Node.js and npm versions, run the following commands:

Node.js

```
node --version
```

Node.js

```
npm --version
```

To download Node.js (including `npm`), see the Node.js download page.

Clone sample source code

To clone the sample source code from GitHub, run the following command:

Bash

```
git clone https://github.com/dotnet/aspire-samples.git
```

After cloning the repository, navigate to the *samples/AspireWithJavaScript* folder:

Bash

```
cd samples/AspireWithJavaScript
```

From this directory, there are six child directories described in the following list:

- **AspireJavaScript.Angular**: An Angular app that consumes the weather forecast API and displays the data in a table.
- **AspireJavaScript.AppHost**: A .NET Aspire project that orchestrates the other apps in this sample. For more information, see .NET Aspire orchestration overview.
- **AspireJavaScript.MinimalApi**: An HTTP API that returns randomly generated weather forecast data.
- **AspireJavaScript.React**: A React app that consumes the weather forecast API and displays the data in a table.

- **AspireJavaScript.ServiceDefaults**: The default shared project for .NET Aspire projects. For more information, see .NET Aspire service defaults.
- **AspireJavaScript.Vue**: A Vue app that consumes the weather forecast API and displays the data in a table.

Install client dependencies

The sample app demonstrates how to use JavaScript client apps that are built on top of Node.js. Each client app was written either using a `npm create` template command or manually. The following table lists the template commands used to create each client app, along with the default port:

Expand table

App type	Create template command	Default port
Angular	`npm create @angular@latest`	4200
React	Didn't use a template.	PORT env var
Vue	`npm create vue@latest`	5173

> **Tip**
>
> You don't need to run any of these commands, since the sample app already includes the clients. Instead, this is a point of reference from which the clients were created. For more information, see **npm-init**.

To run the app, you first need to install the dependencies for each client. To do so, navigate to each client folder and run npm install (or the install alias npm i) commands.

Install Angular dependencies

Node.js

```
npm i ./AspireJavaScript.Angular/
```

For more information on the Angular app, see explore the Angular client.

Install React dependencies

Node.js

```
npm i ./AspireJavaScript.React/
```

For more information on the React app, see explore the React client.

Install Vue dependencies

Node.js

```
npm i ./AspireJavaScript.Vue/
```

For more information on the Vue app, see explore the Vue client.

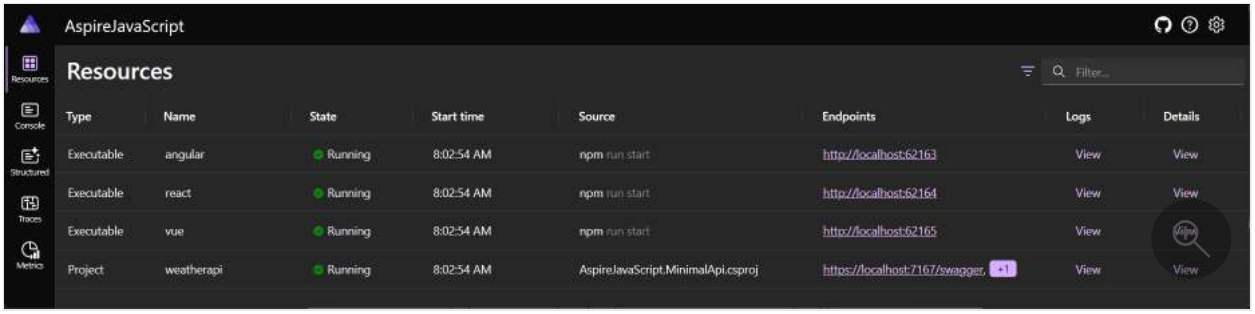
Run the sample app

To run the sample app, call the dotnet run command given the orchestrator app host *AspireJavaScript.AppHost.csproj* as the `--project` switch:

.NET CLI

```
dotnet run --project
./AspireJavaScript.AppHost/AspireJavaScript.AppHost.csproj
```

The .NET Aspire dashboard launches in your default browser, and each client app endpoint displays under the **Endpoints** column of the **Resources** page. The following image depicts the dashboard for this sample app:



The `weatherapi` service endpoint resolves to a Swagger UI page that documents the HTTP API. Each client app consumes this service to display the weather forecast data. You can view each client app by navigating to the corresponding endpoint in the .NET Aspire dashboard. Their screenshots and the modifications made from the template starting point are detailed in the following sections.

In the same terminal session that you used to run the app, press `Ctrl` +

The project file also defines a build target that ensures that the npm dependencies are installed before the app host is built. The app host code (*Program.cs*) declares the client app resources using the AddNpmApp(IDistributedApplicationBuilder, String, String, String, String[]) API.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var weatherApi = builder.AddProject<Projects.AspireJavaScript_MinimalApi>
("weatherapi")
    .WithExternalHttpEndpoints();

builder.AddNpmApp("angular", "../AspireJavaScript.Angular")
    .WithReference(weatherApi)
    .WaitFor(weatherApi)
    .WithHttpEndpoint(env: "PORT")
    .WithExternalHttpEndpoints()
    .PublishAsDockerFile();

builder.AddNpmApp("react", "../AspireJavaScript.React")
    .WithReference(weatherApi)
    .WaitFor(weatherApi)
    .WithEnvironment("BROWSER", "none") // Disable opening browser on npm
start
    .WithHttpEndpoint(env: "PORT")
    .WithExternalHttpEndpoints()
    .PublishAsDockerFile();

builder.AddNpmApp("vue", "../AspireJavaScript.Vue")
    .WithReference(weatherApi)
    .WaitFor(weatherApi)
    .WithHttpEndpoint(env: "PORT")
    .WithExternalHttpEndpoints()
    .PublishAsDockerFile();

builder.AddNpmApp("reactvite", "../AspireJavaScript.Vite")
    .WithReference(weatherApi)
    .WithEnvironment("BROWSER", "none")
    .WithHttpEndpoint(env: "VITE_PORT")
    .WithExternalHttpEndpoints()
    .PublishAsDockerFile();

builder.Build().Run();
```

The preceding code:

- Creates a DistributedApplicationBuilder.
- Adds the "weatherapi" service as a project to the app host.
  - Marks the HTTP endpoints as external.

- With a reference to the "weatherapi" service, adds the "angular", "react", and "vue" client apps as npm apps.
  - Each client app is configured to run on a different container port, and uses the `PORT` environment variable to determine the port.
  - All client apps also rely on a *Dockerfile* to build their container image and are configured to express themselves in the publishing manifest as a container from the PublishAsDockerFile API.

For more information on inner-loop networking, see .NET Aspire inner-loop networking overview. For more information on deploying apps, see .NET Aspire manifest format for deployment tool builders.

When the app host orchestrates the launch of each client app, it uses the `npm run start` command. This command is defined in the `scripts` section of the *package.json* file for each client app. The `start` script is used to start the client app on the specified port. Each client app relies on a proxy to request the "weatherapi" service.

The proxy is configured in:

- The *proxy.conf.js* file for the Angular client.
- The *webpack.config.js* file for the React client.
- The *vite.config.ts* file for the Vue client.

Explore the Angular client

There are several key modifications from the original Angular template. The first is the addition of a *proxy.conf.js* file. This file is used to proxy requests from the Angular client to the "weatherapi" service.

JavaScript

```
module.exports = {
  "/api": {
    target:
      process.env["services__weatherapi__https__0"] ||
      process.env["services__weatherapi__http__0"],
    secure: process.env["NODE_ENV"] !== "development",
    pathRewrite: {
      "^/api": "",
    },
  },
};
```

The .NET Aspire app host sets the `services__weatherapi__http__0` environment variable, which is used to resolve the "weatherapi" service endpoint. The preceding configuration proxies HTTP requests that start with `/api` to the target URL specified in the environment variable.

Then include the proxy file to in the *angular.json* file. Update the `serve` target to include the `proxyConfig` option, referencing to the created *proxy.conf.js* file. The Angular CLI will now use the proxy configuration while serving the Angular client app.

JavaScript

```
"serve": {
  "builder": "@angular-devkit/build-angular:dev-server",
  "configurations": {
    "production": {
      "buildTarget": "weather:build:production"
    },
    "development": {
      "buildTarget": "weather:build:development"
    }
  },
  "defaultConfiguration": "development",
  "options": {
    "proxyConfig": "proxy.conf.js"
  }
},
```

The third update is to the *package.json* file. This file is used to configure the Angular client to run on a different port than the default port. This is achieved by using the `PORT` environment variable, and the `run-script-os` npm package to set the port.

JSON

```
{
  "name": "angular-weather",
  "version": "0.0.0",
  "engines": {
    "node": ">=20.12"
  },
  "scripts": {
    "ng": "ng",
    "start": "run-script-os",
    "start:win32": "ng serve --port %PORT%",
    "start:default": "ng serve --port $PORT",
    "build": "ng build",
    "watch": "ng build --watch --configuration development",
    "test": "ng test"
  },
  "private": true,
```

```
  "dependencies": {
    "@angular/animations": "^19.2.1",
    "@angular/common": "^19.2.1",
    "@angular/compiler": "^19.2.1",
    "@angular/core": "^19.2.1",
    "@angular/forms": "^19.2.1",
    "@angular/platform-browser": "^19.2.1",
    "@angular/platform-browser-dynamic": "^19.2.1",
    "@angular/router": "^19.2.1",
    "rxjs": "~7.8.2",
    "tslib": "^2.8.1",
    "zone.js": "~0.15.0"
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "^19.2.1",
    "@angular/cli": "^19.2.1",
    "@angular/compiler-cli": "^19.2.1",
    "@types/jasmine": "~5.1.7",
    "jasmine-core": "~5.6.0",
    "karma": "~6.4.4",
    "karma-chrome-launcher": "~3.2.0",
    "karma-coverage": "~2.2.1",
    "karma-jasmine": "~5.1.0",
    "karma-jasmine-html-reporter": "~2.1.0",
    "typescript": "~5.8.2",
    "run-script-os": "^1.1.6"
  }
}
```

The `scripts` section of the *package.json* file is used to define the `start` script. This script is used by the `npm start` command to start the Angular client app. The `start` script is configured to use the `run-script-os` package to set the port, which delegates to the `ng serve` command passing the appropriate `--port` switch based on the OS-appropriate syntax.

In order to make HTTP calls to the "weatherapi" service, the Angular client app needs to be configured to provide the Angular `HttpClient` for dependency injection. This is achieved by using the `provideHttpClient` helper function while configuring the application in the *app.config.ts* file.

TypeScript

```
import { ApplicationConfig } from '@angular/core';
import { provideHttpClient } from '@angular/common/http';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [
```

```
    provideRouter(routes),
    provideHttpClient()
  ]
};
```

Finally, the Angular client app needs to call the `/api/WeatherForecast` endpoint to retrieve the weather forecast data. There are several HTML, CSS, and TypeScript updates, all of which are made to the following files:

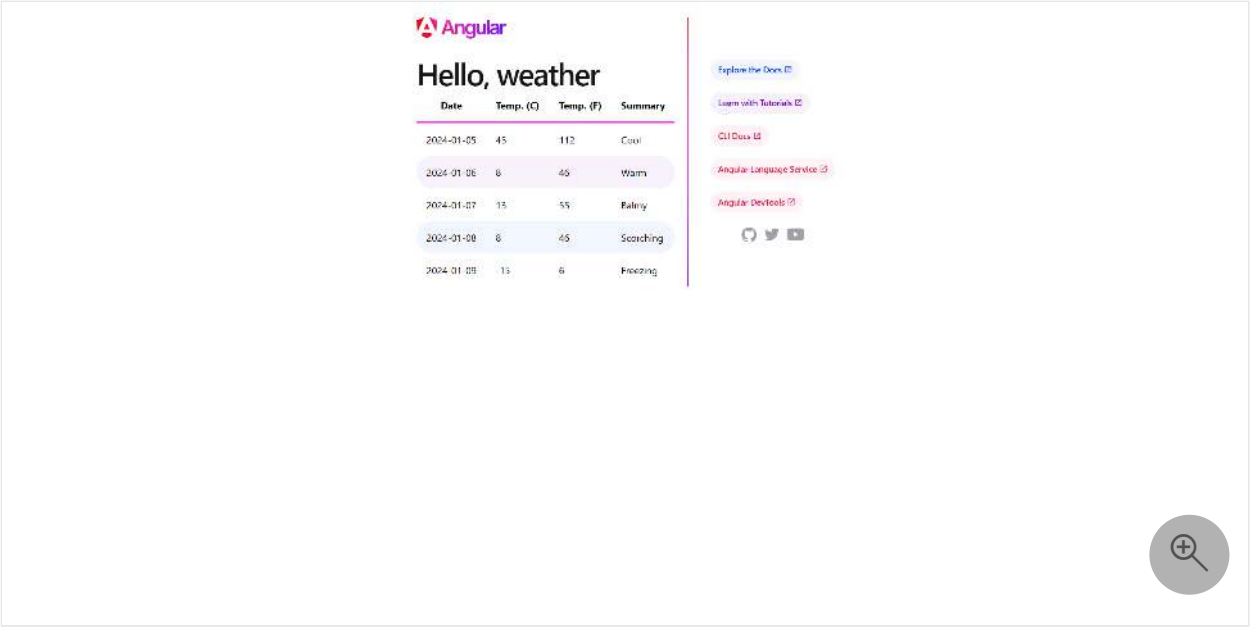
- *app.component.css*: Update the CSS to style the table.
- *app.component.html*: Update the HTML to display the weather forecast data in a table.
- *app.component.ts*: Update the TypeScript to call the /api/WeatherForecast endpoint and display the data in the table.

TypeScript

```typescript
import { Component, Injectable } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { HttpClient } from '@angular/common/http';
import { WeatherForecasts } from '../types/weatherForecast';

@Injectable()
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title = 'weather';
  forecasts: WeatherForecasts = [];

  constructor(private http: HttpClient) {
    http.get<WeatherForecasts>('api/weatherforecast').subscribe({
      next: result => this.forecasts = result,
      error: console.error
    });
  }
}
```

Angular app running

To visualize the Angular client app, navigate to the "angular" endpoint in the .NET Aspire dashboard. The following image depicts the Angular client app:



Explore the React client

The React app wasn't written using a template, and instead was written manually. The complete source code can be found in the dotnet/aspire-samples repository. Some of the key points of interest are found in the *src/App.js* file:

JavaScript

```
import { useEffect, useState } from "react";
import "./App.css";

function App() {
  const [forecasts, setForecasts] = useState([]);

  const requestWeather = async () => {
    const weather = await fetch("api/weatherforecast");
    console.log(weather);

    const weatherJson = await weather.json();
    console.log(weatherJson);

    setForecasts(weatherJson);
  };

  useEffect(() => {
    requestWeather();
  }, []);

  return (
    <div className="App">
      <header className="App-header">
        <h1>React Weather</h1>
        <table>
          <thead>
```

```
            <tr>
              <th>Date</th>
              <th>Temp. (C)</th>
              <th>Temp. (F)</th>
              <th>Summary</th>
            </tr>
          </thead>
          <tbody>
            {(
              forecasts ?? [
                {
                  date: "N/A",
                  temperatureC: "",
                  temperatureF: "",
                  summary: "No forecasts",
                },
              ]
            ).map((w) => {
              return (
                <tr key={w.date}>
                  <td>{w.date}</td>
                  <td>{w.temperatureC}</td>
                  <td>{w.temperatureF}</td>
                  <td>{w.summary}</td>
                </tr>
              );
            })}
          </tbody>
        </table>
      </header>
    </div>
  );
}

export default App;
```

The `App` function is the entry point for the React client app. It uses the `useState` and `useEffect` hooks to manage the state of the weather forecast data. The `fetch` API is used to make an HTTP request to the `/api/WeatherForecast` endpoint. The response is then converted to JSON and set as the state of the weather forecast data.

JavaScript

```
const HTMLWebpackPlugin = require("html-webpack-plugin");

module.exports = (env) => {
  return {
    entry: "./src/index.js",
    devServer: {
      port: env.PORT || 4001,
      allowedHosts: "all",
      proxy: [
```

```
        {
          context: ["/api"],
          target:
            process.env.services__weatherapi__https__0 ||
            process.env.services__weatherapi__http__0,
          pathRewrite: { "^/api": "" },
          secure: false,
        },
      ],
    },
    output: {
      path: `${__dirname}/dist`,
      filename: "bundle.js",
    },
    plugins: [
      new HTMLWebpackPlugin({
        template: "./src/index.html",
        favicon: "./src/favicon.ico",
      }),
    ],
    module: {
      rules: [
        {
          test: /\.js$/,
          exclude: /node_modules/,
          use: {
            loader: "babel-loader",
            options: {
              presets: [
                "@babel/preset-env",
                ["@babel/preset-react", { runtime: "automatic" }],
              ],
            },
          },
        },
        {
          test: /\.css$/,
          exclude: /node_modules/,
          use: ["style-loader", "css-loader"],
        },
      ],
    },
  };
};
```

The preceding code defines the `module.exports` as follows:

- The `entry` property is set to the *src/index.js* file.
- The `devServer` relies on a proxy to forward requests to the "weatherapi" service, sets the port to the `PORT` environment variable, and allows all hosts.
- The `output` results in a *dist* folder with a *bundle.js* file.

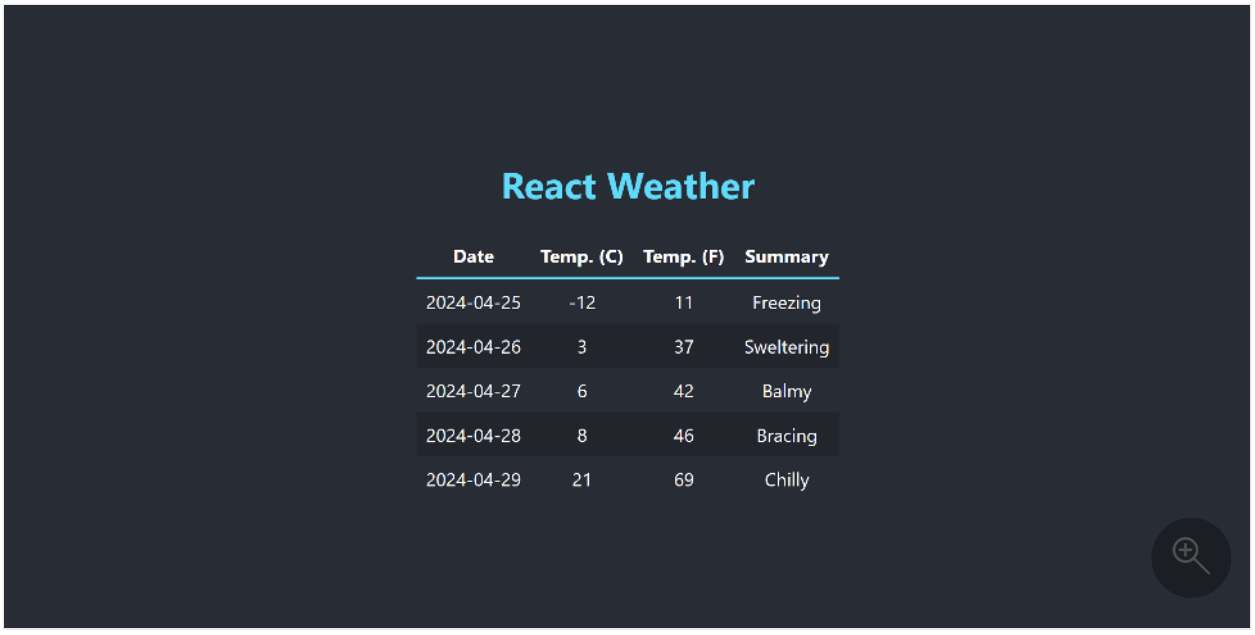
- The `plugins` set the *src/index.html* file as the template, and expose the *favicon.ico* file.

The final updates are to the following files:

- *App.css*: Update the CSS to style the table.
- *App.js*: Update the JavaScript to call the /api/WeatherForecast endpoint and display the data in the table.

React app running

To visualize the React client app, navigate to the "react" endpoint in the .NET Aspire dashboard. The following image depicts the React client app:



Explore the Vue client

There are several key modifications from the original Vue template. The primary updates were the addition of the `fetch` call in the *TheWelcome.vue* file to retrieve the weather forecast data from the `/api/WeatherForecast` endpoint. The following code snippet demonstrates the `fetch` call:

HTML

```
<script lang="ts">
interface WeatherForecast {
  date: string
  temperatureC: number
  temperatureF: number
  summary: string
};
```

```
type Forecasts = WeatherForecast[];

export default {
  name: 'TheWelcome',
  data() {
    return {
      forecasts: [],
      loading: true,
      error: null
    }
  },
  mounted() {
    fetch('api/weatherforecast')
      .then(response => response.json())
      .then(data => {
        this.forecasts = data
      })
      .catch(error => {
        this.error = error
      })
      .finally(() => (this.loading = false))
  }
}
</script>

<template>
  <table>
    <thead>
      <tr>
        <th>Date</th>
        <th>Temp. (C)</th>
        <th>Temp. (F)</th>
        <th>Summary</th>
      </tr>
    </thead>
    <tbody>
      <tr v-for="forecast in (forecasts as Forecasts)">
        <td>{{ forecast.date }}</td>
        <td>{{ forecast.temperatureC }}</td>
        <td>{{ forecast.temperatureF }}</td>
        <td>{{ forecast.summary }}</td>
      </tr>
    </tbody>
  </table>
</template>

<style>
table {
  border: none;
  border-collapse: collapse;
}

th {
  font-size: x-large;
```

```
  font-weight: bold;
  border-bottom: solid .2rem hsla(160, 100%, 37%, 1);
}

th,
td {
  padding: 1rem;
}

td {
  text-align: center;
  font-size: large;
}

tr:nth-child(even) {
  background-color: var(--vt-c-black-soft);
}
</style>
```

As the `TheWelcome` integration is `mounted`, it calls the `/api/weatherforecast` endpoint to retrieve the weather forecast data. The response is then set as the `forecasts` data property. To set the server port, the Vue client app uses the `PORT` environment variable. This is achieved by updating the *vite.config.ts* file:

TypeScript

```
import { fileURLToPath, URL } from 'node:url'

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [
    vue(),
  ],
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url))
    }
  },
  server: {
    host: true,
    port: parseInt(process.env.PORT ?? "5173"),
    proxy: {
      '/api': {
        target: process.env.services__weatherapi__https__0 ||
process.env.services__weatherapi__http__0,
        changeOrigin: true,
        rewrite: path => path.replace(/^\/api/, ''),
        secure: false
      }
```

```
    }
  }
})
```

Additionally, the Vite config specifies the `server.proxy` property to forward requests to the "weatherapi" service. This is achieved by using the `services__weatherapi__http__0` environment variable, which is set by the .NET Aspire app host.

The final update from the template is made to the *TheWelcome.vue* file. This file calls the `/api/WeatherForecast` endpoint to retrieve the weather forecast data, and displays the data in a table. It includes CSS, HTML, and TypeScript updates.

Vue app running

To visualize the Vue client app, navigate to the "vue" endpoint in the .NET Aspire dashboard. The following image depicts the Vue client app:

Deployment considerations

The sample source code for this article is designed to run locally. Each client app deploys as a container image. The *Dockerfile* for each client app is used to build the container image. Each *Dockerfile* is identical, using a multistage build to create a production-ready container image.

Dockerfile

```
FROM node:20 as build

WORKDIR /app
```

```
COPY package.json package.json
COPY package-lock.json package-lock.json

RUN npm install

COPY . .

RUN npm run build

FROM nginx:alpine

COPY --from=build /app/default.conf.template
/etc/nginx/templates/default.conf.template
COPY --from=build /app/dist/weather/browser /usr/share/nginx/html

# Expose the default nginx port
EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

The client apps are currently configured to run as true SPA apps, and aren't configured to run in a server-side rendered (SSR) mode. They sit behind **nginx**, which is used to serve the static files. They use a *default.conf.template* file to configure **nginx** to proxy requests to the client app.

nginx

```
server {
    listen       ${PORT};
    listen  [::]:${PORT};
    server_name  localhost;

    access_log  /var/log/nginx/server.access.log  main;

    location / {
        root /usr/share/nginx/html;
        try_files $uri $uri/ /index.html;
    }

    location /api/ {
        proxy_pass ${services__weatherapi__https__0};
        proxy_http_version 1.1;
        proxy_ssl_server_name on;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        rewrite ^/api(/.*)$ $1 break;
    }
}
```

Node.js server app considerations

While this article focuses on client apps, you might have scenarios where you need to host a Node.js server app. The same semantics are required to host a Node.js server app as a SPA client app. The .NET Aspire app host requires a package reference to the Aspire.Hosting.NodeJS NuGet package and the code needs to call either `AddNodeApp` or `AddNpmApp`. These APIs are useful for adding existing JavaScript apps to the .NET Aspire app host.

When configuring secrets and passing environment variables to JavaScript-based apps, whether they are client or server apps, use parameters. For more information, see .NET Aspire: External parameters—secrets.

Use the OpenTelemetry JavaScript SDK

To export OpenTelemetry logs, traces, and metrics from a Node.js server app, you use the OpenTelemetry JavaScript SDK.

For a complete example of a Node.js server app using the OpenTelemetry JavaScript SDK, you can refer to the Code Samples: .NET Aspire Node.js sample page. Consider the sample's *instrumentation.js* file, which demonstrates how to configure the OpenTelemetry JavaScript SDK to export logs, traces, and metrics:

JavaScript

```
import { env } from 'node:process';
import { NodeSDK } from '@opentelemetry/sdk-node';
import { OTLPTraceExporter } from '@opentelemetry/exporter-trace-otlp-grpc';
import { OTLPMetricExporter } from '@opentelemetry/exporter-metrics-otlp-grpc';
import { OTLPLogExporter } from '@opentelemetry/exporter-logs-otlp-grpc';
import { SimpleLogRecordProcessor } from '@opentelemetry/sdk-logs';
import { PeriodicExportingMetricReader } from '@opentelemetry/sdk-metrics';
import { HttpInstrumentation } from '@opentelemetry/instrumentation-http';
import { ExpressInstrumentation } from '@opentelemetry/instrumentation-express';
import { RedisInstrumentation } from '@opentelemetry/instrumentation-redis-4';
import { diag, DiagConsoleLogger, DiagLogLevel } from '@opentelemetry/api';
import { credentials } from '@grpc/grpc-js';

const environment = process.env.NODE_ENV || 'development';

// For troubleshooting, set the log level to DiagLogLevel.DEBUG
//diag.setLogger(new DiagConsoleLogger(), environment === 'development' ? DiagLogLevel.INFO : DiagLogLevel.WARN);

const otlpServer = env.OTEL_EXPORTER_OTLP_ENDPOINT;

if (otlpServer) {
```

```
    console.log(`OTLP endpoint: ${otlpServer}`);

    const isHttps = otlpServer.startsWith('https://');
    const collectorOptions = {
        credentials: !isHttps
            ? credentials.createInsecure()
            : credentials.createSsl()
    };

    const sdk = new NodeSDK({
        traceExporter: new OTLPTraceExporter(collectorOptions),
        metricReader: new PeriodicExportingMetricReader({
            exportIntervalMillis: environment === 'development' ? 5000 :
10000,
            exporter: new OTLPMetricExporter(collectorOptions),
        }),
        logRecordProcessor: new SimpleLogRecordProcessor({
            exporter: new OTLPLogExporter(collectorOptions)
        }),
        instrumentations: [
            new HttpInstrumentation(),
            new ExpressInstrumentation(),
            new RedisInstrumentation()
        ],
    });

    sdk.start();
}
```

> **Tip**
>
> To configure the .NET Aspire dashboard OTEL CORS settings, see the .NET Aspire dashboard OTEL CORS settings page.

Summary

While there are several considerations that are beyond the scope of this article, you learned how to build .NET Aspire projects that use Node.js and Node Package Manager (`npm`). You also learned how to use the AddNpmApp APIs to host Node.js apps and apps that execute from a *package.json* file, respectively. Finally, you learned how to use the `npm` CLI to create Angular, React, and Vue client apps, and how to configure them to run on different ports.

See also

- Code Samples: .NET Aspire with Angular, React, and Vue

- Code Samples: .NET Aspire Node.js App

Orchestrate Python apps in .NET Aspire

Article • 11/12/2024

In this article, you learn how to use Python apps in a .NET Aspire app host. The sample app in this article demonstrates launching a Python application. The Python extension for .NET Aspire requires the use of virtual environments.

Prerequisites

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0
- An OCI compliant container runtime, such as:
  - Docker Desktop or Podman. For more information, see Container runtime.
- An Integrated Developer Environment (IDE) or code editor, such as:
  - Visual Studio 2022 version 17.9 or higher (Optional)
  - Visual Studio Code (Optional)
    - C# Dev Kit: Extension (Optional)
  - JetBrains Rider with .NET Aspire plugin (Optional)

For more information, see .NET Aspire setup and tooling, and .NET Aspire SDK.

Additionally, you need to install Python on your machine. The sample app in this article was built with Python version 3.12.4 and pip version 24.1.2. To verify your Python and pip versions, run the following commands:

Python

```
python --version
```

Python

```
pip --version
```

To download Python (including `pip`), see the Python download page.

Create a .NET Aspire project using the template

To get started launching a Python project in .NET Aspire first use the starter template to create a .NET Aspire application host:

.NET CLI

```
dotnet new aspire -o PythonSample
```

In the same terminal session, change directories into the newly created project:

.NET CLI

```
cd PythonSample
```

Once the template has been created launch the app host with the following command to ensure that the app host and the .NET Aspire dashboard launches successfully:

.NET CLI

```
dotnet run --project PythonSample.AppHost/PythonSample.AppHost.csproj
```

Once the app host starts it should be possible to click on the dashboard link in the console output. At this point the dashboard will not show any resources. Stop the app host by pressing `Ctrl` + `C` in the terminal.

Prepare a Python app

From your previous terminal session where you created the .NET Aspire solution, create a new directory to contain the Python source code.

Console

```
mkdir hello-python
```

Change directories into the newly created *hello-python* directory:

Console

```
cd hello-python
```

Initialize the Python virtual environment

To work with Python apps, they need to be within a virtual environment. To create a virtual environment, run the following command:

Python

```
python -m venv .venv
```

For more information on virtual environments, see the Python: Install packages in a virtual environment using pip and venv.

To activate the virtual environment, enabling installation and usage of packages, run the following command:

Unix/macOS

Bash

```
source .venv/bin/activate
```

Ensure that pip within the virtual environment is up-to-date by running the following command:

Python

```
python -m pip install --upgrade pip
```

Install Python packages

Install the Flask package by creating a *requirements.txt* file in the *hello-python* directory and adding the following line:

Python

```
Flask==3.0.3
```

Then, install the Flask package by running the following command:

Python

```
python -m pip install -r requirements.txt
```

After Flask is installed, create a new file named *main.py* in the *hello-python* directory and add the following code:

Python

```python
import os
import flask

app = flask.Flask(__name__)

@app.route('/', methods=['GET'])
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    port = int(os.environ.get('PORT', 8111))
    app.run(host='0.0.0.0', port=port)
```

The preceding code creates a simple Flask app that listens on port 8111 and returns the message `"Hello, World!"` when the root endpoint is accessed.

Update the app host project

Install the Python hosting package by running the following command:

.NET CLI

```dotnetcli
dotnet add ../PythonSample.AppHost/PythonSample.AppHost.csproj package
Aspire.Hosting.Python --version 9.0.0
```

After the package is installed, the project XML should have a new package reference similar to the following:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <Sdk Name="Aspire.AppHost.Sdk" Version="9.1.0" />

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net9.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <IsAspireHost>true</IsAspireHost>
    <UserSecretsId>5fd92a87-fff8-4a09-9f6e-2c0d656e25ba</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Aspire.Hosting.AppHost" Version="9.1.0" />
    <PackageReference Include="Aspire.Hosting.Python" Version="9.1.0" />
  </ItemGroup>
```

```
</Project>
```

Update the app host *Program.cs* file to include the Python project, by calling the `AddPythonApp` API and specifying the project name, project path, and the entry point file:

C#

```
using Microsoft.Extensions.Hosting;

var builder = DistributedApplication.CreateBuilder(args);

#pragma warning disable ASPIREHOSTINGPYTHON001
var pythonapp = builder.AddPythonApp("hello-python", "../hello-python",
"main.py")
       .WithHttpEndpoint(env: "PORT")
       .WithExternalHttpEndpoints()
       .WithOtlpExporter();
#pragma warning restore ASPIREHOSTINGPYTHON001

if (builder.ExecutionContext.IsRunMode &&
builder.Environment.IsDevelopment())
{
    pythonapp.WithEnvironment("DEBUG", "True");
}

builder.Build().Run();
```

> **ⓘ Important**
>
> The `AddPythonApp` API is experimental and may change in future releases. For more information, see ASPIREHOSTINGPYTHON001.

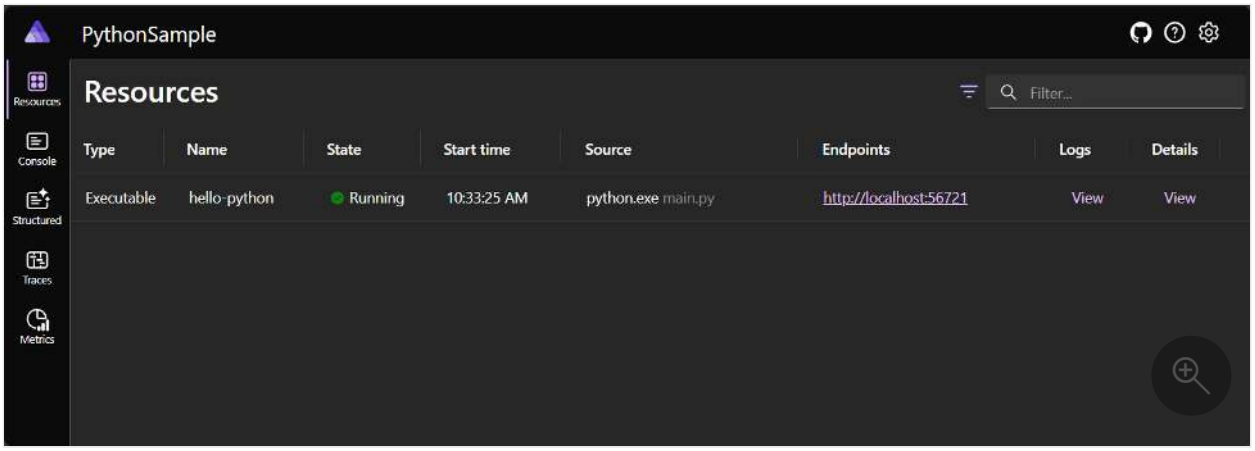
Run the app

Now that you've added the Python hosting package, updated the app host *Program.cs* file, and created a Python project, you can run the app host:

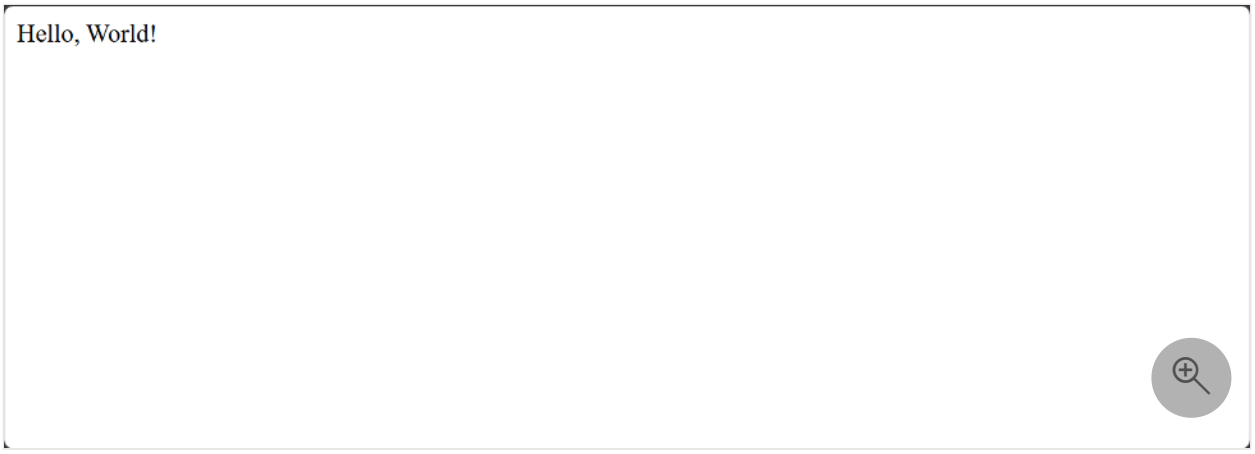
.NET CLI

```
dotnet run --project ../PythonSample.AppHost/PythonSample.AppHost.csproj
```

Launch the dashboard by clicking the link in the console output. The dashboard should display the Python project as a resource.



Select the **Endpoints** link to open the `hello-python` endpoint in a new browser tab. The browser should display the message "Hello, World!":



Stop the app host by pressing `Ctrl` + `C` in the terminal.

Add telemetry support.

To add a bit of observability, add telemetry to help monitor the dependant Python app. In the Python project, add the following OpenTelemetry package as a dependency in the *requirements.txt* file:

Python

```
Flask==3.0.3
opentelemetry-distro
opentelemetry-exporter-otlp-proto-grpc
opentelemetry-instrumentation-flask
gunicorn
```

The preceding requirement update, adds the OpenTelemetry package and the OTLP exporter. Next, re-install the Python app requirements into the virtual environment by running the following command:

Python

```
python -m pip install -r requirements.txt
```

The preceding command installs the OpenTelemetry package and the OTLP exporter, in the virtual environment. Update the Python app to include the OpenTelemetry code, by replacing the existing *main.py* code with the following:

Python

```
import os
import logging
import flask
from opentelemetry import trace
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import
OTLPSpanExporter
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.instrumentation.flask import FlaskInstrumentor

app = flask.Flask(__name__)

trace.set_tracer_provider(TracerProvider())
otlpExporter = OTLPSpanExporter()
processor = BatchSpanProcessor(otlpExporter)
trace.get_tracer_provider().add_span_processor(processor)

FlaskInstrumentor().instrument_app(app)

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@app.route('/', methods=['GET'])
def hello_world():
    logger.info("request received!")
    return 'Hello, World!'

if __name__ == '__main__':
    port = int(os.environ.get('PORT', 8111))
    debug = bool(os.environ.get('DEBUG', False))
    host = os.environ.get('HOST', '127.0.0.1')
    app.run(port=port, debug=debug, host=host)
```

Update the app host project's *launchSettings.json* file to include the `ASPIRE_ALLOW_UNSECURED_TRANSPORT` environment variable:

JSON

```
{
  "$schema": "https://json.schemastore.org/launchsettings.json",
```

The `ASPIRE_ALLOW_UNSECURED_TRANSPORT` variable is required because when running locally the OpenTelemetry client in Python rejects the local development certificate. Launch the *app host* again:

.NET CLI

Summary

While there are several considerations that are beyond the scope of this article, you learned how to build .NET Aspire solution that integrates with Python. You also learned how to use the `AddPythonApp` API to host Python apps.

See also

- GitHub: .NET Aspire Samples—Python hosting integration

App host configuration

Article • 11/22/2024

The app host project configures and starts your distributed application (DistributedApplication). When a `DistributedApplication` runs it reads configuration from the app host. Configuration is loaded from environment variables that are set on the app host and DistributedApplicationOptions.

Configuration includes:

- Settings for hosting the resource service, such as the address and authentication options.
- Settings used to start the .NET Aspire dashboard, such the dashboard's frontend and OpenTelemetry Protocol (OTLP) addresses.
- Internal settings that .NET Aspire uses to run the app host. These are set internally but can be accessed by integrations that extend .NET Aspire.

App host configuration is provided by the app host launch profile. The app host has a launch settings file call *launchSettings.json* which has a list of launch profiles. Each launch profile is a collection of related options which defines how you would like `dotnet` to start your application.

JSON

```
{
  "$schema": "https://json.schemastore.org/launchsettings.json",
  "profiles": {
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:17134;http://localhost:15170",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "DOTNET_ENVIRONMENT": "Development",
        "DOTNET_DASHBOARD_OTLP_ENDPOINT_URL": "https://localhost:21030",
        "DOTNET_RESOURCE_SERVICE_ENDPOINT_URL": "https://localhost:22057"
      }
    }
  }
}
```

The preceding launch settings file:

- Has one launch profile named `https`.
- Configures an .NET Aspire app host project:
  - The `applicationUrl` property configures the dashboard launch address (`ASPNETCORE_URLS`).
  - Environment variables such as `DOTNET_DASHBOARD_OTLP_ENDPOINT_URL` and `DOTNET_RESOURCE_SERVICE_ENDPOINT_URL` are set on the app host.

For more information, see .NET Aspire and launch profiles.

> **Note**
>
> Configuration described on this page is for .NET Aspire app host project. To configure the standalone dashboard, see dashboard configuration.

Common configuration

Expand table

Option	Default value	Description
`ASPIRE_ALLOW_UNSECURED_TRANSPORT`	`false`	Allows communication with the app host without https. `ASPNETCORE_URLS` (dashboard address) and `DOTNET_RESOURCE_SERVICE_ENDPOINT_URL` (app host resource service address) must be secured with HTTPS unless true.
`DOTNET_ASPIRE_CONTAINER_RUNTIME`	`docker`	Allows the user of alternative container runtimes for resources backed by containers. Possible values are `docker` (default) or `podman`. See Setup and tooling overview for more details.

Resource service

A resource service is hosted by the app host. The resource service is used by the dashboard to fetch information about resources which are being orchestrated by .NET Aspire.

Expand table

Option	Default value	Description
`DOTNET_RESOURCE_SERVICE_ENDPOINT_URL`	`null`	Configures the address of the resource service hosted by the app host. Automatically generated with *launchSettings.json* to have a random port on localhost. For example, `https://localhost:17037`.
`DOTNET_DASHBOARD_RESOURCESERVICE_APIKEY`	Automatically generated 128-bit entropy token.	The API key used to authenticate requests made to the app host's resource service. The API key is required if the app host is in run mode, the dashboard isn't disabled, and the dashboard isn't configured to allow anonymous access with `DOTNET_DASHBOARD_UNSECURED_ALLOW_ANONYMOUS`.

Dashboard

By default, the dashboard is automatically started by the app host. The dashboard supports its own set of configuration, and some settings can be configured from the app host.

Expand table

Option	Default value	Description
`ASPNETCORE_URLS`	`null`	Dashboard address. Must be `https` unless `ASPIRE_ALLOW_UNSECURED_TRANSPORT` or `DistributedApplicationOptions.AllowUnsecuredTransport` is true. Automatically generated with *launchSettings.json* to have a random port on localhost. The value in launch settings is set on the `applicationUrls` property.
`ASPNETCORE_ENVIRONMENT`	`Production`	Configures the environment the dashboard runs as. For more information, see Use multiple environments in ASP.NET Core.

Option	Default value	Description
`DOTNET_DASHBOARD_OTLP_ENDPOINT_URL`	`http://localhost:18889` if no gRPC endpoint is configured.	Configures the dashboard OTLP gRPC address. Used by the dashboard to receive telemetry over OTLP. Set on resources as the `OTEL_EXPORTER_OTLP_ENDPOINT` env var. The `OTEL_EXPORTER_OTLP_PROTOCOL` env var is `grpc`. Automatically generated with *launchSettings.json* to have a random port on localhost.
`DOTNET_DASHBOARD_OTLP_HTTP_ENDPOINT_URL`	`null`	Configures the dashboard OTLP HTTP address. Used by the dashboard to receive telemetry over OTLP. If only `DOTNET_DASHBOARD_OTLP_HTTP_ENDPOINT_URL` is configured then it is set on resources as the `OTEL_EXPORTER_OTLP_ENDPOINT` env var. The `OTEL_EXPORTER_OTLP_PROTOCOL` env var is `http/protobuf`.
`DOTNET_DASHBOARD_CORS_ALLOWED_ORIGINS`	`null`	Overrides the CORS allowed origins configured in the dashboard. This setting replaces the default behavior of calculating allowed origins based on resource endpoints.
`DOTNET_DASHBOARD_FRONTEND_BROWSERTOKEN`	Automatically generated 128-bit entropy token.	Configures the frontend browser token. This is the value that must be entered to access the dashboard when the auth mode is BrowserToken. If no browser token is specified then a new token is generated each time the app host is launched.

Internal

Internal settings are used by the app host and integrations. Internal settings aren't designed to be configured directly.

Expand table

Option	Default value	Description
`AppHost:Directory`	The content root if there's no project.	Directory of the project where the app host is located. Accessible from the IDistributedApplicationBuilder.AppHostDirectory.
`AppHost:Path`	The directory combined with the application name.	The path to the app host. It combines the directory with the application name.
`AppHost:Sha256`	It is created from the app host name when the app host is in publish mode. Otherwise it is created from the app host path.	Hex encoded hash for the current application. The hash is based on the location of the app on the current machine so it is stable between launches of the app host.
`AppHost:OtlpApiKey`	Automatically generated 128-bit entropy token.	The API key used to authenticate requests sent to the dashboard OTLP service. The value is present if needed: the app host is in run mode, the dashboard isn't disabled, and the dashboard isn't configured to allow anonymous access with `DOTNET_DASHBOARD_UNSECURED_ALLOW_ANONYMOUS`.
`AppHost:BrowserToken`	Automatically generated 128-bit entropy token.	The browser token used to authenticate browsing to the dashboard when it is launched by the app host. The browser token can be set by `DOTNET_DASHBOARD_FRONTEND_BROWSERTOKEN`. The value is present if needed: the app host is in run mode, the dashboard isn't disabled, and the

Option	Default value	Description
		dashboard isn't configured to allow anonymous access with `DOTNET_DASHBOARD_UNSECURED_ALLOW_ANONYMOUS`.
`AppHost:ResourceService:AuthMode`	`ApiKey`. If `DOTNET_DASHBOARD_UNSECURED_ALLOW_ANONYMOUS` is true then the value is `Unsecured`.	The authentication mode used to access the resource service. The value is present if needed: the app host is in run mode and the dashboard isn't disabled.
`AppHost:ResourceService:ApiKey`	Automatically generated 128-bit entropy token.	The API key used to authenticate requests made to the app host's resource service. The API key can be set by `DOTNET_DASHBOARD_RESOURCESERVICE_APIKEY`. The value is present if needed: the app host is in run mode, the dashboard isn't disabled, and the dashboard isn't configured to allow anonymous access with `DOTNET_DASHBOARD_UNSECURED_ALLOW_ANONYMOUS`.

Custom resource commands in .NET Aspire

Article • 11/12/2024

Each resource in the .NET Aspire app model is represented as an IResource and when added to the distributed application builder, it's the generic-type parameter of the IResourceBuilder<T> interface. You use the *resource builder* API to chain calls, configuring the underlying resource, and in some situations, you might want to add custom commands to the resource. Some common scenario for creating a custom command might be running database migrations or seeding/resetting a database. In this article, you learn how to add a custom command to a Redis resource that clears the cache.

> **Important**
>
> These .NET Aspire dashboard commands are only available when running the dashboard locally. They're not available when running the dashboard in Azure Container Apps.

Add custom commands to a resource

Start by creating a new .NET Aspire Starter App from the available templates. To create the solution from this template, follow the Quickstart: Build your first .NET Aspire solution. After creating this solution, add a new class named *RedisResourceBuilderExtensions.cs* to the app host project. Replace the contents of the file with the following code:

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Diagnostics.HealthChecks;
using Microsoft.Extensions.Logging;
using StackExchange.Redis;

namespace Aspire.Hosting;

internal static class RedisResourceBuilderExtensions
{
    public static IResourceBuilder<RedisResource> WithClearCommand(
        this IResourceBuilder<RedisResource> builder)
    {
        builder.WithCommand(
```

```
            name: "clear-cache",
            displayName: "Clear Cache",
            executeCommand: context => OnRunClearCacheCommandAsync(builder,
context),
            updateState: OnUpdateResourceState,
            iconName: "AnimalRabbitOff",
            iconVariant: IconVariant.Filled);

        return builder;
    }

    private static async Task<ExecuteCommandResult>
OnRunClearCacheCommandAsync(
        IResourceBuilder<RedisResource> builder,
        ExecuteCommandContext context)
    {
        var connectionString = await
builder.Resource.GetConnectionStringAsync() ??
            throw new InvalidOperationException(
                $"Unable to get the '{context.ResourceName}' connection
string.");

        await using var connection =
ConnectionMultiplexer.Connect(connectionString);

        var database = connection.GetDatabase();

        await database.ExecuteAsync("FLUSHALL");

        return CommandResults.Success();
    }

    private static ResourceCommandState OnUpdateResourceState(
        UpdateCommandStateContext context)
    {
        var logger =
context.ServiceProvider.GetRequiredService<ILogger<Program>>();

        if (logger.IsEnabled(LogLevel.Information))
        {
            logger.LogInformation(
                "Updating resource state: {ResourceSnapshot}",
                context.ResourceSnapshot);
        }

        return context.ResourceSnapshot.HealthStatus is HealthStatus.Healthy
            ? ResourceCommandState.Enabled
            : ResourceCommandState.Disabled;
    }
}
```

The preceding code:

- Shares the Aspire.Hosting namespace so that it's visible to the app host project.

In the preceding example, the `executeCommand` delegate is implemented as an `async` method that clears the cache of the Redis resource. It delegates out to a private class-scoped function named `OnRunClearCacheCommandAsync` to perform the actual cache clearing. Consider the following code:

C#

```
private static async Task<ExecuteCommandResult> OnRunClearCacheCommandAsync(
    IResourceBuilder<RedisResource> builder,
    ExecuteCommandContext context)
{
    var connectionString = await builder.Resource.GetConnectionStringAsync()
??
        throw new InvalidOperationException(
            $"Unable to get the '{context.ResourceName}' connection
string.");

    await using var connection =
ConnectionMultiplexer.Connect(connectionString);

    var database = connection.GetDatabase();

    await database.ExecuteAsync("FLUSHALL");

    return CommandResults.Success();
}
```

The preceding code:

- Retrieves the connection string from the Redis resource.
- Connects to the Redis instance.
- Gets the database instance.
- Executes the `FLUSHALL` command to clear the cache.
- Returns a `CommandResults.Success()` instance to indicate that the command was successful.

Update command state logic

The `updateState` delegate is where the command state is determined. This parameter is defined as a `Func<UpdateCommandStateContext, ResourceCommandState>`. The `UpdateCommandStateContext` provides the following properties:

- `UpdateCommandStateContext.ServiceProvider`: The `IServiceProvider` instance that's used to resolve services.

- `UpdateCommandStateContext.ResourceSnapshot`: The snapshot of the resource instance that the command is being executed on.

The immutable snapshot is an instance of `CustomResourceSnapshot`, which exposes all sorts of valuable details about the resource instance. Consider the following code:

C#

```
private static ResourceCommandState OnUpdateResourceState(
    UpdateCommandStateContext context)
{
    var logger =
context.ServiceProvider.GetRequiredService<ILogger<Program>>();

    if (logger.IsEnabled(LogLevel.Information))
    {
        logger.LogInformation(
            "Updating resource state: {ResourceSnapshot}",
            context.ResourceSnapshot);
    }

    return context.ResourceSnapshot.HealthStatus is HealthStatus.Healthy
        ? ResourceCommandState.Enabled
        : ResourceCommandState.Disabled;
}
```

The preceding code:

- Retrieves the logger instance from the service provider.
- Logs the resource snapshot details.
- Returns `ResourceCommandState.Enabled` if the resource is healthy; otherwise, it returns `ResourceCommandState.Disabled`.

Test the custom command

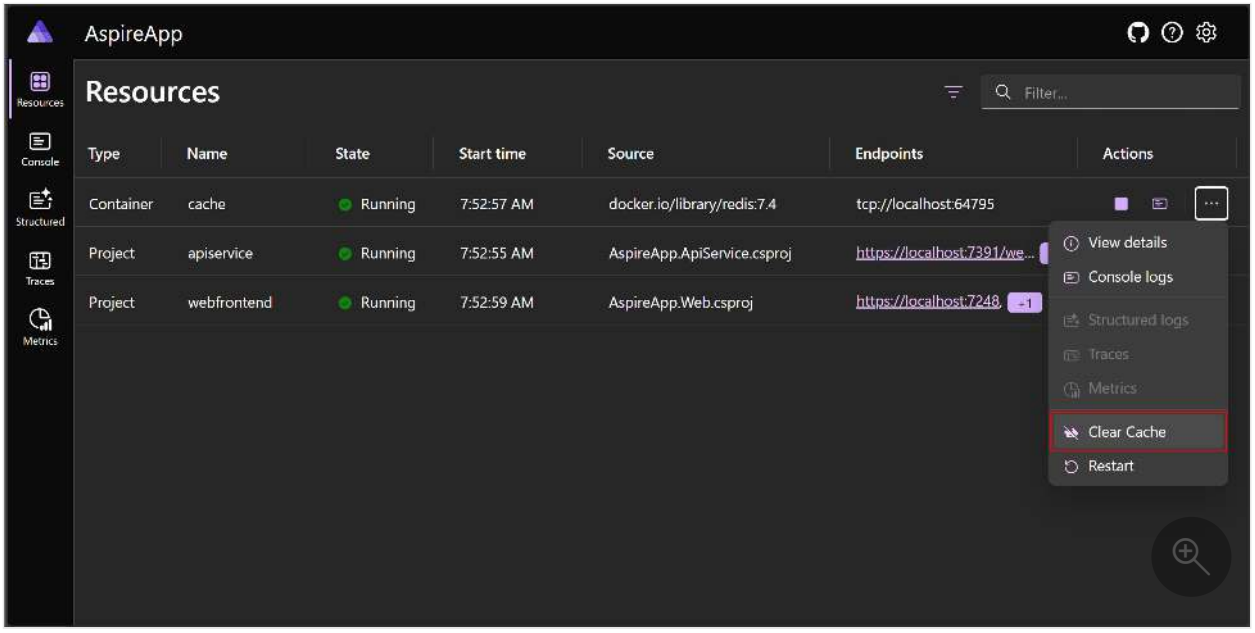
To test the custom command, update your app host project's *Program.cs* file to include the following code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
                   .WithClearCommand();

var apiService = builder.AddProject<Projects.AspireApp_ApiService>
("apiservice");
```

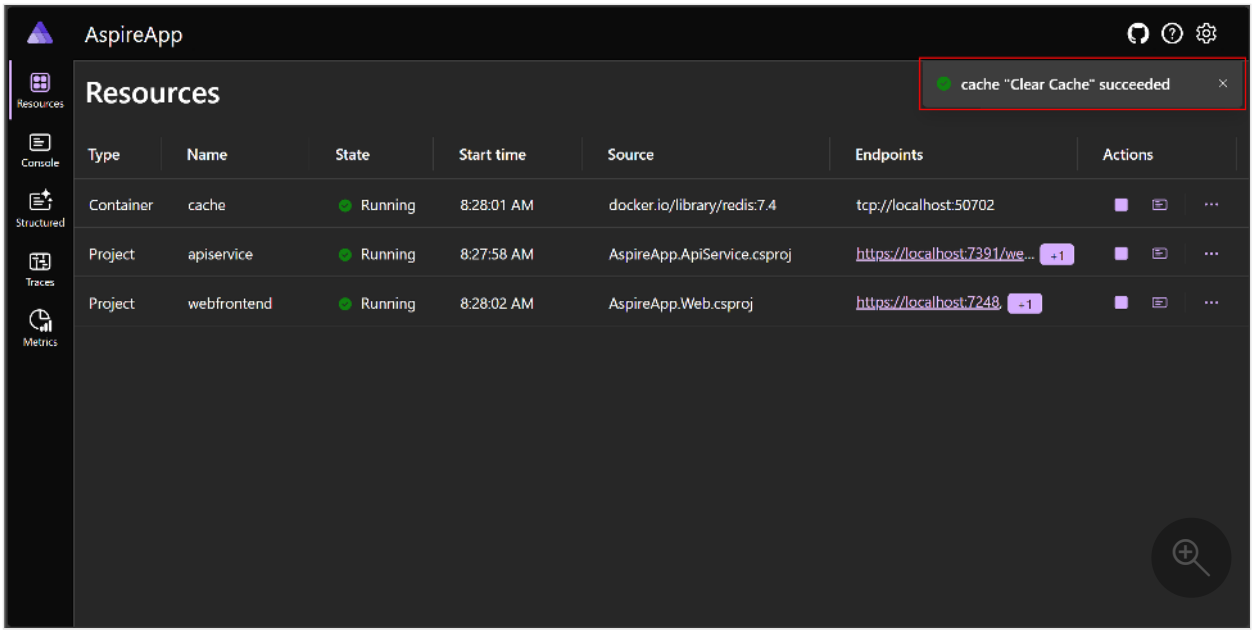
```
builder.AddProject<Projects.AspireApp_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(cache)
    .WaitFor(cache)
    .WithReference(apiService)
    .WaitFor(apiService);

builder.Build().Run();
```

The preceding code calls the `WithClearCommand` extension method to add the custom command to the Redis resource. Run the app and navigate to the .NET Aspire dashboard. You should see the custom command listed under the Redis resource. On the **Resources** page of the dashboard, select the ellipsis button under the **Actions** column:



The preceding image shows the **Clear cache** command that was added to the Redis resource. The icon displays as a rabbit crosses out to indicate that the speed of the dependant resource is being cleared.

Select the **Clear cache** command to clear the cache of the Redis resource. The command should execute successfully, and the cache should be cleared:



See also

- .NET Aspire orchestration overview
- .NET Aspire dashboard: Resource submenu actions

Add Dockerfiles to your .NET app model

Article • 07/23/2024

With .NET Aspire it's possible to specify a *Dockerfile* to build when the app host is started using either the AddDockerfile or WithDockerfile extension methods.

Add a Dockerfile to the app model

In the following example the AddDockerfile extension method is used to specify a container by referencing the context path for the container build.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var container = builder.AddDockerfile(
    "mycontainer", "relative/context/path");
```

Unless the context path argument is a rooted path the context path is interpreted as being relative to the app host projects directory (where the AppHost `*.csproj` folder is located).

By default the name of the *Dockerfile* which is used is `Dockerfile` and is expected to be within the context path directory. It's possible to explicitly specify the *Dockerfile* name either as an absolute path or a relative path to the context path.

This is useful if you wish to modify the specific *Dockerfile* being used when running locally or when the app host is deploying.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var container = builder.ExecutionContext.IsRunMode
    ? builder.AddDockerfile(
          "mycontainer", "relative/context/path", "Dockerfile.debug")
    : builder.AddDockerfile(
          "mycontainer", "relative/context/path", "Dockerfile.release");
```

Customize existing container resources

When using AddDockerfile the return value is an `IResourceBuilder<ContainerResource>`. .NET Aspire includes many custom resource types that are derived from ContainerResource.

Using the WithDockerfile extension method it's possible to continue using these strongly typed resource types and customize the underlying container that is used.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var pgsql = builder.AddPostgres("pgsql")
                   .WithDockerfile("path/to/context")
                   .WithPgAdmin();
```

Pass build arguments

The WithBuildArg method can be used to pass arguments into the container image build.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var container = builder.AddDockerfile("mygoapp", "relative/context/path")
                       .WithBuildArg("GO_VERSION", "1.22");
```

The value parameter on the WithBuildArg method can be a literal value (`boolean`, `string`, `int`) or it can be a resource builder for a parameter resource. The following code replaces the `GO_VERSION` with a parameter value that can be specified at deployment time.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var goVersion = builder.AddParameter("goversion");

var container = builder.AddDockerfile("mygoapp", "relative/context/path")
                       .WithBuildArg("GO_VERSION", goVersion);
```

Build arguments correspond to the ARG command in *Dockerfiles*. Expanding the preceding example, this is a multi-stage *Dockerfile* which specifies specific container image version to use as a parameter.

Dockerfile

```
# Stage 1: Build the Go program
ARG GO_VERSION=1.22
FROM golang:${GO_VERSION} AS builder
WORKDIR /build
COPY . .
RUN go build mygoapp.go

# Stage 2: Run the Go program
FROM mcr.microsoft.com/cbl-mariner/base/core:2.0
WORKDIR /app
COPY --from=builder /build/mygoapp .
CMD ["./mygoapp"]
```

> **Note**
>
> Instead of hardcoding values into the container image, it's recommended to use environment variables for values that frequently change. This avoids the need to rebuild the container image whenever a change is required.

Pass build secrets

In addition to build arguments it's possible to specify build secrets using WithBuildSecret which are made selectively available to individual commands in the *Dockerfile* using the `--mount=type=secret` syntax on `RUN` commands.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var accessToken = builder.AddParameter("accesstoken", secret: true);

var container = builder.AddDockerfile("myapp", "relative/context/path")
                       .WithBuildSecret("ACCESS_TOKEN", accessToken);
```

For example, consider the `RUN` command in a *Dockerfile* which exposes the specified secret to the specific command:

Dockerfile

```
# The helloworld command can read the secret from /run/secrets/ACCESS_TOKEN
RUN --mount=type=secret,id=ACCESS_TOKEN helloworld
```

⊗ Caution

Caution should be exercised when passing secrets in build environments. This is often done when using a token to retrieve dependencies from private repositories or feeds before a build. It is important to ensure that the injected secrets are not copied into the final or intermediate images.

.NET Aspire inner-loop networking overview

Article • 11/12/2024

One of the advantages of developing with .NET Aspire is that it enables you to develop, test, and debug cloud-native apps locally. Inner-loop networking is a key aspect of .NET Aspire that allows your apps to communicate with each other in your development environment. In this article, you learn how .NET Aspire handles various networking scenarios with proxies, endpoints, endpoint configurations, and launch profiles.

Networking in the inner loop

The inner loop is the process of developing and testing your app locally before deploying it to a target environment. .NET Aspire provides several tools and features to simplify and enhance the networking experience in the inner loop, such as:

- **Launch profiles**: Launch profiles are configuration files that specify how to run your app locally. You can use launch profiles (such as the *launchSettings.json* file) to define the endpoints, environment variables, and launch settings for your app.
- **Kestrel configuration**: Kestrel configuration allows you to specify the endpoints that the Kestrel web server listens on. You can configure Kestrel endpoints in your app settings, and .NET Aspire automatically uses these settings to create endpoints.
- **Endpoints/Endpoint configurations**: Endpoints are the connections between your app and the services it depends on, such as databases, message queues, or APIs. Endpoints provide information such as the service name, host port, scheme, and environment variable. You can add endpoints to your app either implicitly (via launch profiles) or explicitly by calling WithEndpoint.
- **Proxies**: .NET Aspire automatically launches a proxy for each service binding you add to your app, and assigns a port for the proxy to listen on. The proxy then forwards the requests to the port that your app listens on, which might be different from the proxy port. This way, you can avoid port conflicts and access your app and services using consistent and predictable URLs.

How endpoints work

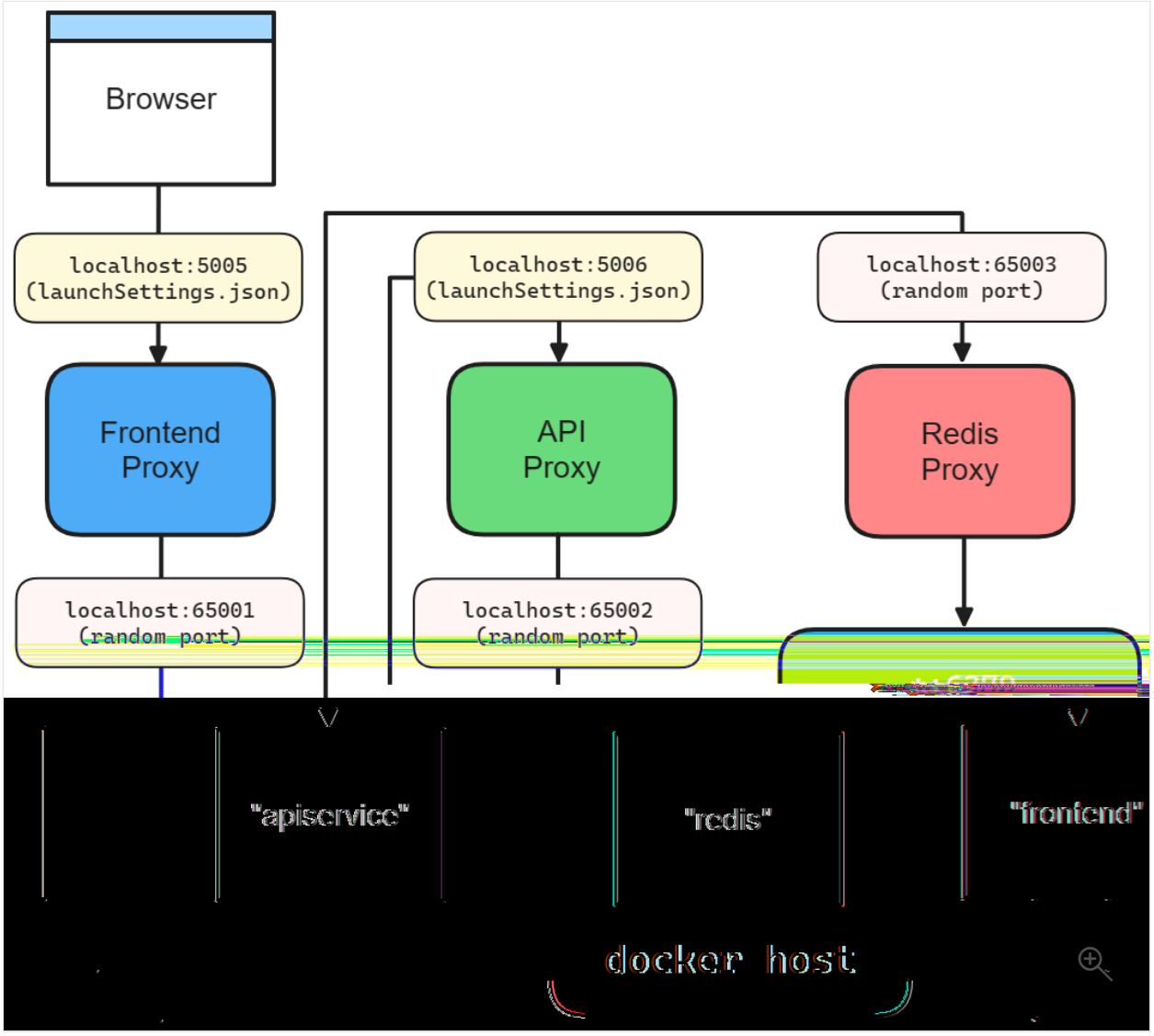
A service binding in .NET Aspire involves two integrations: a **service** representing an external resource your app requires (for example, a database, message queue, or API),

and a **binding** that establishes a connection between your app and the service and provides necessary information.

.NET Aspire supports two service binding types: **implicit**, automatically created based on specified launch profiles defining app behavior in different environments, and **explicit**, manually created using WithEndpoint.

Upon creating a binding, whether implicit or explicit, .NET Aspire launches a lightweight reverse proxy on a specified port, handling routing and load balancing for requests from your app to the service. The proxy is a .NET Aspire implementation detail, requiring no configuration or management concern.

To help visualize how endpoints work, consider the .NET Aspire starter templates inner-loop networking diagram:



Launch profiles

When you call AddProject, the app host looks for *Properties/launchSettings.json* to determine the default set of endpoints. The app host selects a specific launch profile

using the following rules:

1. An explicit `launchProfileName` argument passed when calling `AddProject`.
2. The `DOTNET_LAUNCH_PROFILE` environment variable. For more information, see .NET environment variables.
3. The first launch profile defined in *launchSettings.json*.

Consider the following *launchSettings.json* file:

JSON

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": false,
      "inspectUri": "{wsProtocol}://{url.hostname}:
{url.port}/_framework/debug/ws-proxy?browser={browserInspectUri}",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "inspectUri": "{wsProtocol}://{url.hostname}:
{url.port}/_framework/debug/ws-proxy?browser={browserInspectUri}",
      "applicationUrl": "https://localhost:7239;http://localhost:5066",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

For the remainder of this article, imagine that you've created an IDistributedApplicationBuilder assigned to a variable named `builder` with the CreateBuilder() API:

C#

```
var builder = DistributedApplication.CreateBuilder(args);
```

To specify the **http** and **https** launch profiles, configure the `applicationUrl` values for both in the *launchSettings.json* file. These URLs are used to create endpoints for this

project. This is the equivalent of:

C#

```
builder.AddProject<Projects.Networking_Frontend>("frontend")
       .WithHttpEndpoint(port: 5066)
       .WithHttpsEndpoint(port: 7239);
```

> **ⓘ Important**
>
> If there's no *launchSettings.json* (or launch profile), there are no bindings by default.

For more information, see .NET Aspire and launch profiles.

Kestrel configured endpoints

.NET Aspire supports Kestrel endpoint configuration. For example, consider an *appsettings.json* file for a project that defines a Kestrel endpoint with the HTTPS scheme and port 5271:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "Kestrel": {
    "Endpoints": {
      "Https": {
        "Url": "https://*:5271"
      }
    }
  }
}
```

The preceding configuration specifies an `Https` endpoint. The `Url` property is set to `https://*:5271`, which means the endpoint listens on all interfaces on port 5271. For more information, see Configure endpoints for the ASP.NET Core Kestrel web server.

With the Kestrel endpoint configured, the project should remove any configured `applicationUrl` from the *launchSettings.json* file.

> **Note**
>
> If the `applicationUrl` is present in the *launchSettings.json* file and the Kestrel endpoint is configured, the app host will throw an exception.

When you add a project resource, there's an overload that lets you specify that the Kestrel endpoint should be used instead of the *launchSettings.json* file:

C#

```
builder.AddProject<Projects.Networking_ApiService>(
    name: "apiservice",
    configure: static project =>
    {
        project.ExcludeLaunchProfile = true;
        project.ExcludeKestrelEndpoints = false;
    })
    .WithHttpsEndpoint();
```

For more information, see AddProject.

Ports and proxies

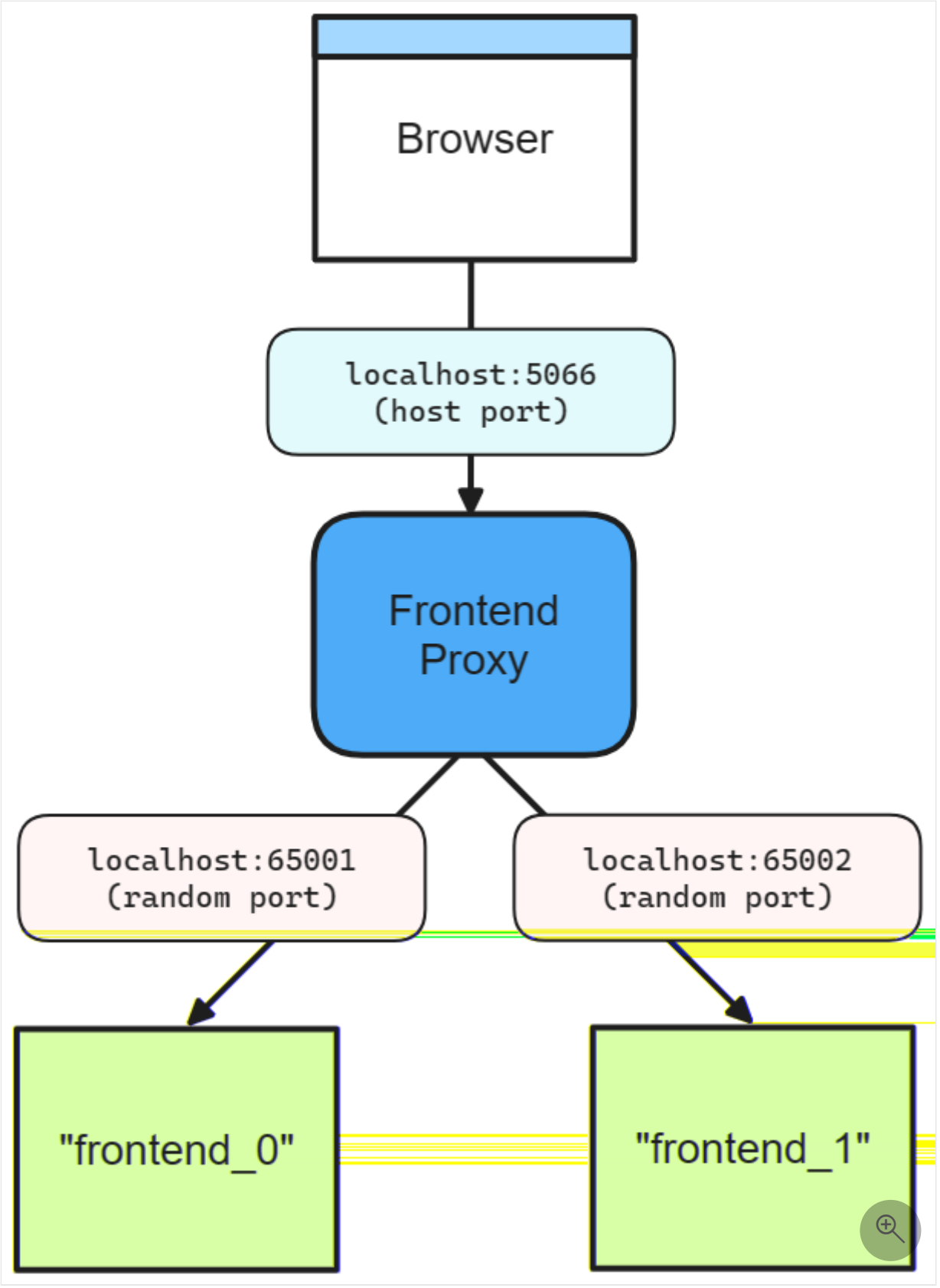
When defining a service binding, the host port is *always* given to the proxy that sits in front of the service. This allows single or multiple replicas of a service to behave similarly. Additionally, all resource dependencies that use the WithReference API rely of the proxy endpoint from the environment variable.

Consider the following method chain that calls AddProject, WithHttpEndpoint, and then WithReplicas:

C#

```
builder.AddProject<Projects.Networking_Frontend>("frontend")
       .WithHttpEndpoint(port: 5066)
       .WithReplicas(2);
```

The preceding code results in the following networking diagram:



The preceding diagram depicts the following:

- A web browser as an entry point to the app.
- A host port of 5066.
- The frontend proxy sitting between the web browser and the frontend service replicas, listening on port 5066.

- The `frontend_0` frontend service replica listening on the randomly assigned port 65001.
- The `frontend_1` frontend service replica listening on the randomly assigned port 65002.

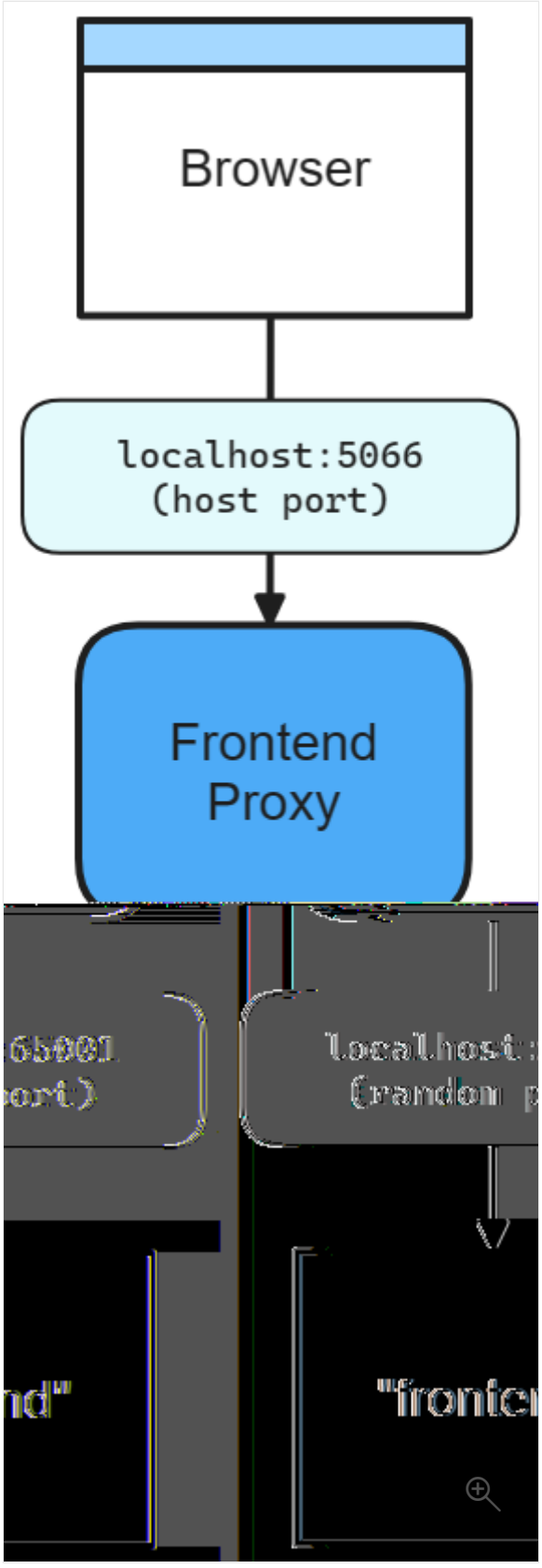
Without the call to `WithReplicas`, there's only one frontend service. The proxy still listens on port 5066, but the frontend service listens on a random port:

C#

```
builder.AddProject<Projects.Networking_Frontend>("frontend")
       .WithHttpEndpoint(port: 5066);
```

There are two ports defined:

- A host port of 5066.
- A random proxy port that the underlying service will be bound to.



The preceding diagram depicts the following:

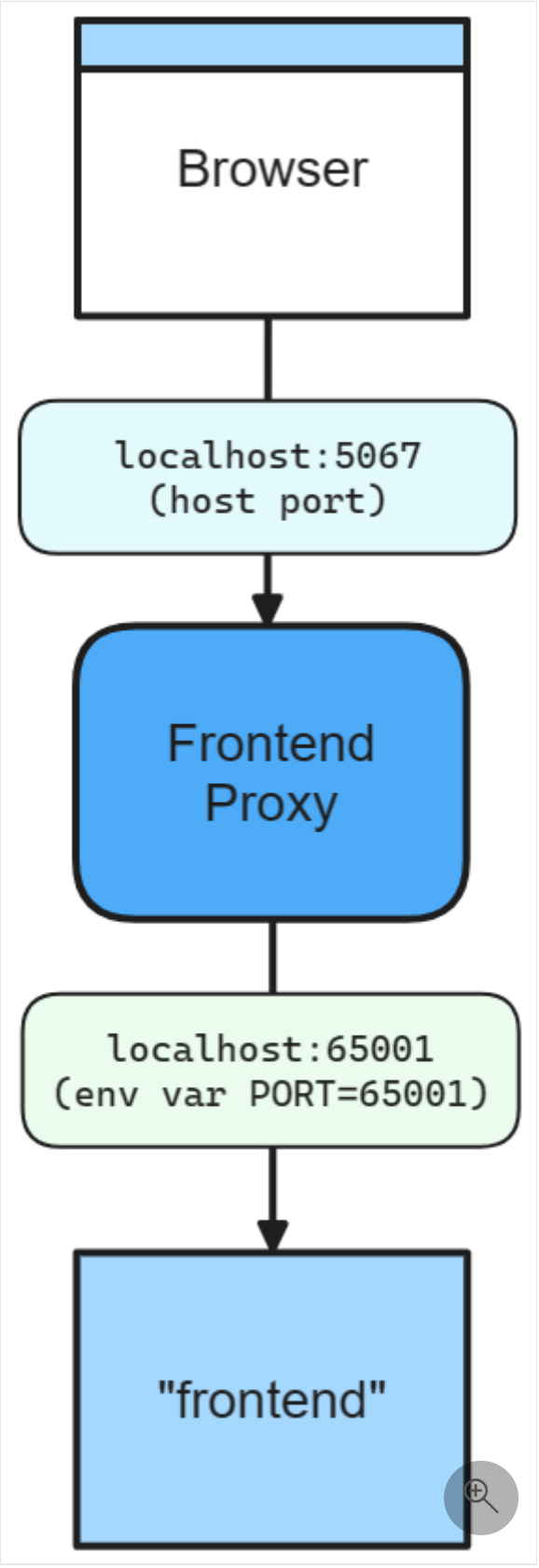
- A web browser as an entry point to the app.
- A host port of 5066.
- The frontend proxy sitting between the web browser and the frontend service, listening on port 5066.
- The frontend service listening on random port of 65001.

The underlying service is fed this port via `ASPNETCORE_URLS` for project resources. Other resources access to this port by specifying an environment variable on the service binding:

C#

```
builder.AddNpmApp("frontend", "../NodeFrontend", "watch")
       .WithHttpEndpoint(port: 5067, env: "PORT");
```

The previous code makes the random port available in the `PORT` environment variable. The app uses this port to listen to incoming connections from the proxy. Consider the following diagram:



The preceding diagram depicts the following:

- A web browser as an entry point to the app.
- A host port of 5067.
- The frontend proxy sitting between the web browser and the frontend service, listening on port 5067.
- The frontend service listening on an environment 65001.

> **Tip**
>
> To avoid an endpoint being proxied, set the `IsProxied` property to `false` when calling the `WithEndpoint` extension method. For more information, see **Endpoint extensions: additional considerations**.

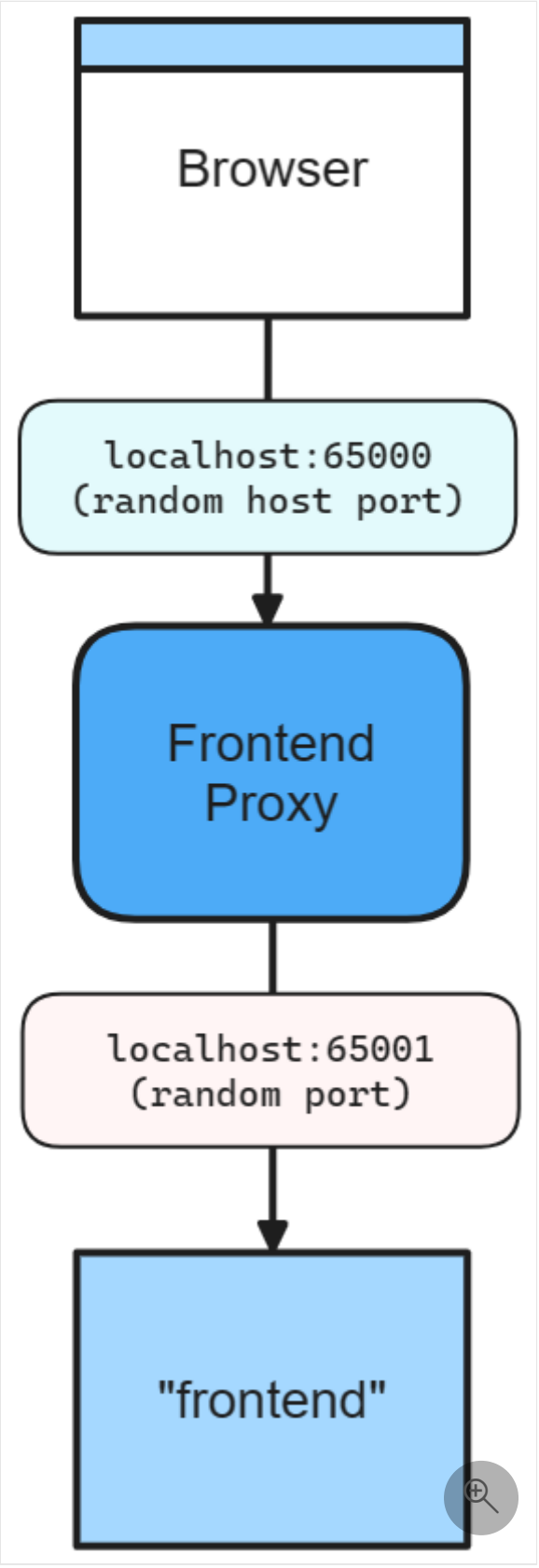
Omit the host port

When you omit the host port, .NET Aspire generates a random port for both host and service port. This is useful when you want to avoid port conflicts and don't care about the host or service port. Consider the following code:

C#

```
builder.AddProject<Projects.Networking_Frontend>("frontend")
       .WithHttpEndpoint();
```

In this scenario, both the host and service ports are random, as shown in the following diagram:



The preceding diagram depicts the following:

- A web browser as an entry point to the app.
- A random host port of 65000.
- The frontend proxy sitting between the web browser and the frontend service, listening on port 65000.
- The frontend service listening on a random port of 65001.

Container ports

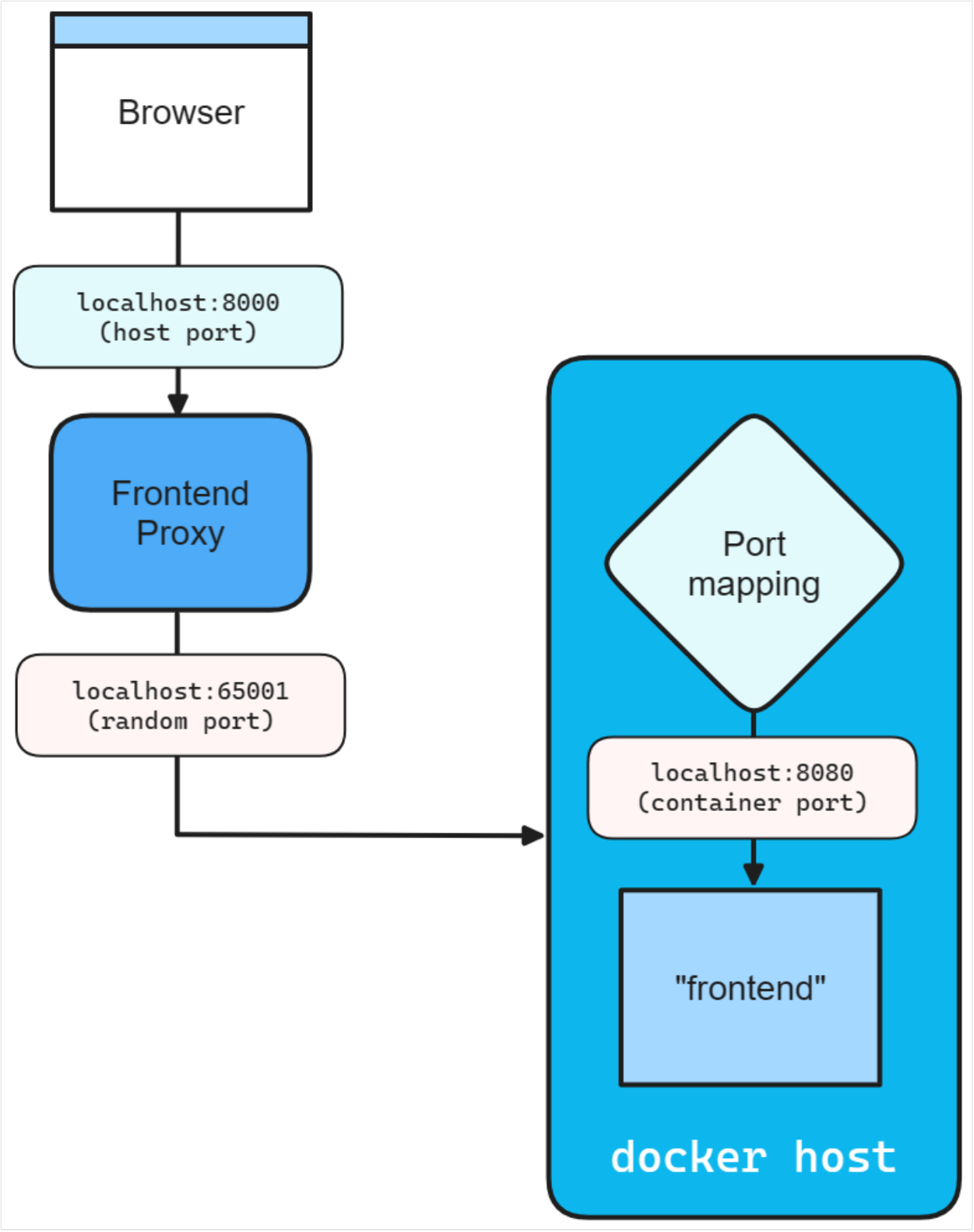
When you add a container resource, .NET Aspire automatically assigns a random port to the container. To specify a container port, configure the container resource with the desired port:

```C#
builder.AddContainer("frontend", "mcr.microsoft.com/dotnet/samples",
"aspnetapp")
       .WithHttpEndpoint(port: 8000, targetPort: 8080);
```

The preceding code:

- Creates a container resource named `frontend`, from the `mcr.microsoft.com/dotnet/samples:aspnetapp` image.
- Exposes an `http` endpoint by binding the host to port 8000 and mapping it to the container's port 8080.

Consider the following diagram:



Endpoint extension methods

Any resource that implements the IResourceWithEndpoints interface can use the `WithEndpoint` extension methods. There are several overloads of this extension, allowing you to specify the scheme, container port, host port, environment variable name, and whether the endpoint is proxied.

There's also an overload that allows you to specify a delegate to configure the endpoint. This is useful when you need to configure the endpoint based on the environment or other factors. Consider the following code:

C#

```
builder.AddProject<Projects.Networking_ApiService>("apiService")
       .WithEndpoint(
            endpointName: "admin",
            callback: static endpoint =>
       {
           endpoint.Port = 17003;
           endpoint.UriScheme = "http";
           endpoint.Transport = "http";
       });
```

The preceding code provides a callback delegate to configure the endpoint. The endpoint is named `admin` and configured to use the `http` scheme and transport, as well as the 17003 host port. The consumer references this endpoint by name, consider the following `AddHttpClient` call:

C#

```
builder.Services.AddHttpClient<WeatherApiClient>(
    client => client.BaseAddress = new Uri("http://_admin.apiservice"));
```

The `Uri` is constructed using the `admin` endpoint name prefixed with the `_` sentinel. This is a convention to indicate that the `admin` segment is the endpoint name belonging to the `apiservice` service. For more information, see .NET Aspire service discovery.

Additional considerations

When calling the `WithEndpoint` extension method, the `callback` overload exposes the raw EndpointAnnotation, which allows the consumer to customize many aspects of the endpoint.

The `AllocatedEndpoint` property allows you to get or set the endpoint for a service. The `IsExternal` and `IsProxied` properties determine how the endpoint is managed and exposed: `IsExternal` decides if it should be publicly accessible, while `IsProxied` ensures DCP manages it, allowing for internal port differences and replication.

> **Tip**

If you're hosting an external executable that runs its own proxy and encounters port binding issues due to DCP already binding the port, try setting the `IsProxied` property to `false`. This prevents DCP from managing the proxy, allowing your executable to bind the port successfully.

The `Name` property identifies the service, whereas the `Port` and `TargetPort` properties specify the desired and listening ports, respectively.

For network communication, the `Protocol` property supports **TCP** and **UDP**, with potential for more in the future, and the `Transport` property indicates the transport protocol (**HTTP**, **HTTP2**, **HTTP3**). Lastly, if the service is URI-addressable, the `UriScheme` property provides the URI scheme for constructing the service URI.

For more information, see the available properties of the EndpointAnnotation properties.

Endpoint filtering

All .NET Aspire project resource endpoints follow a set of default heuristics. Some endpoints are included in `ASPNETCORE_URLS` at runtime, some are published as `HTTP/HTTPS_PORTS`, and some configurations are resolved from Kestrel configuration. Regardless of the default behavior, you can filter the endpoints that are included in environment variables by using the WithEndpointsInEnvironment extension method:

C#

```
builder.AddProject<Projects.Networking_ApiService>("apiservice")
    .WithHttpsEndpoint() // Adds a default "https" endpoint
    .WithHttpsEndpoint(port: 19227, name: "admin")
    .WithEndpointsInEnvironment(
        filter: static endpoint =>
        {
            return endpoint.Name is not "admin";
        });
```

The preceding code adds a default HTTPS endpoint, as well as an `admin` endpoint on port 19227. However, the `admin` endpoint is excluded from the environment variables. This is useful when you want to expose an endpoint for internal use only.

Eventing in .NET Aspire

Article • 11/13/2024

In .NET Aspire, eventing allows you to publish and subscribe to events during various app host life cycles. Eventing is more flexible than life cycle events. Both let you run arbitrary code during event callbacks, but eventing offers finer control of event timing, publishing, and provides supports for custom events.

The eventing mechanisms in .NET Aspire are part of the Aspire.Hosting NuGet package. This package provides a set of interfaces and classes in the Aspire.Hosting.Eventing namespace that you use to publish and subscribe to events in your .NET Aspire app host project. Eventing is scoped to the app host itself and the resources within.

In this article, you learn how to use the eventing features in .NET Aspire.

App host eventing

The following events are available in the app host and occur in the following order:

1. BeforeStartEvent: This event is raised before the app host starts.
2. AfterEndpointsAllocatedEvent: This event is raised after the app host allocated endpoints.
3. AfterResourcesCreatedEvent: This event is raised after the app host created resources.

All of the preceding events are analogous to the app host life cycles. That is, an implementation of the IDistributedApplicationLifecycleHook could handle these events just the same. With the eventing API, however, you can run arbitrary code when these events are raised and event define custom events—any event that implements the IDistributedApplicationEvent interface.

Subscribe to app host events

To subscribe to the built-in app host events, use the eventing API. After you have a distributed application builder instance, walk up to the IDistributedApplicationBuilder.Eventing property and call the Subscribe<T>(Func<T,CancellationToken,Task>) API. Consider the following sample app host *Program.cs* file:

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache");

var apiService = builder.AddProject<Projects.AspireApp_ApiService>
("apiservice");

builder.AddProject<Projects.AspireApp_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(cache)
    .WaitFor(cache)
    .WithReference(apiService)
    .WaitFor(apiService);

builder.Eventing.Subscribe<BeforeStartEvent>(
    static (@event, cancellationToken) =>
    {
        var logger = @event.Services.GetRequiredService<ILogger<Program>>();

        logger.LogInformation("1. BeforeStartEvent");

        return Task.CompletedTask;
    });

builder.Eventing.Subscribe<AfterEndpointsAllocatedEvent>(
    static (@event, cancellationToken) =>
    {
        var logger = @event.Services.GetRequiredService<ILogger<Program>>();

        logger.LogInformation("2. AfterEndpointsAllocatedEvent");

        return Task.CompletedTask;
    });

builder.Eventing.Subscribe<AfterResourcesCreatedEvent>(
    static (@event, cancellationToken) =>
    {
        var logger = @event.Services.GetRequiredService<ILogger<Program>>();

        logger.LogInformation("3. AfterResourcesCreatedEvent");

        return Task.CompletedTask;
    });

builder.Build().Run();
```

The preceding code is based on the starter template with the addition of the calls to the `Subscribe` API. The `Subscribe<T>` API returns a DistributedApplicationEventSubscription instance that you can use to unsubscribe from the event. It's common to discard the

returned subscriptions, as you don't usually need to unsubscribe from events as the entire app is torn down when the app host is shut down.

When the app host is run, by the time the .NET Aspire dashboard is displayed, you should see the following log output in the console:

Plaintext

```
info: Program[0]
      1. BeforeStartEvent
info: Aspire.Hosting.DistributedApplication[0]
      Aspire version: 9.0.0
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application starting.
info: Aspire.Hosting.DistributedApplication[0]
      Application host directory is: ..\AspireApp\AspireApp.AppHost
info: Program[0]
      2. AfterEndpointsAllocatedEvent
info: Aspire.Hosting.DistributedApplication[0]
      Now listening on: https://localhost:17178
info: Aspire.Hosting.DistributedApplication[0]
      Login to the dashboard at https://localhost:17178/login?t=<YOUR_TOKEN>
info: Program[0]
      3. AfterResourcesCreatedEvent
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application started. Press Ctrl+C to shut down.
```

The log output confirms that event handlers are executed in the order of the app host life cycle events. The subscription order doesn't affect execution order. The `BeforeStartEvent` is triggered first, followed by `AfterEndpointsAllocatedEvent`, and finally `AfterResourcesCreatedEvent`.

Resource eventing

In addition to the app host events, you can also subscribe to resource events. Resource events are raised specific to an individual resource. Resource events are defined as implementations of the IDistributedApplicationResourceEvent interface. The following resource events are available in the listed order:

1. ConnectionStringAvailableEvent: Raised when a connection string becomes available for a resource.
2. BeforeResourceStartedEvent: Raised before the orchestrator starts a new resource.
3. ResourceReadyEvent: Raised when a resource initially transitions to a ready state.

Subscribe to resource events

To subscribe to resource events, use the eventing API. After you have a distributed application builder instance, walk up to the IDistributedApplicationBuilder.Eventing property and call the Subscribe<T>(IResource, Func<T,CancellationToken,Task>) API. Consider the following sample app host *Program.cs* file:

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache");

builder.Eventing.Subscribe<ResourceReadyEvent>(
    cache.Resource,
    static (@event, cancellationToken) =>
    {
        var logger = @event.Services.GetRequiredService<ILogger<Program>>();

        logger.LogInformation("3. ResourceReadyEvent");

        return Task.CompletedTask;
    });

builder.Eventing.Subscribe<BeforeResourceStartedEvent>(
    cache.Resource,
    static (@event, cancellationToken) =>
    {
        var logger = @event.Services.GetRequiredService<ILogger<Program>>();

        logger.LogInformation("2. BeforeResourceStartedEvent");

        return Task.CompletedTask;
    });

builder.Eventing.Subscribe<ConnectionStringAvailableEvent>(
    cache.Resource,
    static (@event, cancellationToken) =>
    {
        var logger = @event.Services.GetRequiredService<ILogger<Program>>();

        logger.LogInformation("1. ConnectionStringAvailableEvent");

        return Task.CompletedTask;
    });

var apiService = builder.AddProject<Projects.AspireApp_ApiService>
("apiservice");

builder.AddProject<Projects.AspireApp_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(cache)
```

```
    .WaitFor(cache)
    .WithReference(apiService)
    .WaitFor(apiService);

builder.Build().Run();
```

The preceding code subscribes to the `ResourceReadyEvent`, `ConnectionStringAvailableEvent`, and `BeforeResourceStartedEvent` events on the `cache` resource. When AddRedis is called, it returns an IResourceBuilder<T> where `T` is a RedisResource. The resource builder exposes the resource as the IResourceBuilder<T>.Resource property. The resource in question is then passed to the `Subscribe` API to subscribe to the events on the resource.

When the app host is run, by the time the .NET Aspire dashboard is displayed, you should see the following log output in the console:

Plaintext

```
info: Aspire.Hosting.DistributedApplication[0]
      Aspire version: 9.0.0
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application starting.
info: Aspire.Hosting.DistributedApplication[0]
      Application host directory is: ..\AspireApp\AspireApp.AppHost
info: Program[0]
      1. ConnectionStringAvailableEvent
info: Program[0]
      2. BeforeResourceStartedEvent
info: Program[0]
      3. ResourceReadyEvent
info: Aspire.Hosting.DistributedApplication[0]
      Now listening on: https://localhost:17222
info: Aspire.Hosting.DistributedApplication[0]
      Login to the dashboard at https://localhost:17222/login?t=<YOUR_TOKEN>
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application started. Press Ctrl+C to shut down.
```

> **Note**
>
> Some events are blocking. For example, when the `BeforeResourceStartEvent` is published, the startup of the resource will be blocked until all subscriptions for that event on a given resource have completed executing. Whether an event is blocking or not depends on how it is published (see the following section).

Publish events

When subscribing to any of the built-in events, you don't need to publish the event yourself as the app host orchestrator manages to publish built-in events on your behalf. However, you can publish custom events with the eventing API. To publish an event, you have to first define an event as an implementation of either the IDistributedApplicationEvent or IDistributedApplicationResourceEvent interface. You need to determine which interface to implement based on whether the event is an app host event or a resource-specific event.

Then, you can subscribe and publish the event by calling the either

- PublishAsync<T>(T, CancellationToken): Publishes an e
  specific event type.
- PublishAsync<T>(T, EventDispatchBehavior, Ca
  to all subscribes of the specific event type w

When events are dispatched, yo
subscribers. The event dispat
enum. The following be

- tchBeha
  ey're all processe

External parameters

Article • 12/06/2024

Environments provide context for the application to run in. Parameters express the ability to ask for an external value when running the app. Parameters can be used to provide values to the app when running locally, or to prompt for values when deploying. They can be used to model a wide range of scenarios including secrets, connection strings, and other configuration values that might vary between environments.

Parameter values

Parameter values are read from the `Parameters` section of the app host's configuration and are used to provide values to the app while running locally. When you publish the app, if the value isn't configured you're prompted to provide it.

Consider the following example app host *Program.cs* file:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

// Add a parameter named "value"
var value = builder.AddParameter("value");

builder.AddProject<Projects.ApiService>("api")
       .WithEnvironment("EXAMPLE_VALUE", value);
```

The preceding code adds a parameter named `value` to the app host. The parameter is then passed to the `Projects.ApiService` project as an environment variable named `EXAMPLE_VALUE`.

Configure parameter values

Adding parameters to the builder is only one aspect of the configuration. You must also provide the value for the parameter. The value can be provided in the app host configuration file, set as a user secret, or configured in any other standard configuration. When parameter values aren't found, they're prompted for when publishing the app.

Consider the following app host configuration file *appsettings.json*:

JSON

```
{
    "Parameters": {
        "value": "local-value"
    }
}
```

The preceding JSON configures a parameter in the `Parameters` section of the app host configuration. In other words, that app host is able to find the parameter as its configured. For example, you could walk up to the IDistributedApplicationBuilder.Configuration and access the value using the `Parameters:value` key:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var key = $"Parameters:value";
var value = builder.Configuration[key]; // value = "local-value"
```

> **ⓘ Important**
>
> However, you don't need to access this configuration value yourself in the app host. Instead, the ParameterResource is used to pass the parameter value to dependent resources. Most often as an environment variable.

Parameter representation in the manifest

.NET Aspire uses a deployment manifest to represent the app's resources and their relationships. Parameters are represented in the manifest as a new primitive called `parameter.v0`:

JSON

```
{
  "resources": {
    "value": {
      "type": "parameter.v0",
      "value": "{value.inputs.value}",
      "inputs": {
        "value": {
          "type": "string"
        }
      }
    }
```

```
  }
}
```

Secret values

Parameters can be used to model secrets. When a parameter is marked as a secret, it serves as a hint to the manifest that the value should be treated as a secret. When you publish the app, the value is prompted for and stored in a secure location. When you run the app locally, the value is read from the `Parameters` section of the app host configuration.

Consider the following example app host *Program.cs* file:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

// Add a secret parameter named "secret"
var secret = builder.AddParameter("secret", secret: true);

builder.AddProject<Projects.ApiService>("api")
       .WithEnvironment("SECRET", secret);

builder.Build().Run();
```

Now consider the following app host configuration file *appsettings.json*:

JSON

```
{
    "Parameters": {
        "secret": "local-secret"
    }
}
```

The manifest representation is as follows:

JSON

```
{
  "resources": {
    "value": {
      "type": "parameter.v0",
      "value": "{value.inputs.value}",
      "inputs": {
        "value": {
          "type": "string",
```

```
          "secret": true
        }
      }
    }
  }
}
```

Connection string values

Parameters can be used to model connection strings. When you publish the app, the value is prompted for and stored in a secure location. When you run the app locally, the value is read from the `ConnectionStrings` section of the app host configuration.

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

Consider the following example app host *Program.cs* file:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var redis = builder.AddConnectionString("redis");

builder.AddProject<Projects.WebApplication>("api")
       .WithReference(redis);

builder.Build().Run();
```

Now consider the following app host configuration file *appsettings.json*:

JSON

```
{
    "ConnectionStrings": {
        "redis": "local-connection-string"
    }
}
```

For more information pertaining to connection strings and their representation in the deployment manifest, see Connection string and binding references.

Parameter example

To express a parameter, consider the following example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var db = builder.AddSqlServer("sql")
                .PublishAsConnectionString()
                .AddDatabase("db");

var insertionRows = builder.AddParameter("insertionRows");

builder.AddProject<Projects.Parameters_ApiService>("api")
       .WithEnvironment("InsertionRows", insertionRows)
       .WithReference(db);

builder.Build().Run();
```

The following steps are performed:

- Adds a SQL Server resource named `sql` and publishes it as a connection string.
- Adds a database named `db`.
- Adds a parameter named `insertionRows`.
- Adds a project named `api` and associates it with the `Projects.Parameters_ApiService` project resource type-parameter.
- Passes the `insertionRows` parameter to the `api` project.
- References the `db` database.

The value for the `insertionRows` parameter is read from the `Parameters` section of the app host configuration file *appsettings.json*:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Aspire.Hosting.Dcp": "Warning"
    }
  },
  "Parameters": {
```

```
    "insertionRows": "1"
  }
}
```

The `Parameters_ApiService` project consumes the `insertionRows` parameter. Consider the *Program.cs* example file:

C#

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

int insertionRows = builder.Configuration.GetValue<int>("InsertionRows", 1);

builder.AddServiceDefaults();

builder.AddSqlServerDbContext<MyDbContext>("db");

var app = builder.Build();

app.MapGet("/", async (MyDbContext context) =>
{
    // You wouldn't normally do this on every call,
    // but doing it here just to make this simple.
    context.Database.EnsureCreated();

    for (var i = 0; i < insertionRows; i++)
    {
        var entry = new Entry();
        await context.Entries.AddAsync(entry);
    }

    await context.SaveChangesAsync();

    var entries = await context.Entries.ToListAsync();

    return new
    {
        totalEntries = entries.Count,
        entries
    };
});

app.Run();
```

See also

- .NET Aspire manifest format for deployment tool builders

- Tutorial: Connect an ASP.NET Core app to SQL Server using .NET Aspire and Entity Framework Core

Persist .NET Aspire project data using volumes or bind mounts

Article • 04/02/2025

Every time you start and stop a .NET Aspire project, the app also creates and destroys the app resource containers. Any data or files stored in those containers during a debugging session is lost for subsequent sessions. Many development teams prefer to keep this data across debugging sessions so that, for example, they don't have to repopulate a database with sample data for each run.

In this article, you learn how to configure .NET Aspire projects to persist data across app launches. A continuous set of data during local development is useful in many scenarios. Various .NET Aspire resource container types are able to leverage volumes and bind mounts, such as PostgreSQL, Redis and Azure Storage.

When to persist project data

Suppose you have a .NET Aspire solution with a database resource. By default, data is saved in the container for that resource. Because all the resource containers are destroyed when you stop your app, you lose that data and won't see it the next time you run the solution. This setup creates problems when you want to persist data in a database or storage services between app launches for testing or debugging. For example, you may want to:

- Work with a continuous set of data in a database during an extended development session across multiple restarts.
- Test or debug a changing set of files in an Azure Blob Storage emulator.
- Maintain cached data or messages in a Redis instance across app launches.

You can accomplish these goals using volumes or bind mounts. These objects store data outside the container in a directory on the container host, so it's not destroyed with the container. This way, you decide which services retain data between launches of your .NET Aspire project.

> ⓘ **Note**
>
> Volumes and bind mounts are features of your container runtime: Docker or Podman. .NET Aspire includes methods that make it easy to work with those features.

Compare volumes and bind mounts

Both volumes and bind mounts store data in a directory on the container host. Because this directory is outside the container, data isn't destroyed when the container stops. Volumes and bind mounts, however behave differently:

- **Volumes**: The container runtime creates and controls volumes. Volumes are isolated from the core functionality of the container host.
- **Bind mounts**: The container runtime mounts a file or directory on the host machine. Both the container and the host machine can access the contents of the bind mount.

Volumes are more secure and portable than bind mounts. They also perform better and you should use them wherever possible. Use bind mounts only if you need to access or modify the data from your host machine.

Use volumes

Volumes are the recommended way to persist data generated by containers and they're supported on both Windows and Linux. Volumes can store data from multiple containers at a time, offer high performance, and are easy to back up or migrate. With .NET Aspire, you configure a volume for each resource container using the ContainerResourceBuilderExtensions.WithVolume method, which accepts three parameters:

- `name`: An optional name for the volume.
- `target`: The target path in the container of the data you want to persist.
- `isReadOnly`: A Boolean flag that indicates whether the data in the volume can be changed. The default value is `false`.

For the remainder of this article, imagine that you're exploring a `Program` class in a .NET Aspire app host project that's already defined the distributed app builder bits:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

// TODO:
//   Consider various code snippets for configuring
//   volumes here and persistent passwords.

builder.Build().Run();
```

The first code snippet to consider uses the ContainerResourceBuilderExtensions.WithVolume API to configure a volume for a SQL Server resource. The following code demonstrates how to configure a volume for a SQL Server resource in a .NET Aspire app host project:

```
C#

var sql = builder.AddSqlServer("sql")
                 .WithVolume(target: "/var/opt/mssql")
                 .AddDatabase("sqldb");
```

In this example `/var/opt/mssql` sets the path to the database files in the container.

All .NET Aspire container resources can utilize volumes, and some provide convenient APIs for adding named volumes derived from resources. Using the WithDataVolume method as an example, the following code is functionally equivalent to the previous example but more succinct:

```
C#

var sql = builder.AddSqlServer("sql")
                 .WithDataVolume()
                 .AddDatabase("sqldb");
```

With the app host project being named `VolumeMount.AppHost`, the `WithDataVolume` method automatically creates a named volume as `VolumeMount.AppHost-sql-data` and is mounted to the `/var/opt/mssql` path in the SQL Server container. The naming convention is as follows:

- `{appHostProjectName}-{resourceName}-data`: The volume name is derived from the app host project name and the resource name.

Use bind mounts

Bind mounts enable access to the data from both within the container and from processes on the host machine. For example, once a bind mount is established, you can copy a file into it on your host computer. The file is then available at the bound path within the container for your resource. With .NET Aspire, you configure a bind mount for each resource container using the WithBindMount method, which accepts three parameters:

- **`source`**: The path to the folder on the host machine to mount in the container.
- **`target`**: The target path in the container for the folder.

- `isReadOnly`: A Boolean flag that indicates whether the data in the bind mount can be changed. The default value is `false`.

Consider this code snippet, which uses the WithBindMount API to configure a bind mount for a SQL Server resource:

```
C#

var sql = builder.AddSqlServer("sql")
                 .WithBindMount(source: @"C:\SqlServer\Data", target:
"/var/opt/mssql")
                 .AddDatabase("sqldb");
```

In this example:

- `source: @"C:\SqlServer\Data"` sets the folder on the host computer that will be bound.
- `target: "/var/opt/mssql"` sets the path to the database files in the container.

As for volumes, some .NET Aspire container resources provide convenient APIs for adding bind mounts. Using the WithDataBindMount method as an example, the following code is functionally equivalent to the previous example but more succinct:

```
C#

var sql = builder.AddSqlServer("sql")
                 .WithDataBindMount(source: @"C:\SqlServer\Data")
                 .AddDatabase("sqldb");
```

Create persistent passwords

Named volumes require a consistent password between app launches. .NET Aspire conveniently provides random password generation functionality. Consider the previous example once more, where a password is generated automatically:

```
C#

var sql = builder.AddSqlServer("sql")
                 .WithDataVolume()
                 .AddDatabase("sqldb");
```

Since the `password` parameter isn't provided when calling `AddSqlServer`, .NET Aspire automatically generates a password for the SQL Server resource.

> **Important**
>
> This isn't a persistent password! Instead, it changes every time the app host runs.

To create a *persistent* password, you must override the generated password. To do this, run the following command in your app host project directory to set a local password in your .NET user secrets:

.NET CLI

```
dotnet user-secrets set Parameters:sql-password <password>
```

The naming convention for these secrets is important to understand. The password is stored in configuration with the `Parameters:sql-password` key. The naming convention follows this pattern:

- `Parameters:{resourceName}-password`: In the case of the SQL Server resource (which was named `"sql"`), the password is stored in the configuration with the key `Parameters:sql-password`.

The same pattern applies to the other server-based resource types, such as those shown in the following table:

Expand table

Resource type	Hosting package	Example resource name	Override key
MySQL	Aspire.Hosting.MySql	`mysql`	`Parameters:mysql-password`
Oracle	Aspire.Hosting.Oracle	`oracle`	`Parameters:oracle-password`
PostgreSQL	Aspire.Hosting.PostgreSQL	`postgresql`	`Parameters:postgresql-password`
RabbitMQ	Aspire.Hosting.RabbitMq	`rabbitmq`	`Parameters:rabbitmq-password`
SQL Server	Aspire.Hosting.SqlServer	`sql`	`Parameters:sql-password`

By overriding the generated password, you can ensure that the password remains consistent between app launches. An alternative approach is to use the AddParameter

method to create a parameter that can be used as a password. The following code demonstrates how to create a persistent password for a SQL Server resource:

```C#
var sqlPassword = builder.AddParameter("sql-password", secret: true);

var sql = builder.AddSqlServer("sql", password: sqlPassword)
                 .WithDataVolume()
                 .AddDatabase("sqldb");
```

The `AddParameter` method is used to create a parameter named `sql-password` that's considered a secret. The `AddSqlServer` method is then called with the `password` parameter to set the password for the SQL Server resource. For more information, see External parameters.

Next steps

You can apply the volume concepts in the preceding code to a variety of services, including seeding a database with data that will persist across app launches. Try combining these techniques with the resource implementations demonstrated in the following tutorials:

- Tutorial: Connect an ASP.NET Core app to .NET Aspire storage integrations
- Tutorial: Connect an ASP.NET Core app to SQL Server using .NET Aspire and Entity Framework Core
- .NET Aspire orchestration overview

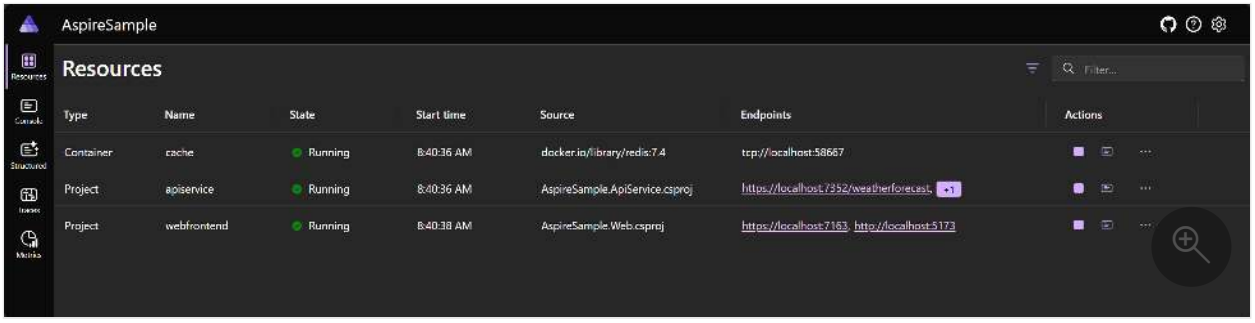
.NET Aspire dashboard overview

Article • 11/12/2024

.NET Aspire project templates offer a sophisticated dashboard for comprehensive app monitoring and inspection, and it's also available in standalone mode. This dashboard allows you to closely track various aspects of your app, including logs, traces, and environment configurations, in real-time. It's purpose-built to enhance the development experience, providing an insightful overview of your app's state and structure. The dashboard exposes the ability to stop, start, and restart resources, as well as view and interact with logs and telemetry.

Use the dashboard with .NET Aspire projects

The dashboard is integrated into the .NET Aspire *.*AppHost*. During development the dashboard is automatically launched when you start the project. It's configured to display the .NET Aspire project's resources and telemetry.



For more information about using the dashboard during .NET Aspire development, see Explore dashboard features.

Standalone mode

The .NET Aspire dashboard is also shipped as a Docker image and can be used standalone, without the rest of .NET Aspire. The standalone dashboard provides a great UI for viewing telemetry and can be used by any application.

Bash

```
docker run --rm -it -p 18888:18888 -p 4317:18889 -d --name aspire-
dashboard \
    mcr.microsoft.com/dotnet/aspire-dashboard:9.0
```

The preceding Docker command:

- Starts a container from the `mcr.microsoft.com/dotnet/aspire-dashboard:9.0` image.
- The container instance exposing two ports:
  - Maps the dashboard's OTLP port `18889` to the host's port `4317`. Port `4317` receives OpenTelemetry data from apps. Apps send data using OpenTelemetry Protocol (OTLP).
  - Maps the dashboard's port `18888` to the host's port `18888`. Port `18888` has the dashboard UI. Navigate to `http://localhost:18888` in the browser to view the dashboard.

For more information, see the Standalone .NET Aspire dashboard.

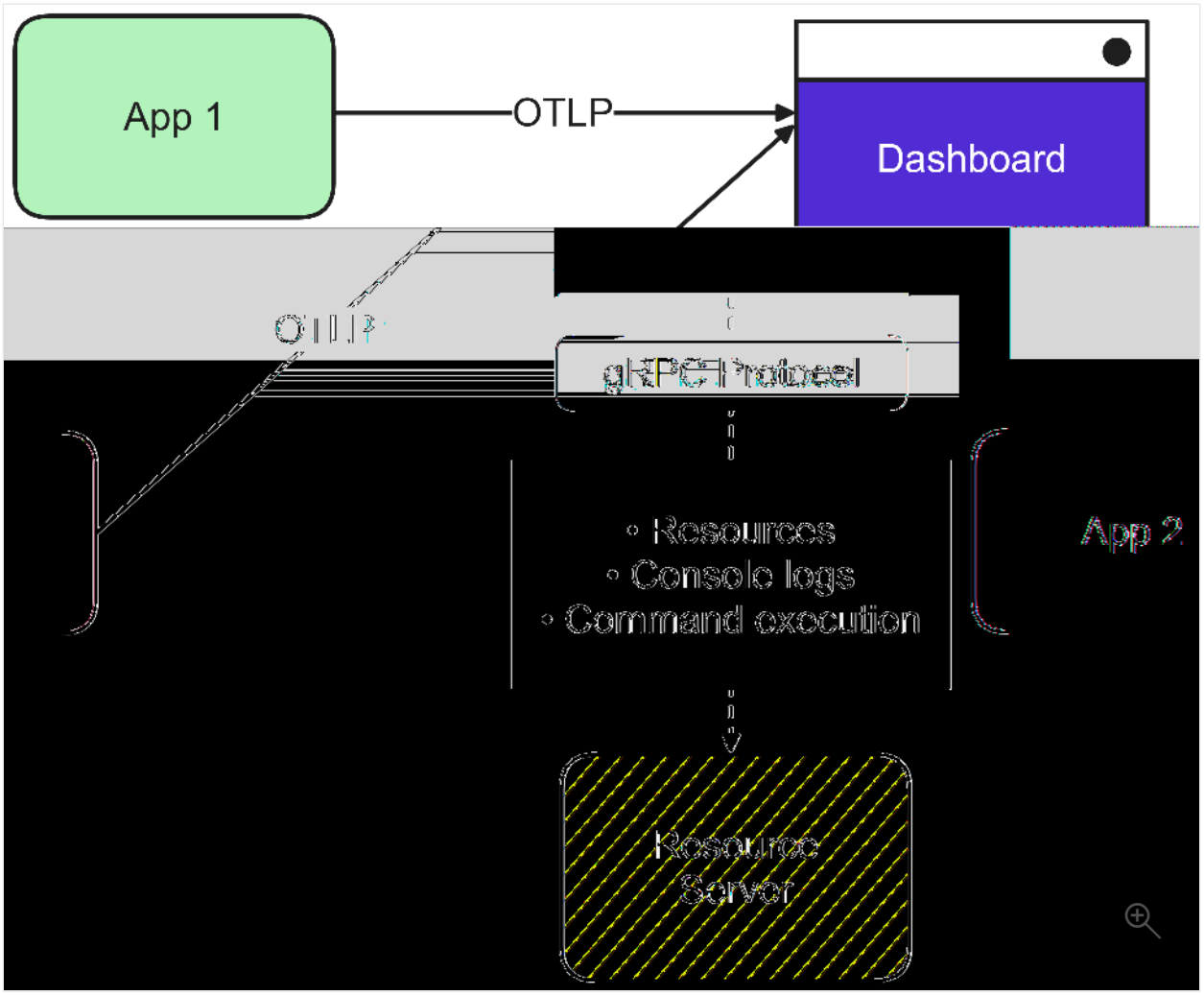
Configuration

The dashboard is configured when it starts up. Configuration includes frontend and OTLP addresses, the resource service endpoint, authentication, telemetry limits and more.

For more information, see .NET Aspire dashboard configuration.

Architecture

The dashboard user experience is built with a variety of technologies. The frontend is built with Grpc.AspNetCore) NuGet package NuGet package) to the resource server. Consider the following diagram that illustrates the architecture of the .NET Aspire dashboard:



Security

The .NET Aspire dashboard offers powerful insights to your apps. The UI displays information about resources, including their configuration, console logs and in-depth telemetry.

Data displayed in the dashboard can be sensitive. For example, configuration can include secrets in environment variables, and telemetry can include sensitive runtime data. Care should be taken to secure access to the dashboard.

For more information, see .NET Aspire dashboard security considerations.

Next steps

Explore the .NET Aspire dashboard

Explore the .NET Aspire dashboard

Article • 11/12/2024

In the upcoming sections, you discover how to create a .NET Aspire project and embark on the following tasks:

- ✔ Investigate the dashboard's capabilities by using the app generated from the project template as explained in the Quickstart: Build your first .NET Aspire project.
- ✔ Delve into the features of the .NET Aspire dashboard app.

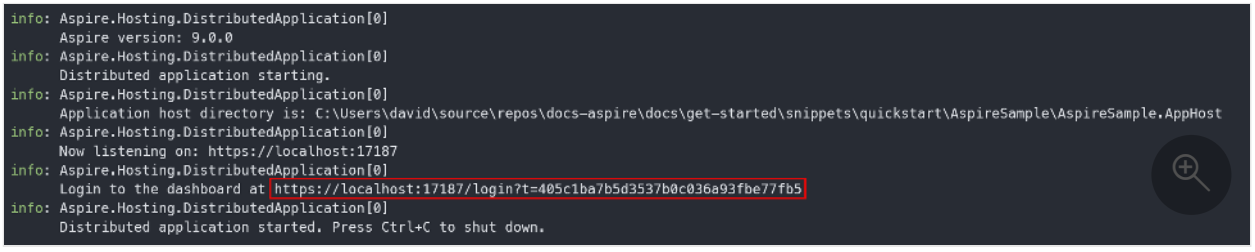
The screenshots featured in this article showcase the dark theme. For more information on theme selection, see Theme selection.

Dashboard authentication

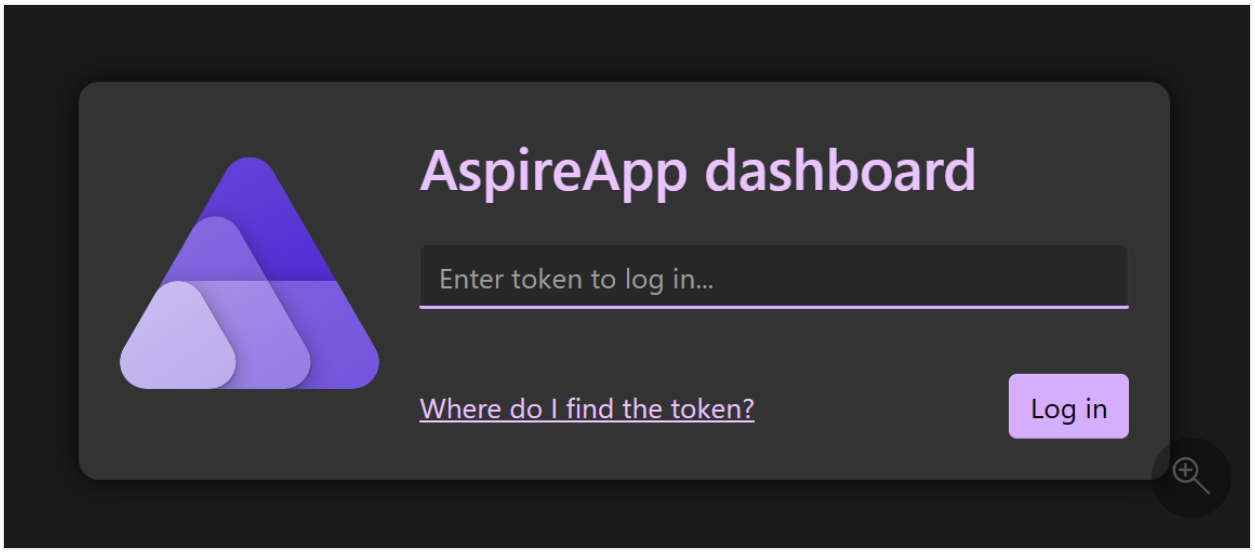
When you run a .NET Aspire app host, the orchestrator starts up all the app's dependent resources and then opens a browser window to the dashboard. The .NET Aspire dashboard requires token-based authentication for its users because it displays environment variables and other sensitive information.

When the dashboard is launched from Visual Studio or Visual Studio Code (with the C# Dev Kit extension), the browser is automatically logged in, and the dashboard opens directly. This is the typical developer `F5` experience, and the authentication login flow is automated by the .NET Aspire tooling.

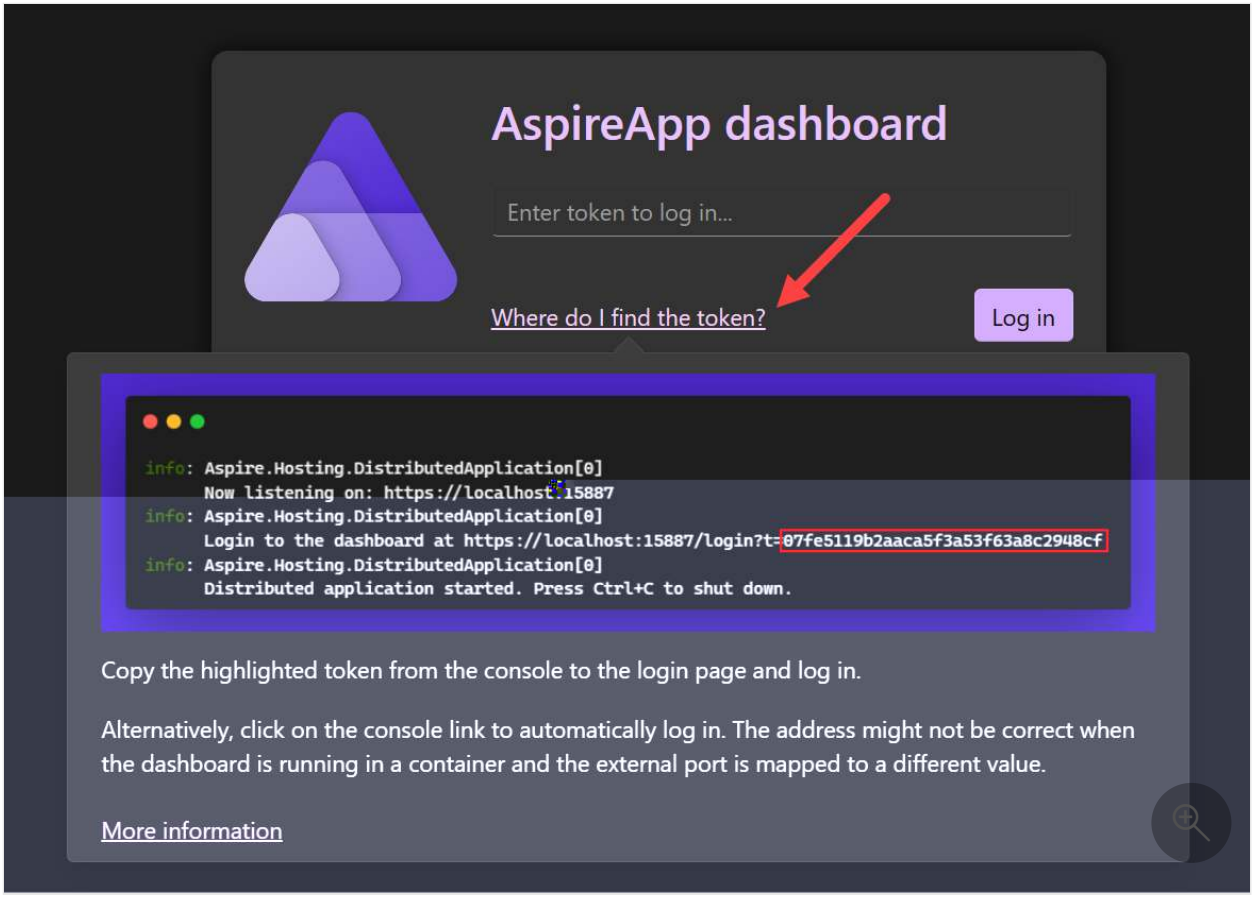
However, if you start the app host from the command line, you're presented with the login page. The console window displays a URL that you can select on to open the dashboard in your browser.



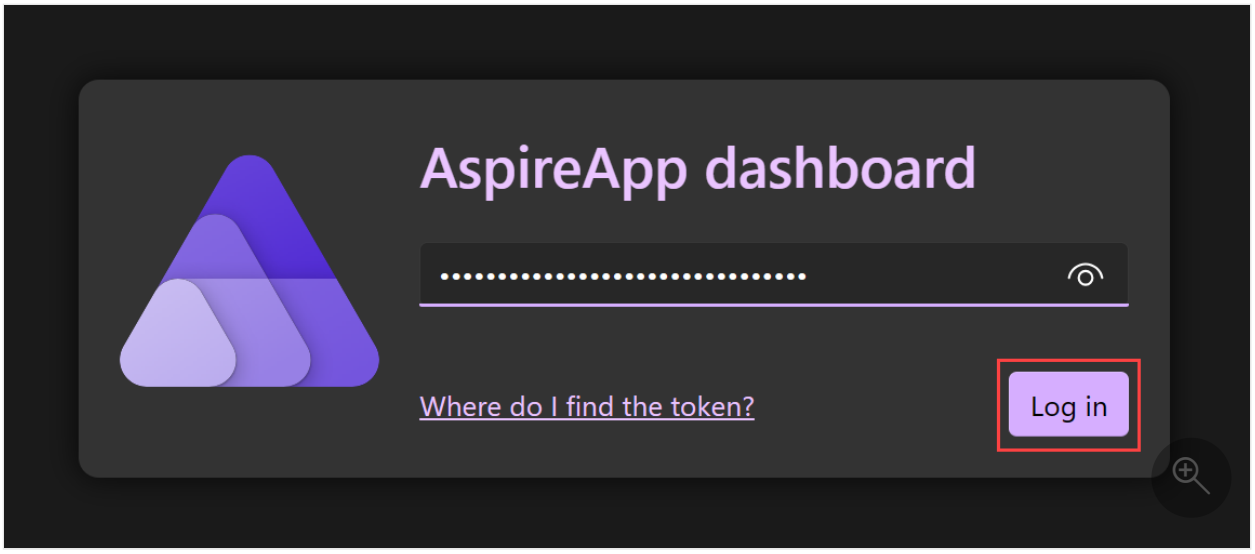
The URL contains a token query string (with the token value mapped to the `t` name part) that's used to *log in* to the dashboard. If your console supports it, you can hold the `Ctrl` key and then select the link to open the dashboard in your browser. This method is easier than copying the token from the console and pasting it into the login page. If you end up on the dashboard login page without either of the previously described methods, you can always return to the console to copy the token.



The login page accepts a token and provides helpful instructions on how to obtain the token, as shown in the following screenshot:



After copying the token from the console and pasting it into the login page, select the **Log in** button.



The dashboard persists the token as a browser persistent cookie, which remains valid for three days. Persistent cookies have an expiration date and remain valid even after closing the browser. This means that users don't need to log in again if they close and reopen the browser. For more information, see the Security considerations for running the .NET Aspire dashboard documentation.

Resources page

The **Resources** page is the default home page of the .NET Aspire dashboard. This page lists all of the .NET projects, containers, and executables included in your .NET Aspire solution. For example, the starter application includes two projects:

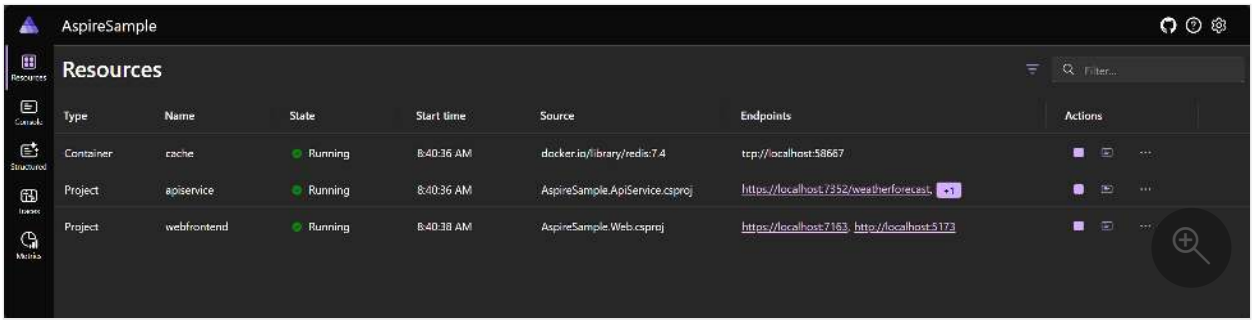
- **apiservice**: A back-end API project built using Minimal APIs.
- **webfrontend**: The front-end UI project built using Blazor.

The dashboard also provides essential details about each resource:

- **Type**: Displays whether the resource is a project, container, or executable.
- **Name**: The name of the resource.
- **State**: Displays whether or not the resource is currently running.
  - **Errors**: Within the **State** column, errors are displayed as a badge with the error count. It's useful to understand quickly what resources are reporting errors. Selecting the badge takes you to the semantic logs for that resource with the filter at an error level.
- **Start time**: When the resource started running.
- **Source**: The location of the resource on the device.
- **Endpoints**: One or more URLs to reach the running resource directly.
- **Logs**: A link to the resource logs page.
- **Actions**: A set of actions that can be performed on the resource:
  - **Stop / Start**: Stop (or Start) the resource—depending on the current **State**.

- **Console logs**: Navigate to the resource's console logs.
- **Ellipsis**: A submenu with extra resource specific actions:
  - **View details**: View the resource details.
  - **Console log**: Navigate to the resource's console logs.
  - **Structured logs**: Navigate to the resource's structured logs.
  - **Traces**: Navigate to the resource's traces.
  - **Metrics**: Navigate to the resource's metrics.
  - **Restart**: Stop and then start the resource.

Consider the following screenshot of the resources page:

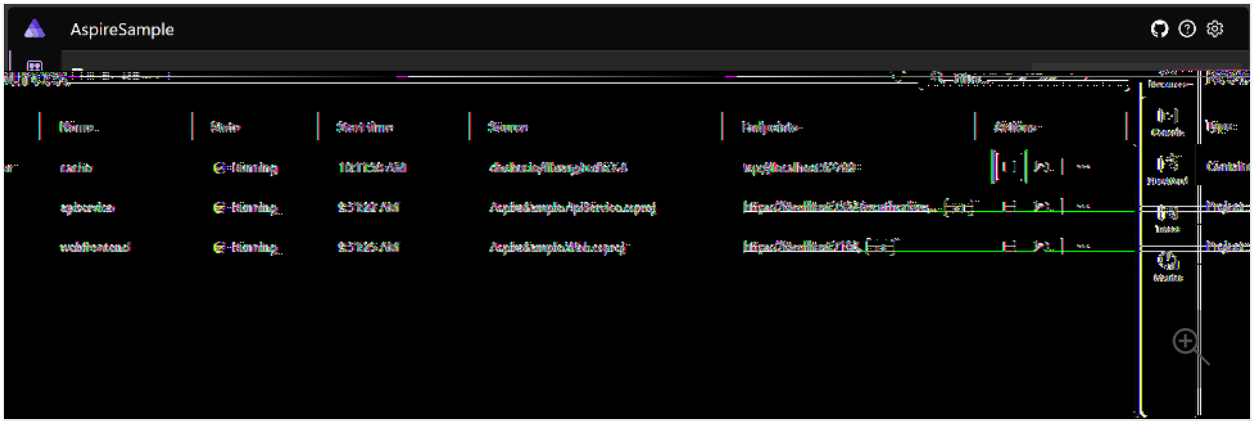


Resource actions

Each resource has a set of available actions that are conditionally enabled based on the resource's current state. For example, if a resource is running, the **Stop** action is enabled. If the resource is stopped, the **Start** action is enabled. Likewise, some actions are disabled when they're unavailable, for example, some resources don't have structured logs. In these situations, the **Structured logs** action is disabled.

Stop or Start a resource

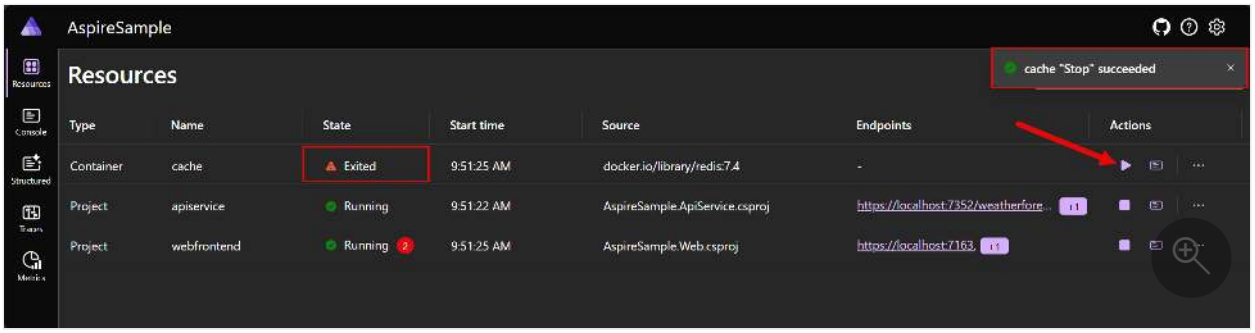
The .NET Aspire dashboard allows you to stop or start a resource by selecting the **Stop** or **Start** button in the **Actions** column. Consider the following screenshot of the resources page with the **Stop** button selected:



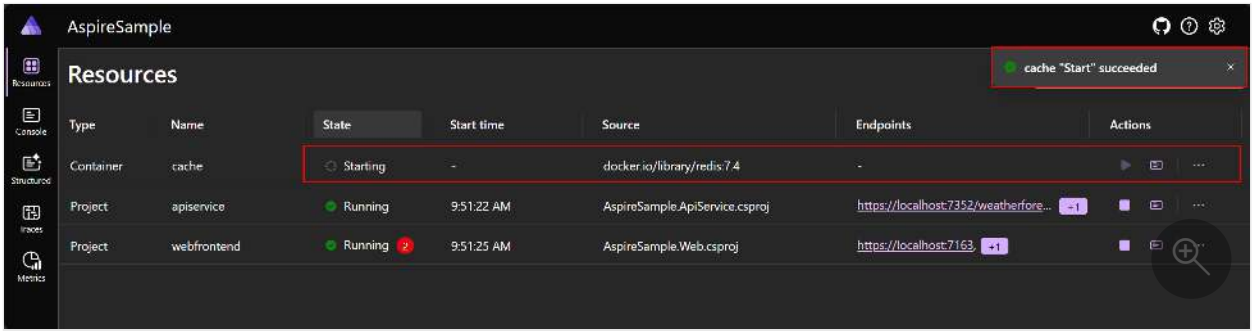
When you select **Stop**, the resource stops running, and the **State** column updates to reflect the change.

> **Note**
>
> For project resources, when the debugger is attached, it's reattached on restart.

The **Start** button is then enabled, allowing you to start the resource again. Additionally, the dashboard displays a toast notification of the result of the action:



When a resource is in a non-running state, the **Start** button is enabled. Selecting **Start** starts the resource, and the **State** column updates to reflect the change. The **Stop** button is then enabled, allowing you to stop the resource again. The dashboard displays a toast notification of the result of the action:

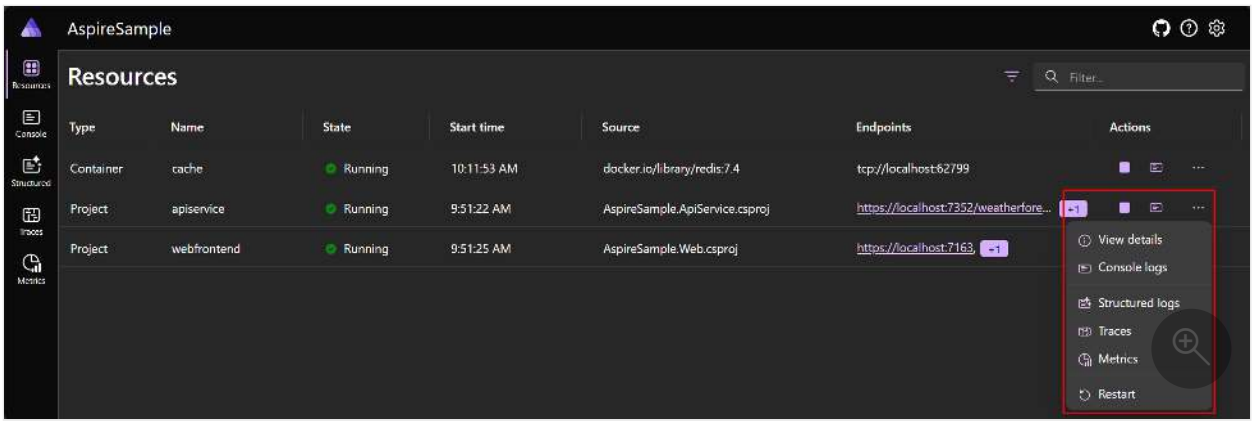


> **Tip**
>
> Resources that depend on other resources that are stopped, or restarted, might experience temporary errors. This is expected behavior and is typically resolved when the dependent resources are in a **Running** state once again.

Resource submenu actions

Selecting the horizontal ellipsis icon in the **Actions** column opens a submenu with additional resource-specific actions. In addition to the built-in resource submenu actions, you can also define custom resource actions by defining custom commands. For

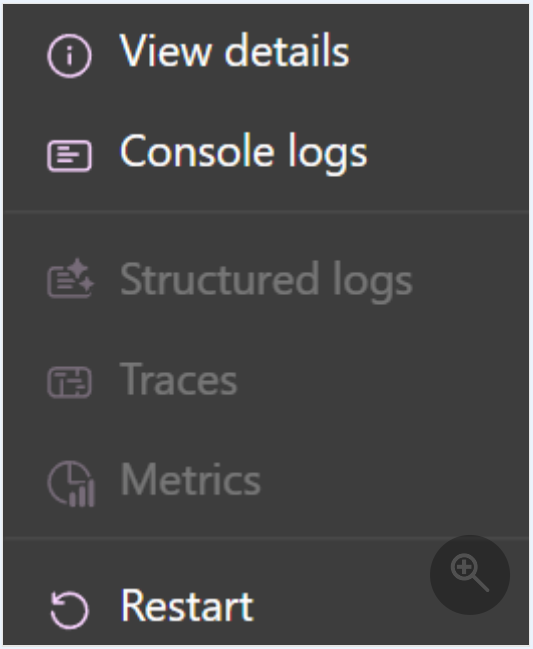
more information, see Custom resource commands in .NET Aspire. For the built-in resource submenu actions, consider the following screenshot:



The following submenu actions are available:

- **View details**: View the resource details.
- **Console log**: Navigate to the resource's console logs.
- **Structured logs**: Navigate to the resource's structured logs.
- **Traces**: Navigate to the resource's traces.
- **Metrics**: Navigate to the resource's metrics.
- **Restart**: Stop and then start the resource.

> **Important**
>
> There might be resources with disabled submenu actions. They're greyed out when they're disabled. For example, the following screenshot shows the submenu actions disabled:

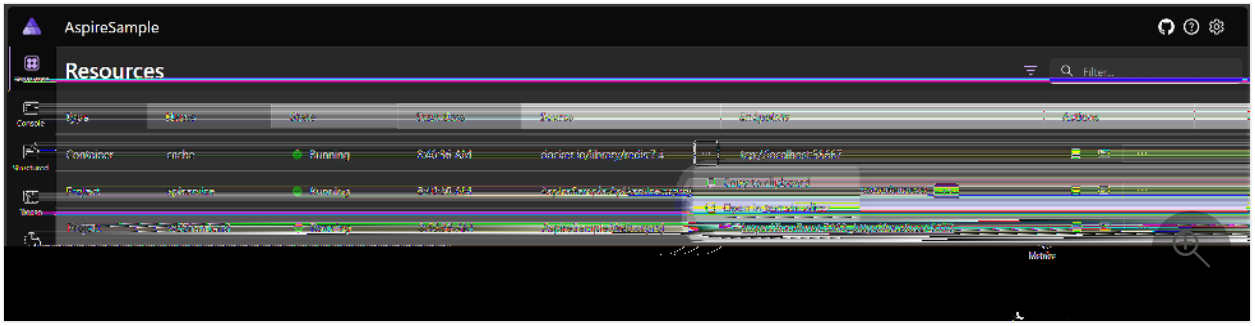


Copy or Open in text visualizer

To view a *text visualizer* of certain columns, on hover you see a vertical ellipsis icon. Select the icon to display the available options:

- **Copy to clipboard**
- **Open in text visualizer**

Consider the following screenshot of the ellipsis menu options:

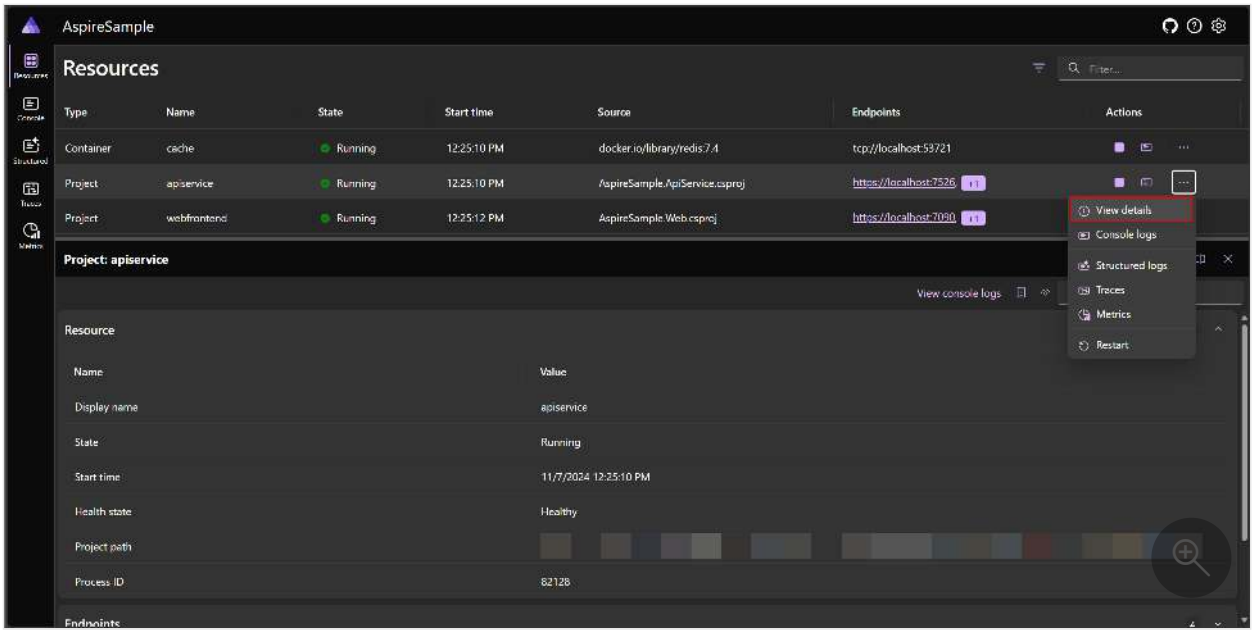


When you select the **Open in text visualizer** option, a modal dialog opens with the text displayed in a larger format. Consider the following screenshot of the text visualizer modal dialog:

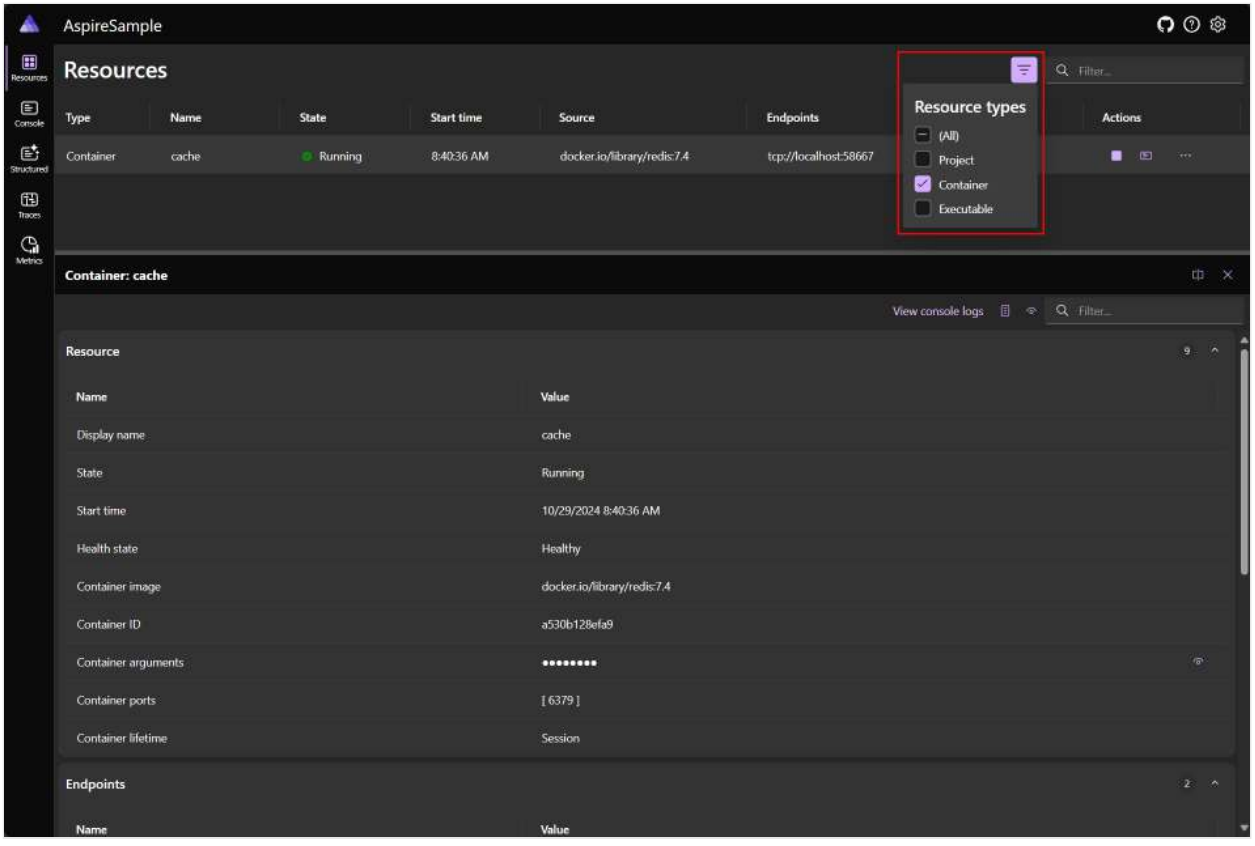
Some values are formatted as JSON or XML. In these cases, the text visualizer enables the **Select format** dropdown to switch between the different formats.

Resource details

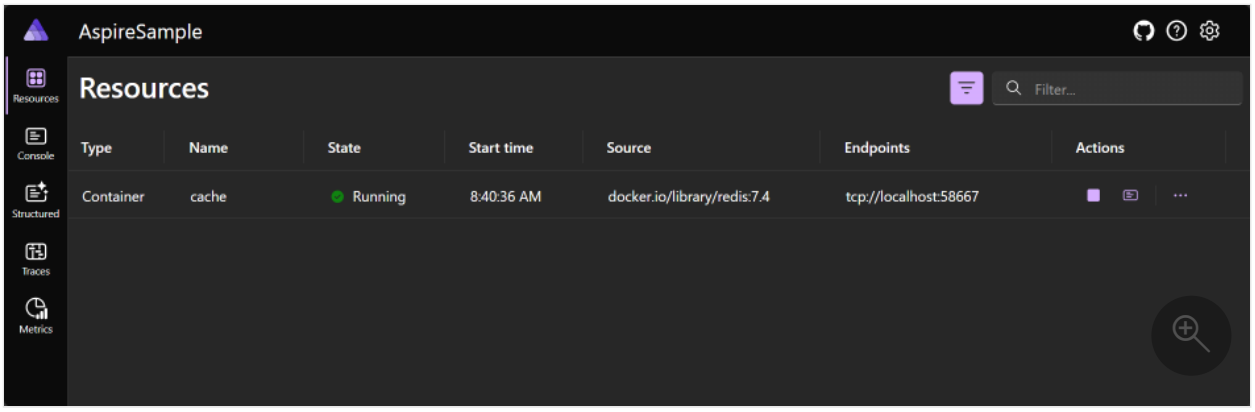
You can obtain full details about each resource by selecting the ellipsis button in the **Actions** column and then selecting **View details**. The **Details** page provides a comprehensive view of the resource:



The search bar in the upper right of the dashboard also provides the option to filter the list, which is useful for .NET Aspire projects with many resources. To select the types of resources that are displayed, drop down the arrow to the left of the filter textbox:

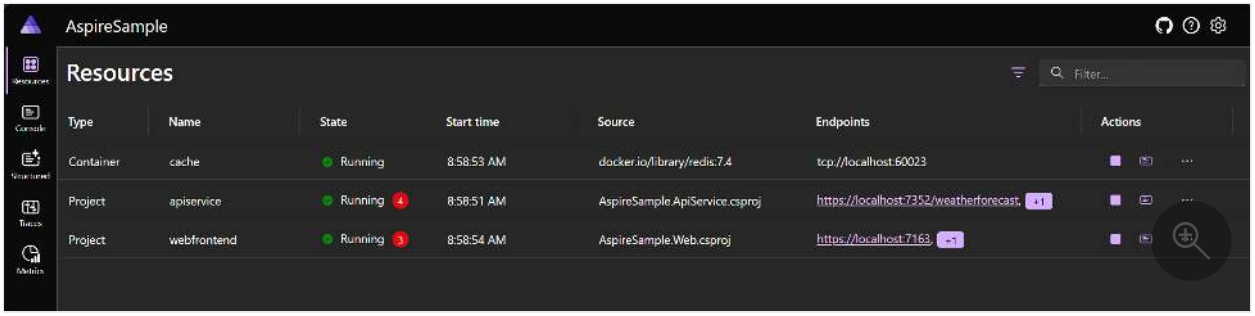


In this example, only containers are displayed in the list. For example, if you enable **Use Redis for caching** when creating a .NET Aspire project, you should see a Redis container listed:

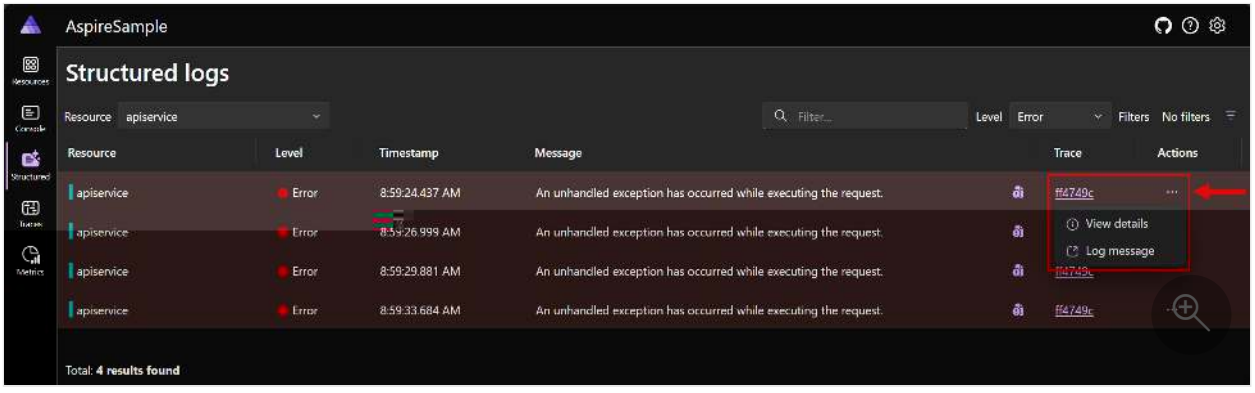


Executables are stand-alone processes. You can configure a .NET Aspire project to run a stand-alone executable during startup, though the default starter templates don't include any executables by default.

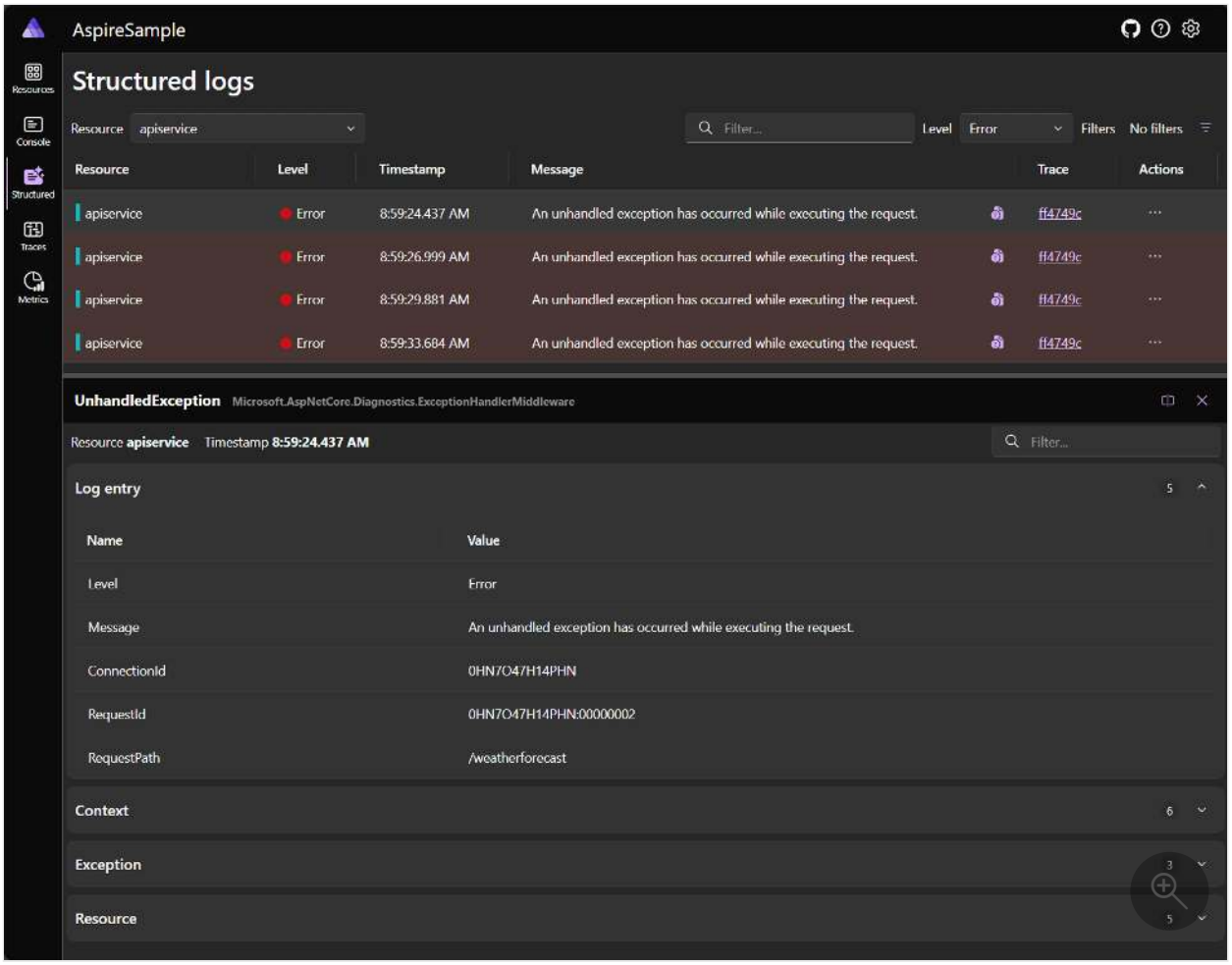
The following screenshot shows an example of a project that has errors:



Selecting the error count badge navigates to the Structured logs page with a filter applied to show only the logs relevant to the resource:



To see the log entry in detail for the error, select the **View** button to open a window below the list with the structured log entry details:



For more information and examples of Structured logs, see the Structured logs page section.

> **Note**
>
> The resources page isn't available if the dashboard is started without a configured resource service. The dashboard starts on the Structured logs page instead. This is the default experience when the dashboard is run in standalone mode without additional configuration.
>
> For more information about configuring a resource service, see Dashboard configuration.

Monitoring pages

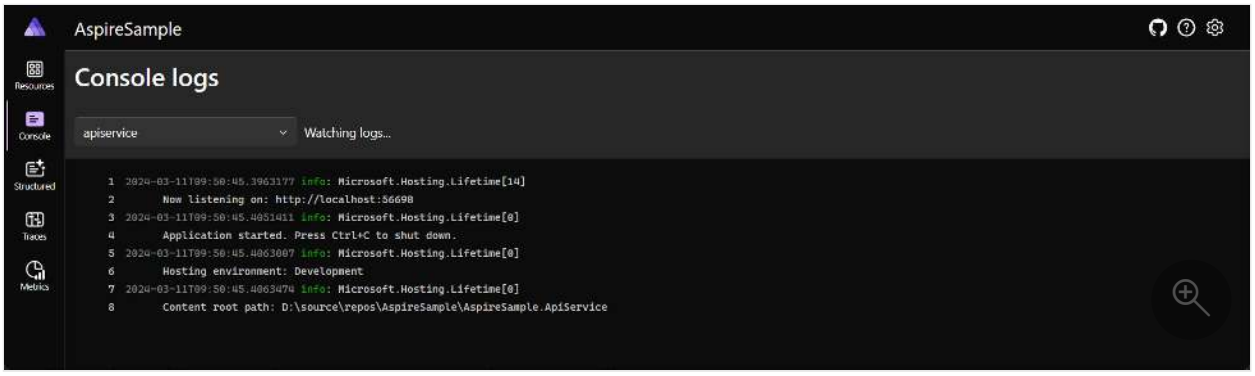
The .NET Aspire dashboard provides various ways to view logs, traces, and metrics for your app. This information enables you to track the behavior and performance of your app and to diagnose any issues that arise.

Console logs page

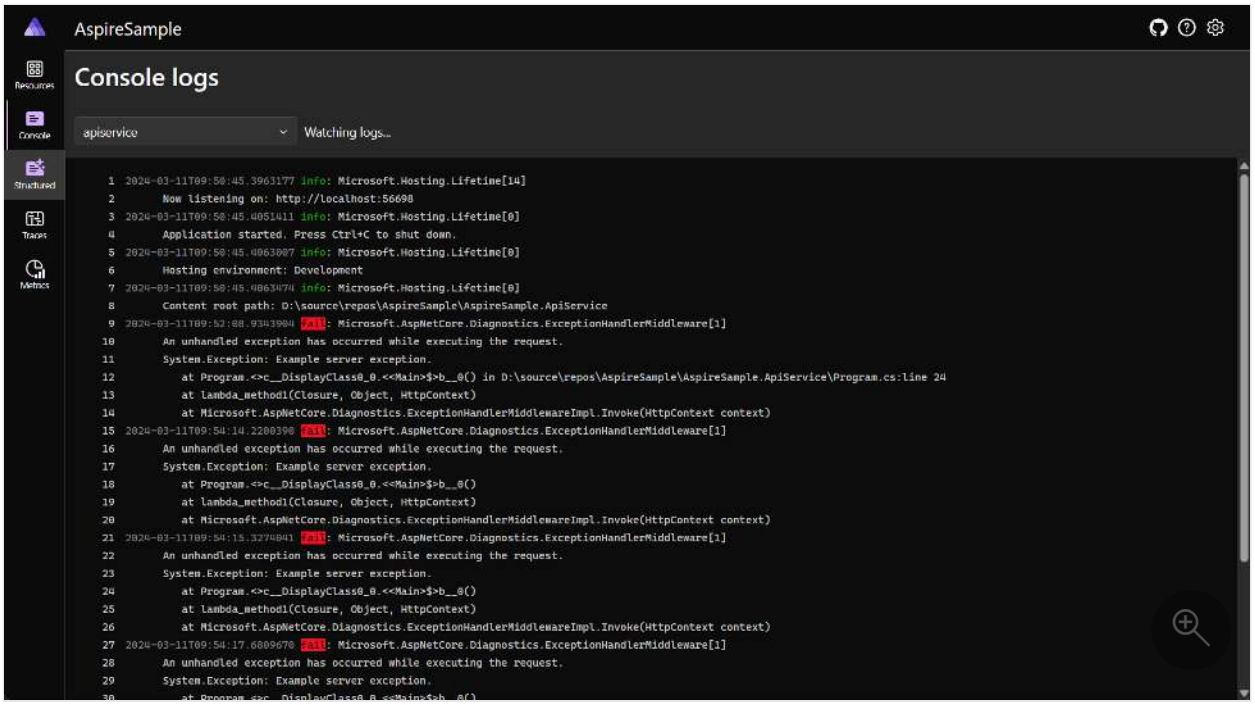
The **Console logs** page displays text that each resource in your app has sent to standard output. Logs are a useful way to monitor the health of your app and diagnose issues. Logs are displayed differently depending on the source, such as project, container, or executable.

When you open the Console logs page, you must select a source in the **Select a resource** drop-down list.

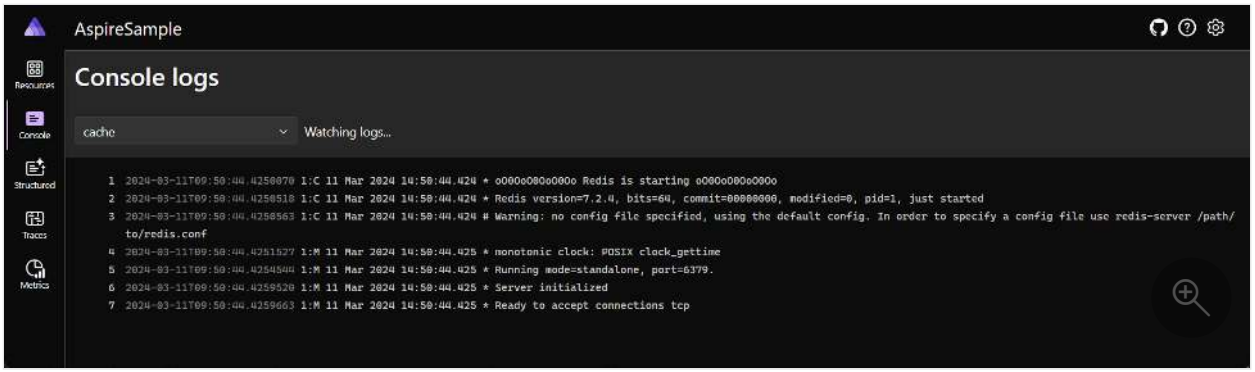
If you select a project, the live logs are rendered with a stylized set of colors that correspond to the severity of the log; green for information as an example. Consider the following example screenshot of project logs with the `apiservice` project selected:



When errors occur, they're styled in the logs such that they're easy to identify. Consider the following example screenshot of project logs with errors:

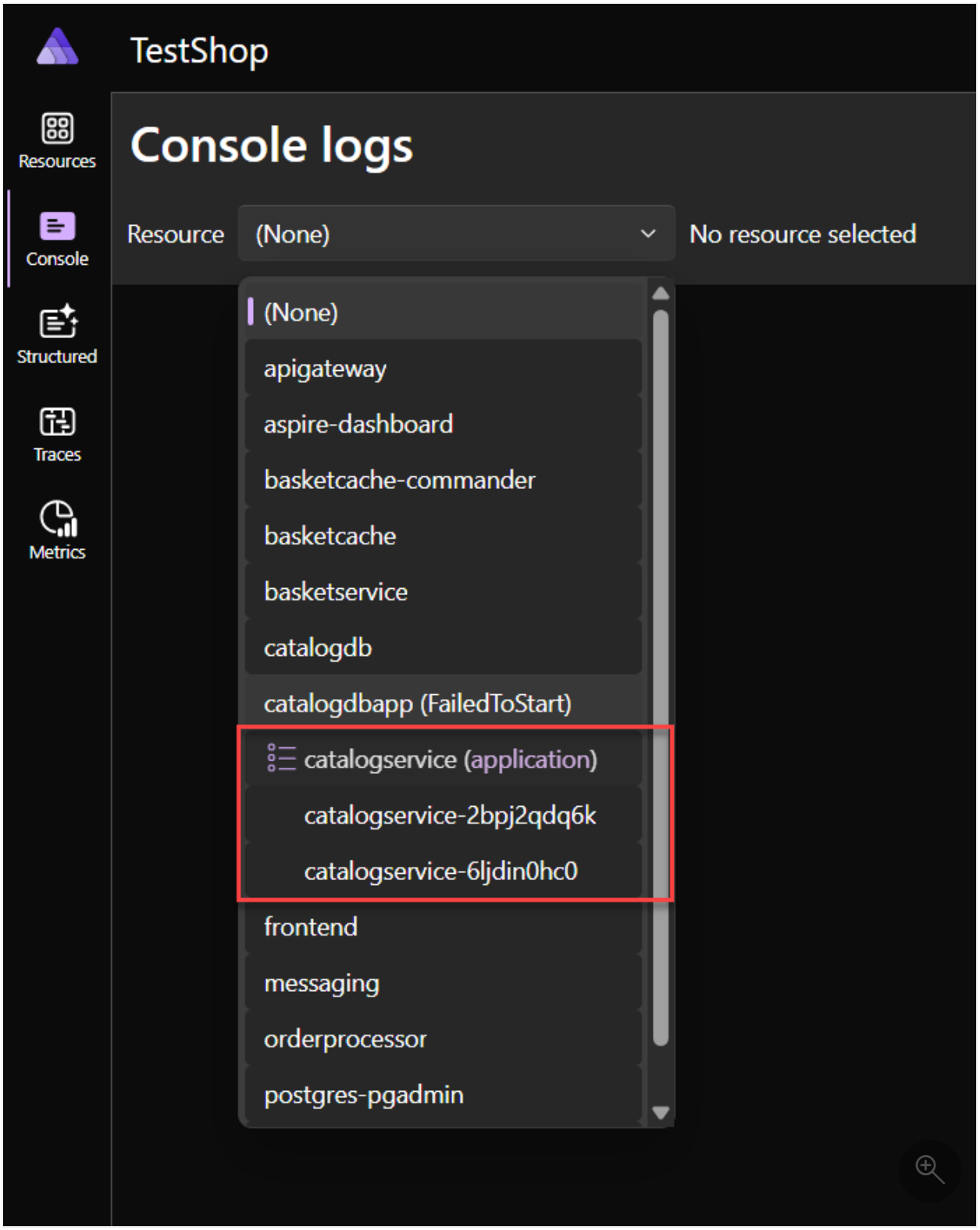


If you select a container or executable, formatting is different from a project but verbose behavior information is still available. Consider the following example screenshot of a container log with the `cache` container selected:



Resource replicas

When project resources are replicated using the WithReplicas API, they're represented in the resource selector under a top-level named resource entry with an icon to indicator. Each replicated resource is listed under the top-level resource entry, with its corresponding unique name. Consider the following example screenshot of a replicated project resource:



The preceding screenshot shows the `catalogservice (application)` project with two replicas, `catalogservice-2bpj2qdq6k` and `catalogservice-6ljdin0hc0`. Each replica has its own set of logs that can be viewed by selecting the replica name.

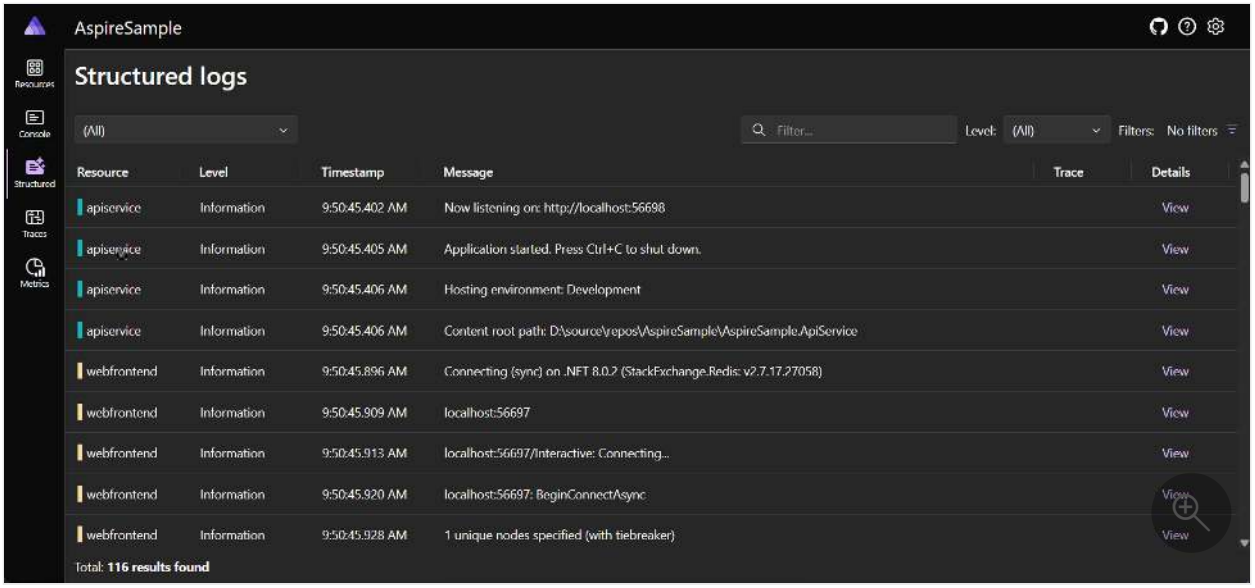
Structured logs page

.NET Aspire automatically configures your projects with logging using OpenTelemetry. Navigate to the **Structured logs** page to view the semantic logs for your .NET Aspire project. Semantic, or structured logging makes it easier to store and query log-events,

as the log-event message-template and message-parameters are preserved, instead of just transforming them into a formatted message. You notice a clean structure for the different logs displayed on the page using columns:

- **Resource**: The resource the log originated from.
- **Level**: The log level of the entry, such as information, warning, or error.
- **Timestamp**: The time that the log occurred.
- **Message**: The details of the log.
- **Trace**: A link to the relevant trace for the log, if applicable.
- **Details**: Additional details or metadata about the log entry.

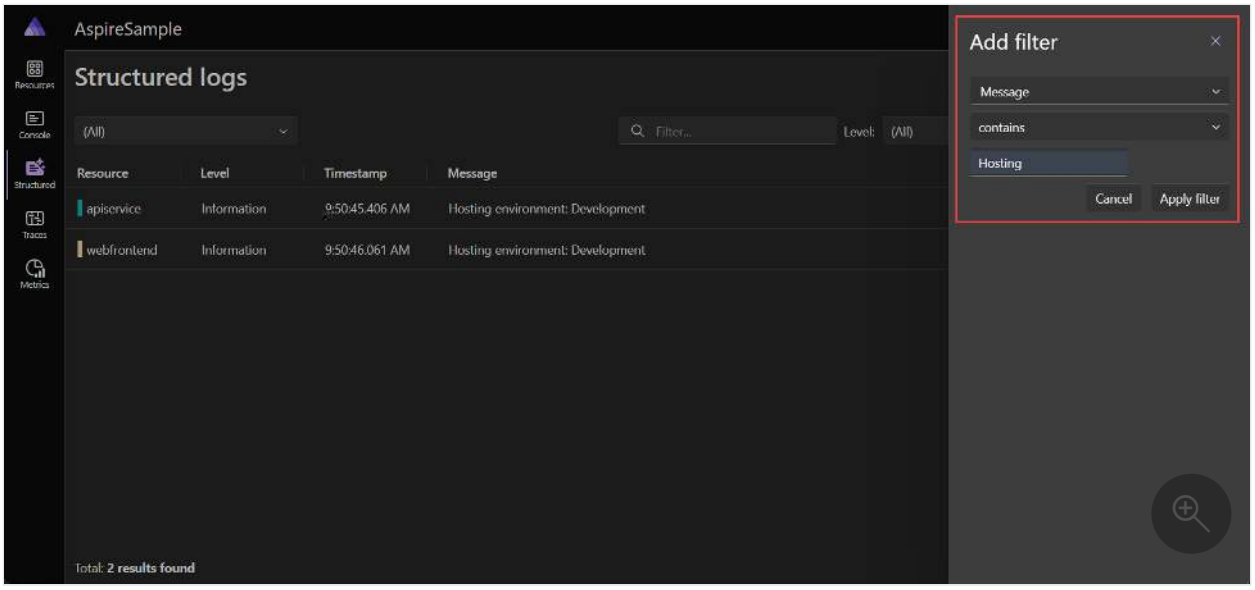
Consider the following example screenshot of semantic logs:



Filter structured logs

The structured logs page also provides a search bar to filter the logs by service, level, or message. You use the **Level** drop down to filter by log level. You can also filter by any log property by selecting the filter icon button, which opens the advanced filter dialog.

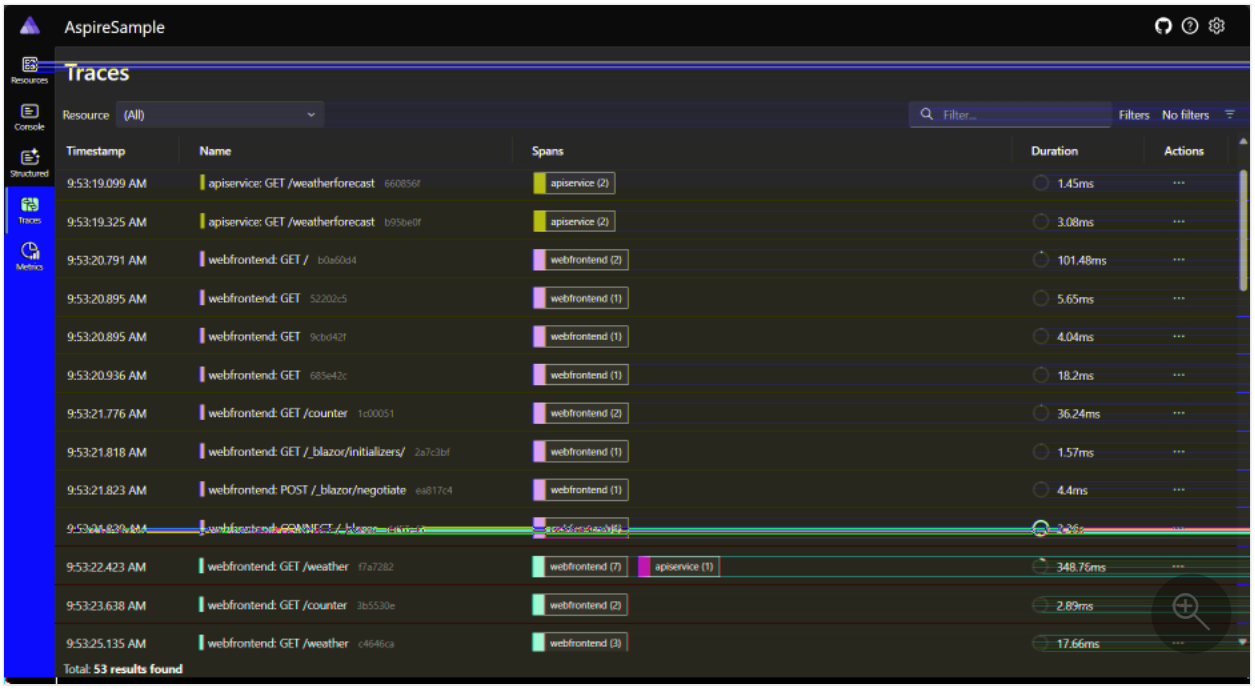
Consider the following screenshots showing the structured logs, filtered to display items with "Hosting" in the message text:



Traces page

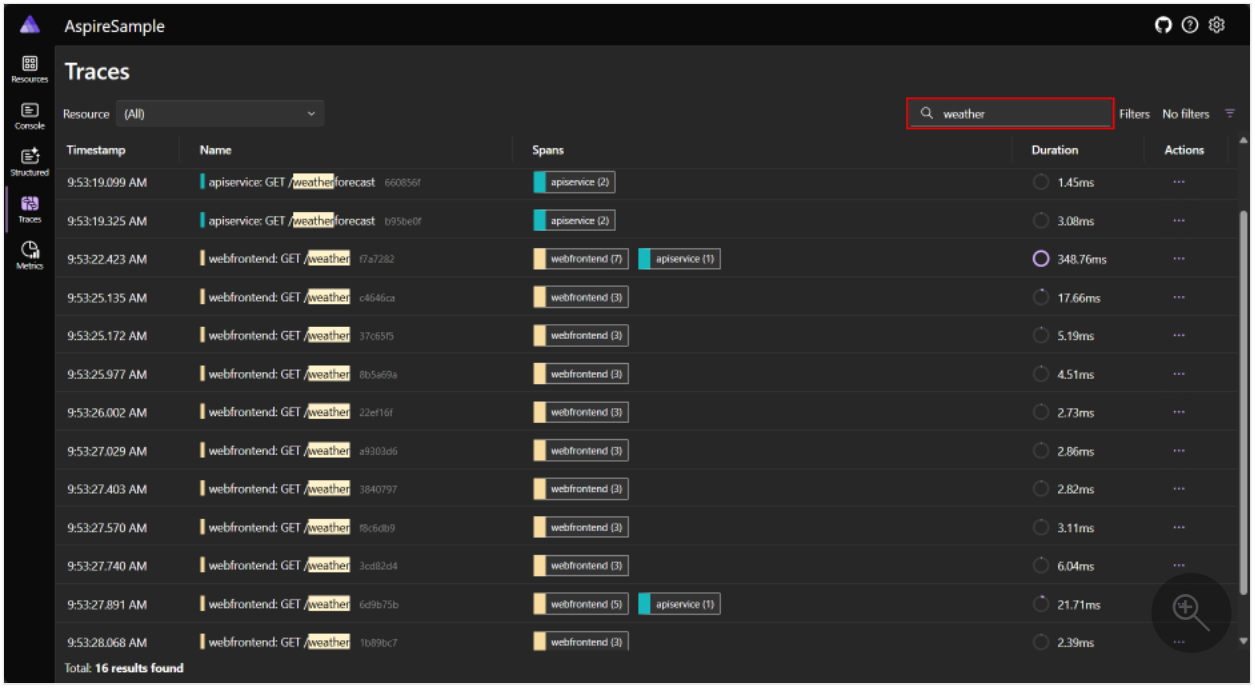
Navigate to the **Traces** page to view all of the traces for your app. .NET Aspire automatically configures tracing for the different projects in your app. Distributed tracing is a diagnostic technique that helps engineers localize failures and performance issues within applications, especially those that might be distributed across multiple machines or processes. For more information, see .NET distributed tracing. This technique tracks requests through an application and correlates work done by different application integrations. Traces also help identify how long different stages of the request took to complete. The traces page displays the following information:

- **Timestamp**: When the trace completed.
- **Name**: The name of the trace, prefixed with the project name.
- **Spans**: The resources involved in the request.
- **Duration**: The time it took to complete the request. This column includes a radial icon that illustrates the duration of the request in comparison with the others in the list.

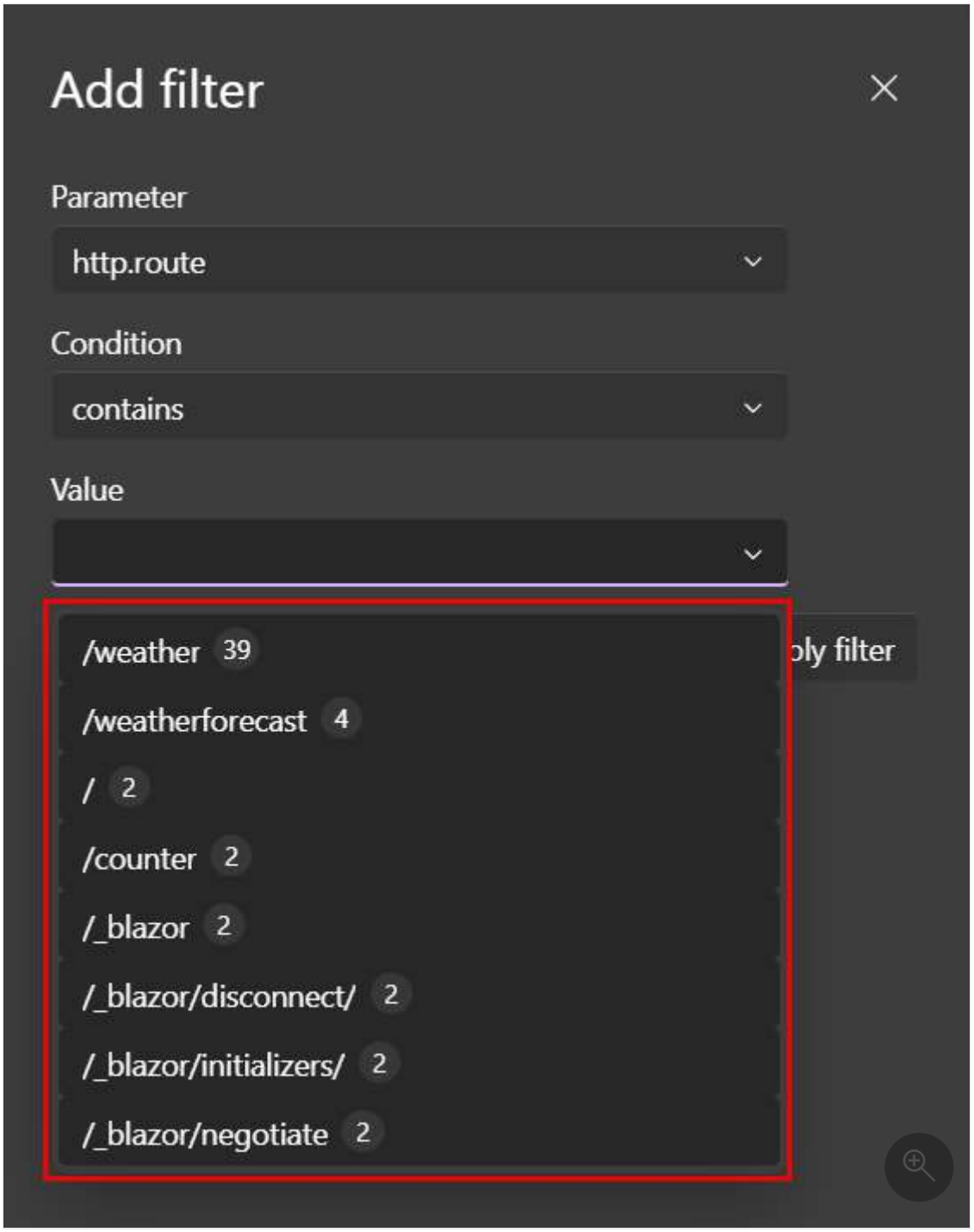


Filter traces

The traces page also provides a search bar to filter the traces by name or span. Apply a filter, and notice the trace results are updated immediately. Consider the following screenshot of traces with a filter applied to `weather` and notice how the search term is highlighted in the results:

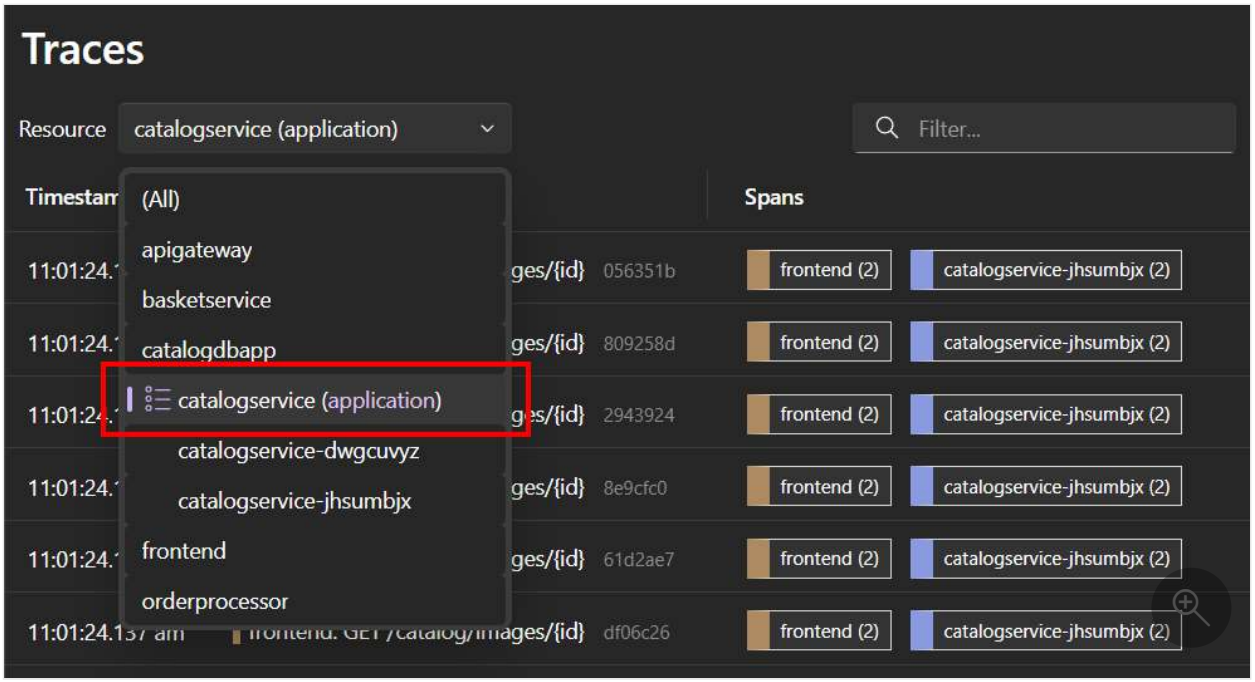


When filtering traces in the **Add filter** dialog, after selecting a **Parameter** and corresponding **Condition**, the **Value** selection is pre-populated with the available values for the selected parameter. Consider the following screenshot of the **Add filter** dialog with the `http.route` parameter selected:



Combine telemetry from multiple resources

When a resource has multiple replicas, you can filter telemetry to view data from all instances at once. Select the parent resource, labeled `(application)`, as shown in the following screenshot:



After selecting the parent resource, the traces page displays telemetry from all instances of the resource.

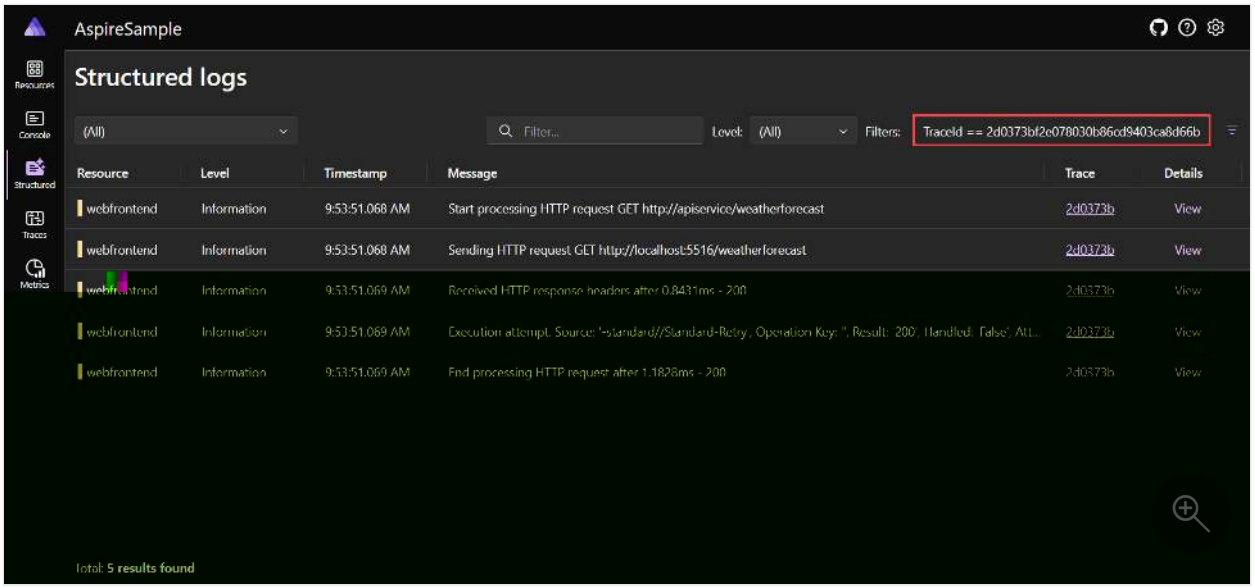
Trace details

The trace details page contains various details pertinent to the request, including:

- **Trace Detail**: When the trace started.
- **Duration**: The time it took to complete the request.
- **Resources**: The number of resources involved in the request.
- **Depth**: The number of layers involved in the request.
- **Total Spans**: The total number of spans involved in the request.

Each span is represented as a row in the table, and contains a **Name**. Spans also display the error icon if an error occurred within that particular span of the trace. Spans that have a type of client/consumer, but don't have a span on the server, show an arrow icon and then the destination address. This represents a client call to a system outside of the .NET Aspire project. For example, an HTTP request an external web API, or a database call.

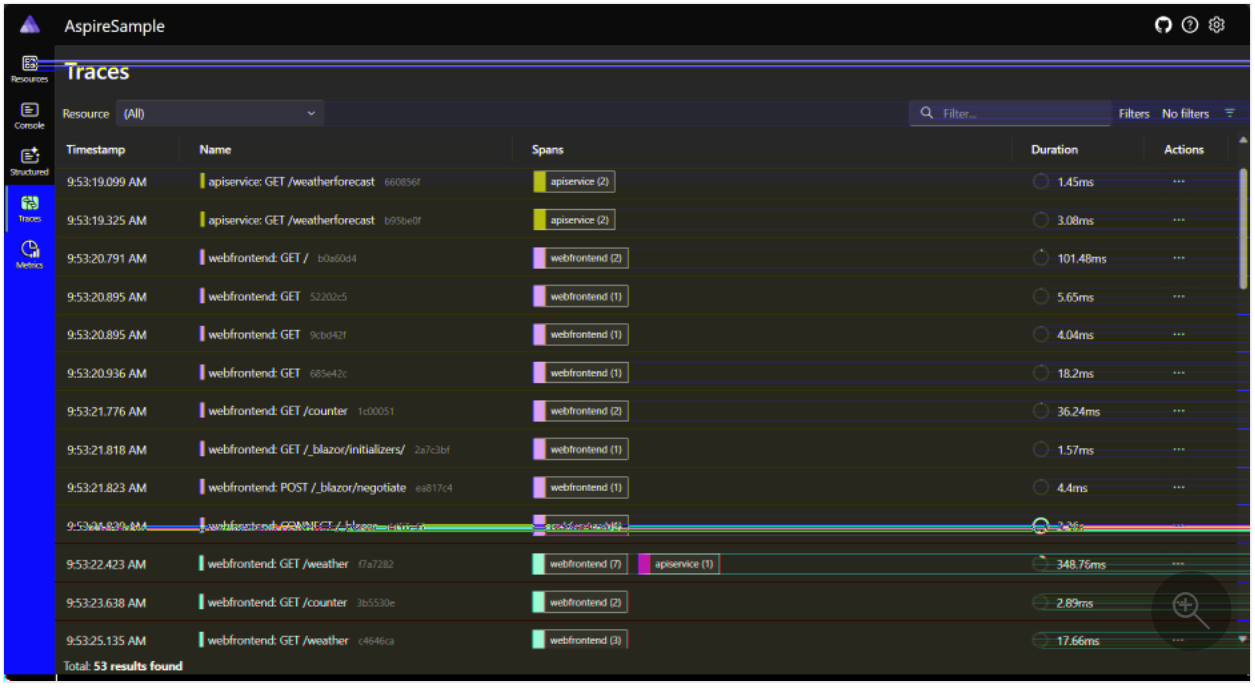
Within the trace details page, there's a **View Logs** button that takes you to the structured logs page with a filter applied to show only the logs relevant to the request. Consider an example screenshot depicting the structured logs page with a filter applied to show only the logs relevant to the trace:



The structured logs page is discussed in more detail in the Structured logs page section.

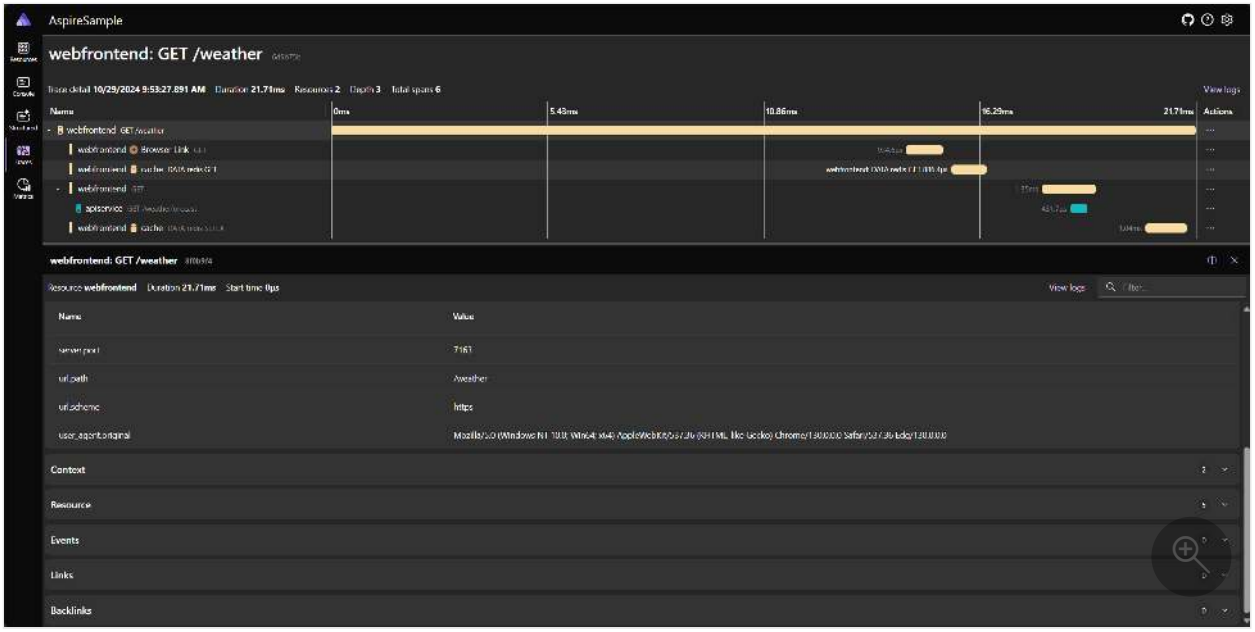
Trace examples

Each trace has a color, which is generated to help differentiate between spans—one color for each resource. The colors are reflected in both the *traces page* and the *trace detail page*. When traces depict an arrow icon, those icons are colorized as well to match the span of the target trace. Consider the following example screenshot of traces:

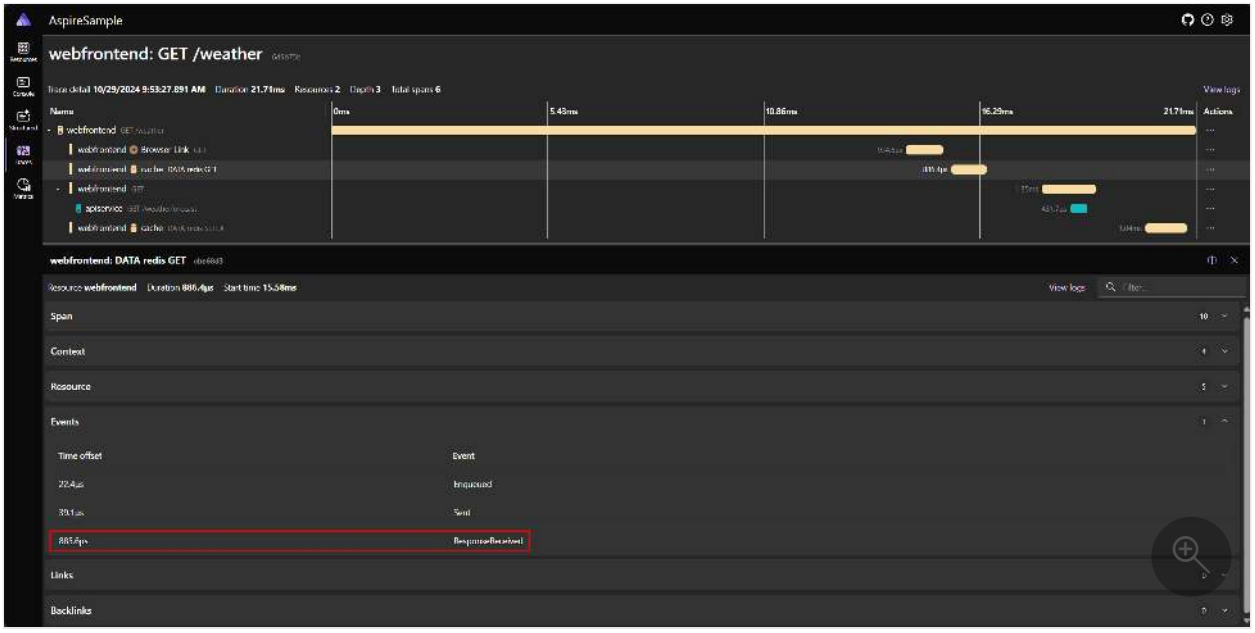


You can also select the **View** button to navigate to a detailed view of the request and the duration of time it spent traveling through each application layer. Consider an example selection of a trace to view its details:

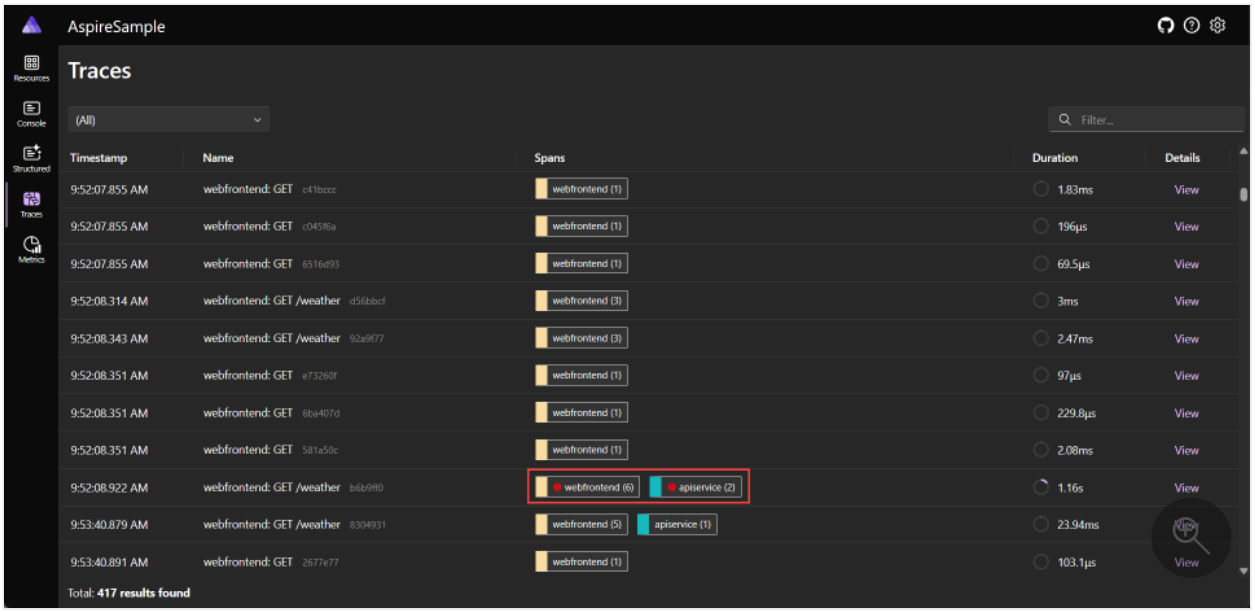
For each span in the trace, select **View** to see more details:



Scroll down in the span details pain to see full information. At the bottom of the span details pane, some span types, such as this call to a cache, show span event timings:



When errors are present, the page renders an error icon next to the trace name. Consider an example screenshot of traces with errors:



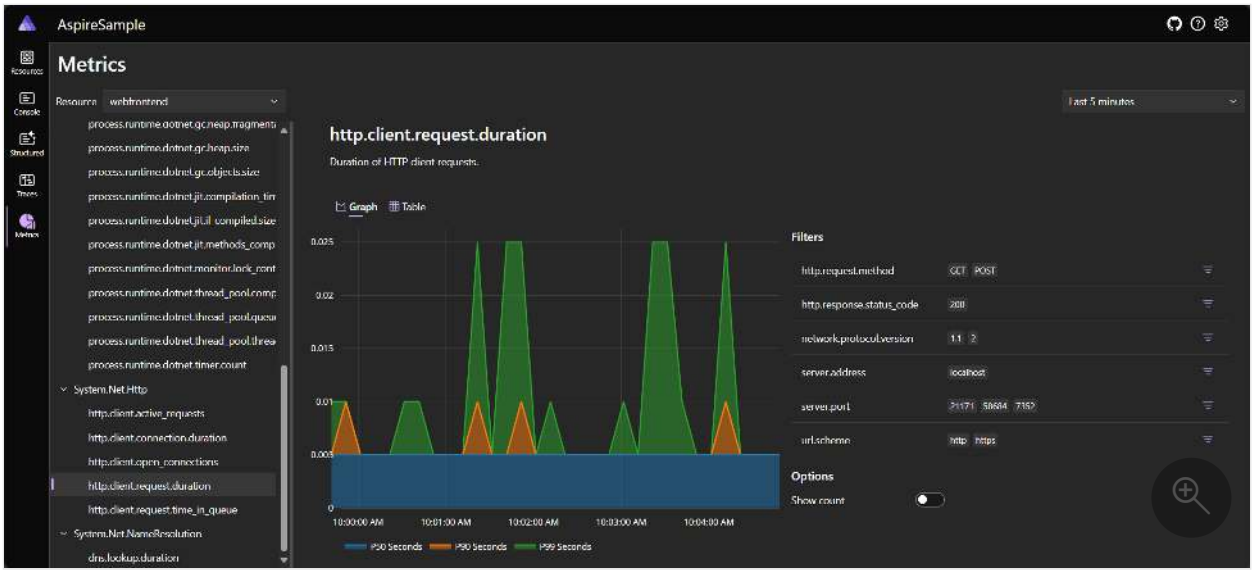
And the corresponding detailed view of the trace with errors:

Metrics page

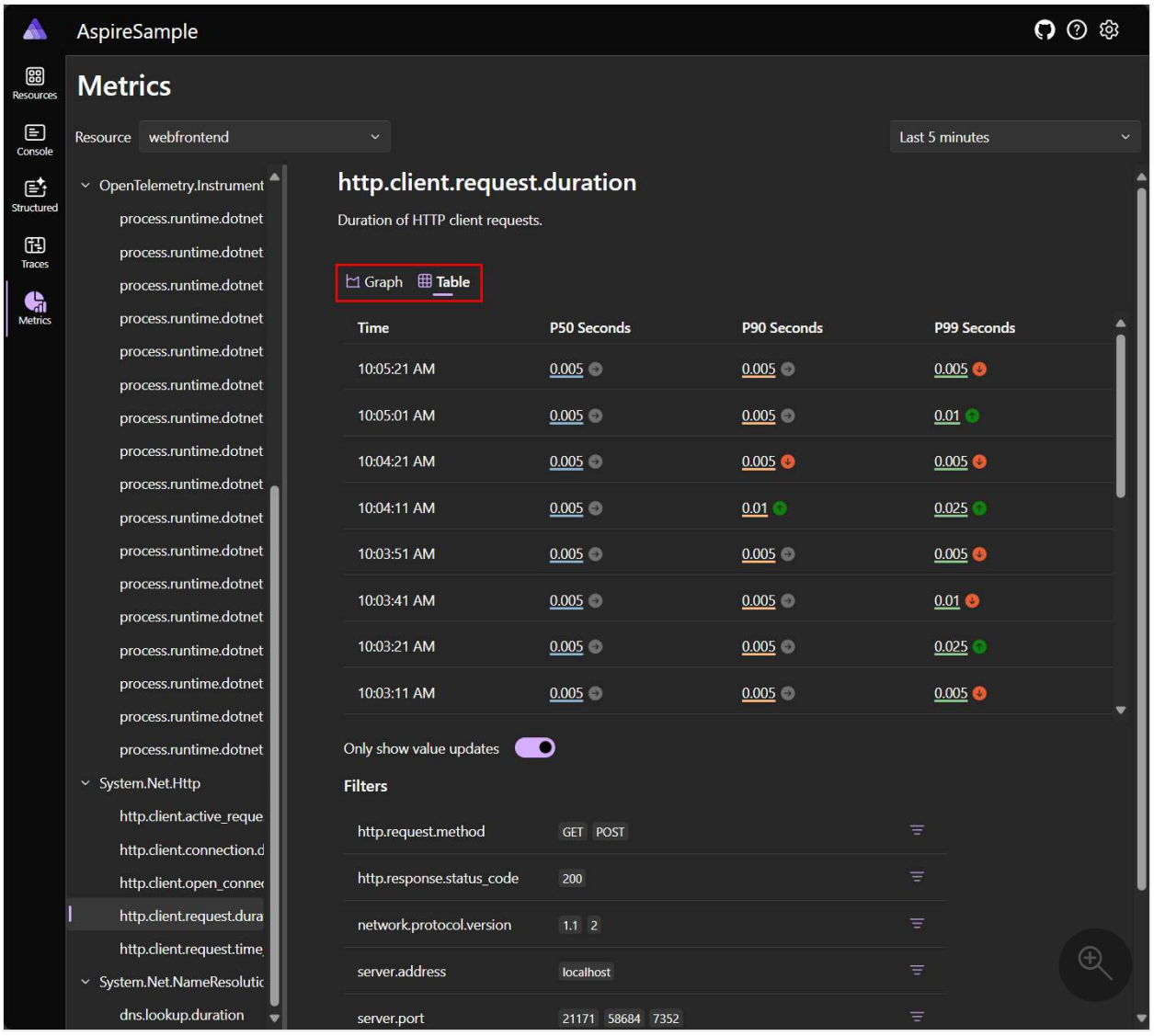
Navigate to the **Metrics** page to view the metrics for your app. .NET Aspire automatically configures metrics for the different projects in your app. Metrics are a way to measure the health of your application and can be used to monitor the performance of your app over time.

Each metric-publishing project in your app has its own metrics. The metrics page displays a selection pane for each top-level meter and the corresponding instruments that you can select to view the metric.

Consider the following example screenshot of the metrics page, with the `webfrontend` project selected and the `System.Net.Http` meter's `http.client.request.duration` metric selected:

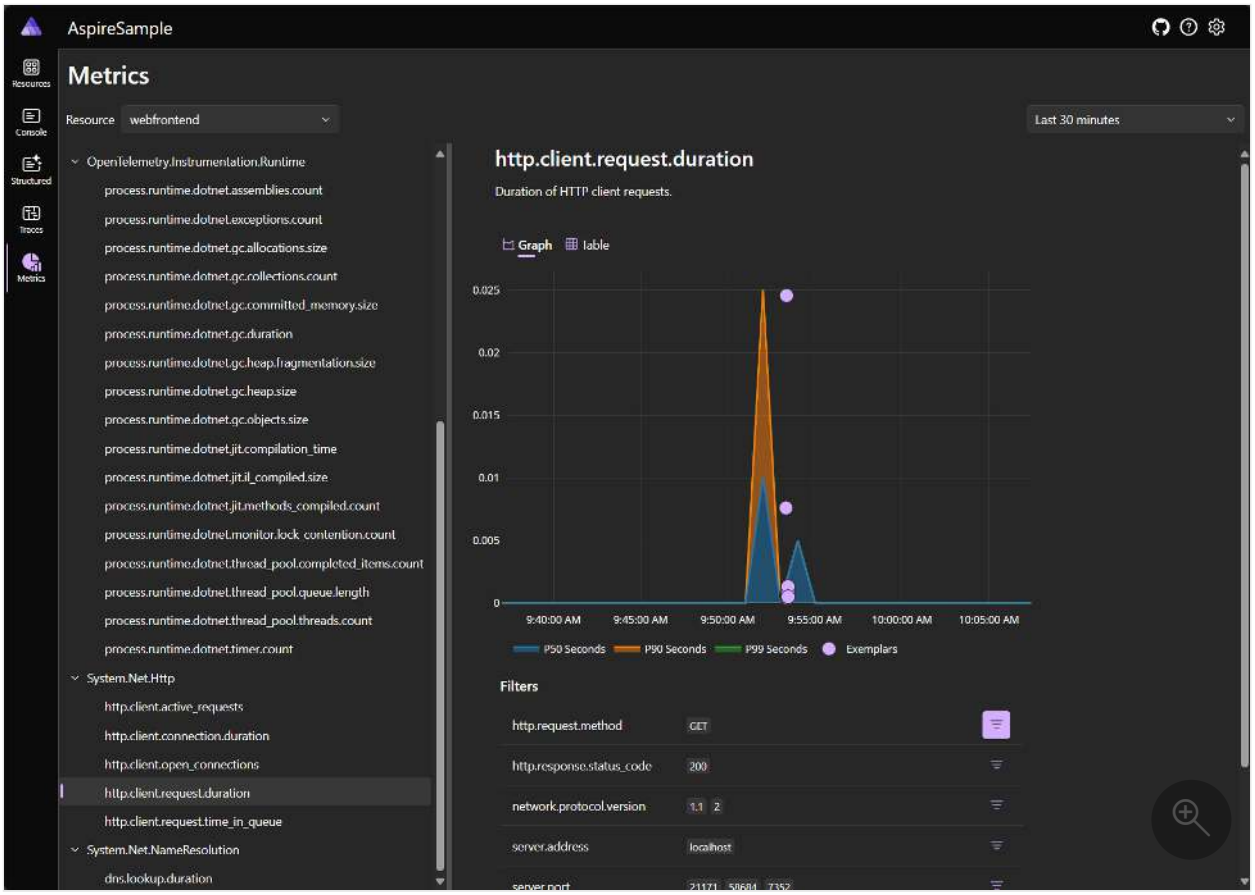


In addition to the metrics chart, the metrics page includes an option to view the data as a table instead. Consider the following screenshot of the metrics page with the table view selected:

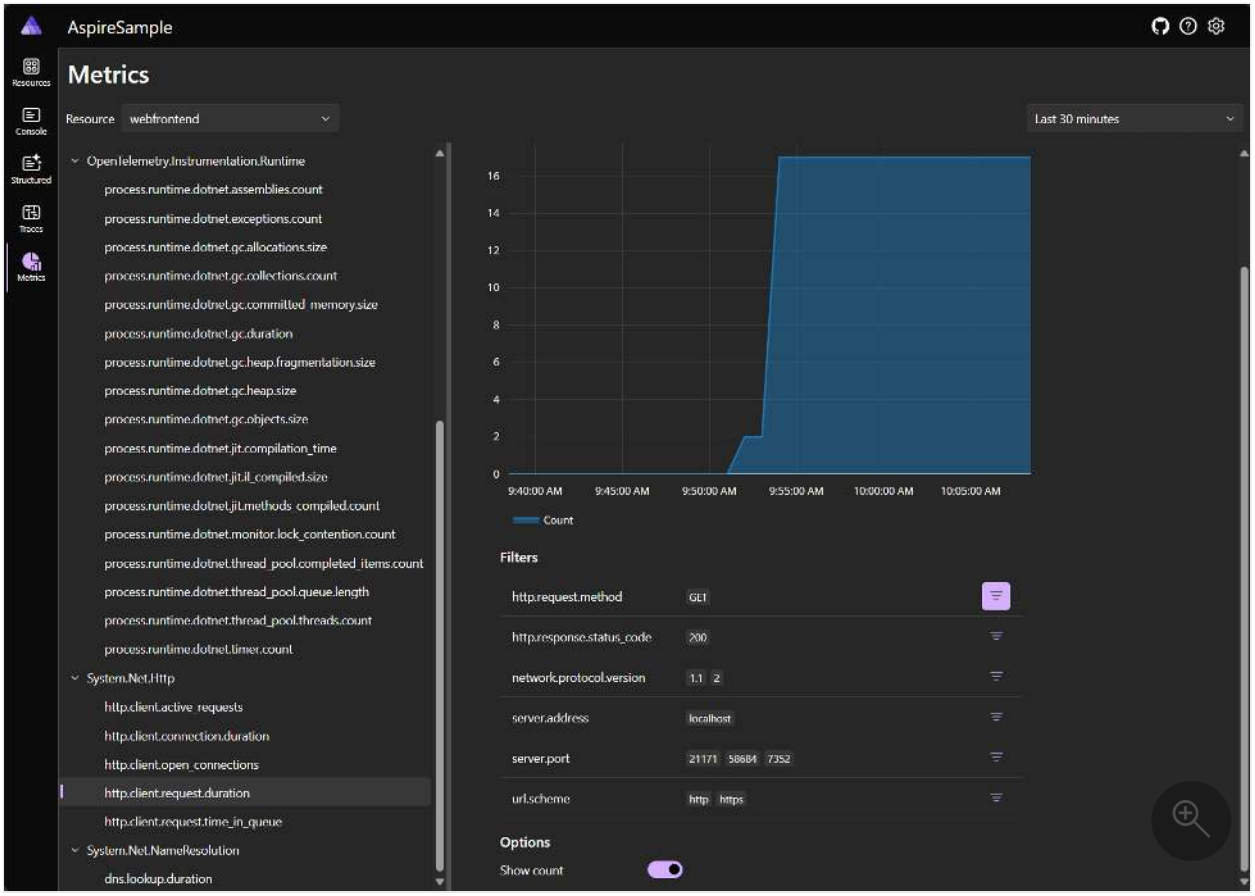


Under the chart, there's a list of filters you can apply to focus on the data that interests you. For example, in the following screenshot, the **http.request.method** field is filtered

to show only **GET** requests:



You can also choose to select the count of the displayed metric on the vertical access, instead of its values:



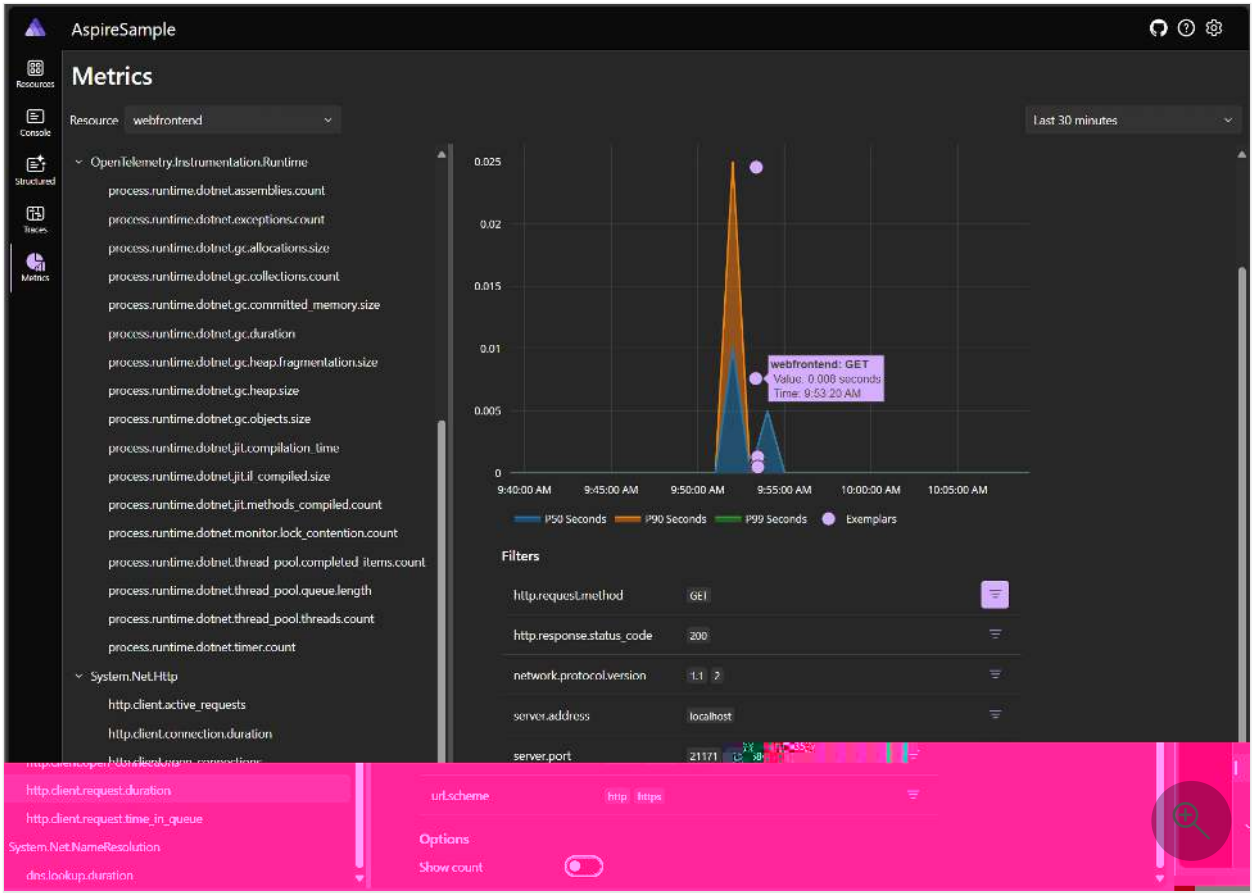
For more information about metrics, see Built-in Metrics in .NET.

Exemplars

The .NET Aspire dashboard supports and displays OpenTelemetry *Exemplars*. An *exemplar* links a metric data point to the operation that recorded it, serving as a bridge between metrics and traces.

Exemplars are useful because they provide additional context about why a specific metric value was recorded. For example, if you notice a spike in latency in the `http.client.request.duration` metric, an exemplar could point to a specific trace or span that caused the spike, helping you understand the root cause.

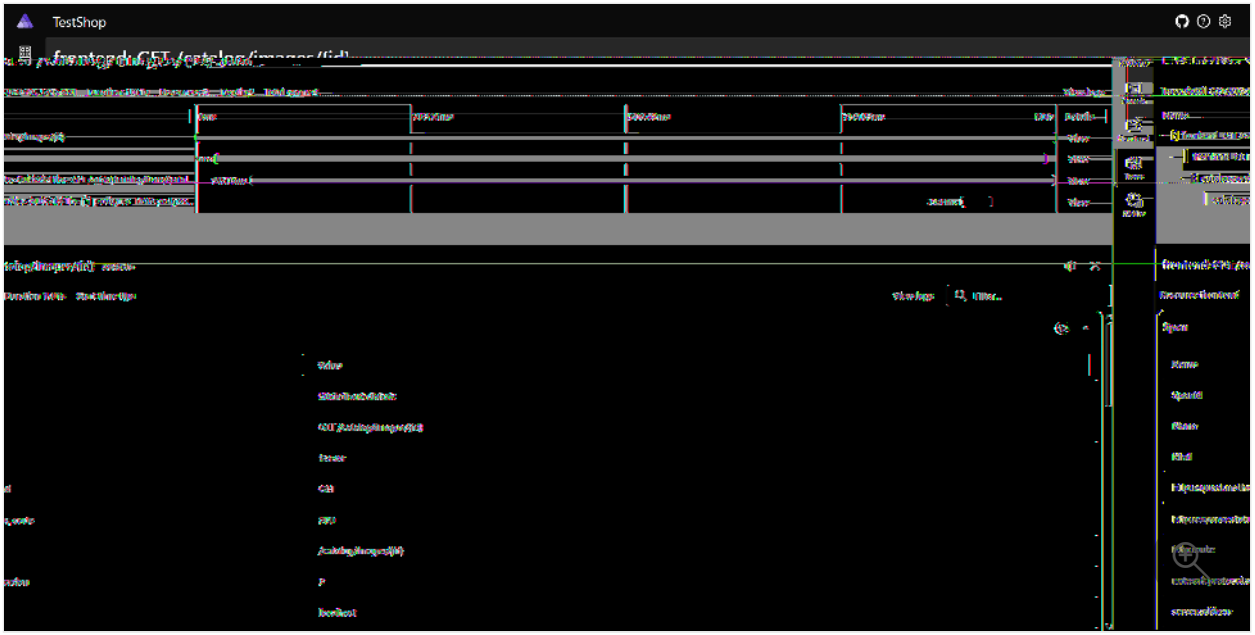
Exemplars are displayed in the metrics chart as a small round dot next to the data point. When you hover over the indicator, a tooltip displays the exemplar details as shown in the following screenshot:



The preceding screenshot shows the exemplar details for the `http.client.request.duration` metric. The exemplar details include the:

- Resource name.
- Operation performed, in this case an HTTP GET to the `/catalog/images/{id}`.
- Corresponding value and the time stamp.

Selecting the exemplar indicator opens the trace details page, where you can view the trace associated, for example consider the following screenshot:

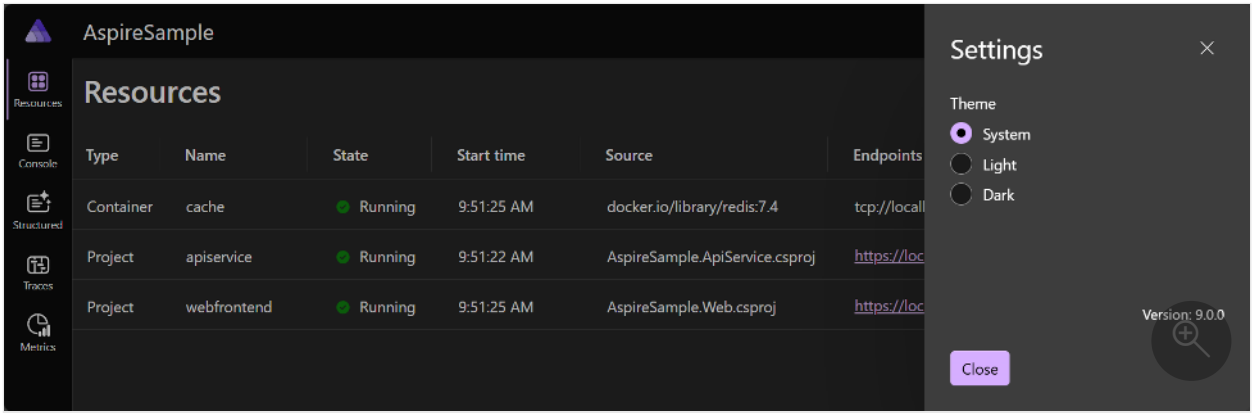


For more information, see OpenTelemetry Docs: Exemplars.

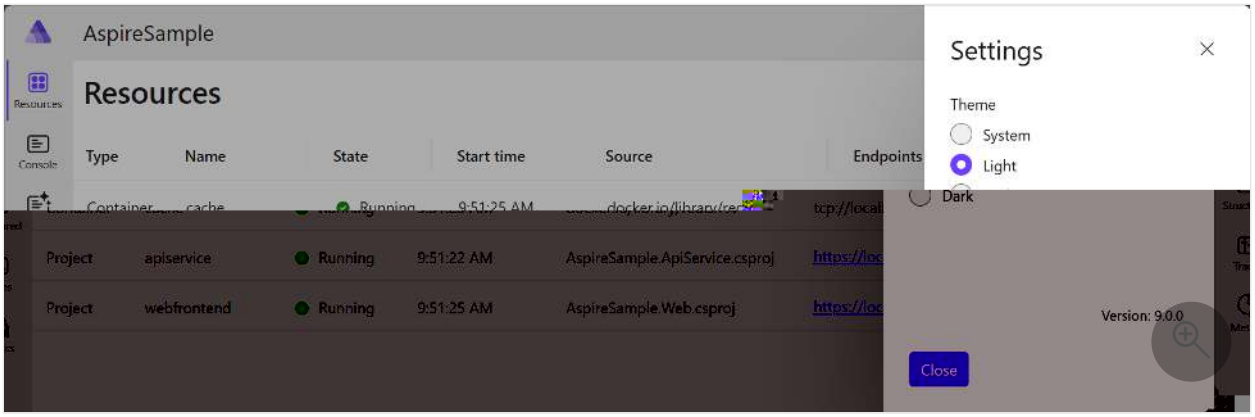
Theme selection

By default, the theme is set to follow the System theme, which means the dashboard uses the same theme as your operating system. You can also select the **Light** or **Dark** theme to override the system theme. Theme selections are persisted.

The following screenshot shows the theme selection dialog, with the default System theme selected:

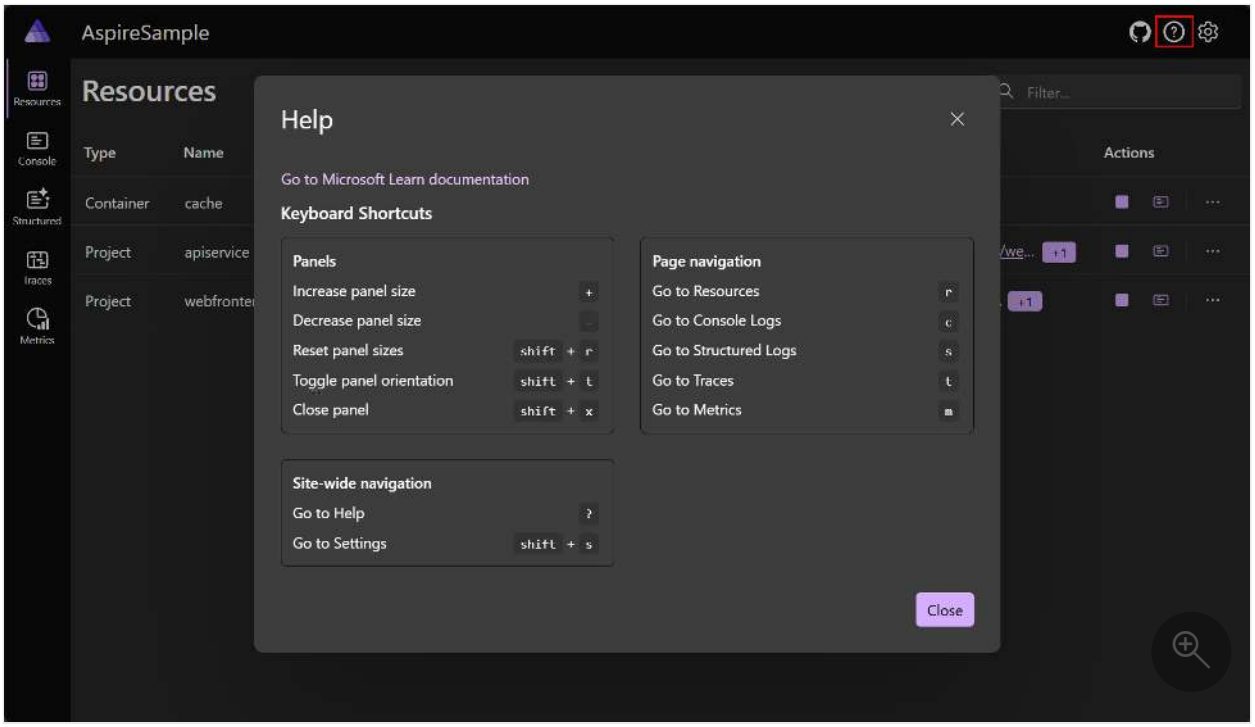


If you prefer the Light theme, you can select it from the theme selection dialog:



Dashboard shortcuts

The .NET Aspire dashboard provides various shortcuts to *help* you navigate and control different parts of the dashboard. To display the keyboard shortcuts, press `Shift` + `?`, or select the question mark icon in the top-right corner of the dashboard:



The following shortcuts are available:

Panels:

- `+`: Increase panel size.
- `-`: Decrease panel size.
- `Shift` + `r`: Reset panel size.
- `Shift` + `t`: Toggle panel orientation.
- `Shift` + `x`: Close panel.

Page navigation:

- `r`: Go to **R**esources.
- `c`: Go to **C**onsole Logs.
- `s`: Go to **S**tructured Logs.
- `t`: Go to **T**races.
- `m`: Go to **M**etrics.

Site-wide navigation:

- `?`: Got to **Help**.
- `Shift` + `s`: Go to **S**ettings.

Next steps

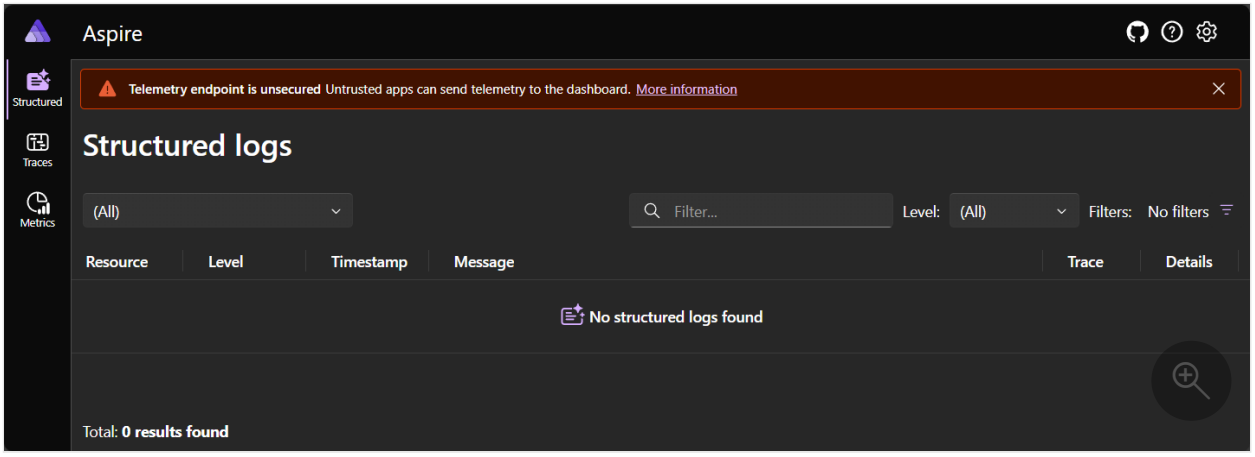
Standalone .NET Aspire dashboard

Standalone .NET Aspire dashboard

Article • 10/29/2024

The .NET Aspire dashboard provides a great UI for viewing telemetry. The dashboard:

- Ships as a container image that can be used with any OpenTelemetry enabled app.
- Can be used standalone, without the rest of .NET Aspire.



Start the dashboard

The dashboard is started using the Docker command line.

Bash

Bash

```
docker run --rm -it -d \
    -p 18888:18888 \
    -p 4317:18889 \
    --name aspire-dashboard \
    mcr.microsoft.com/dotnet/aspire-dashboard:9.0
```

The preceding Docker command:

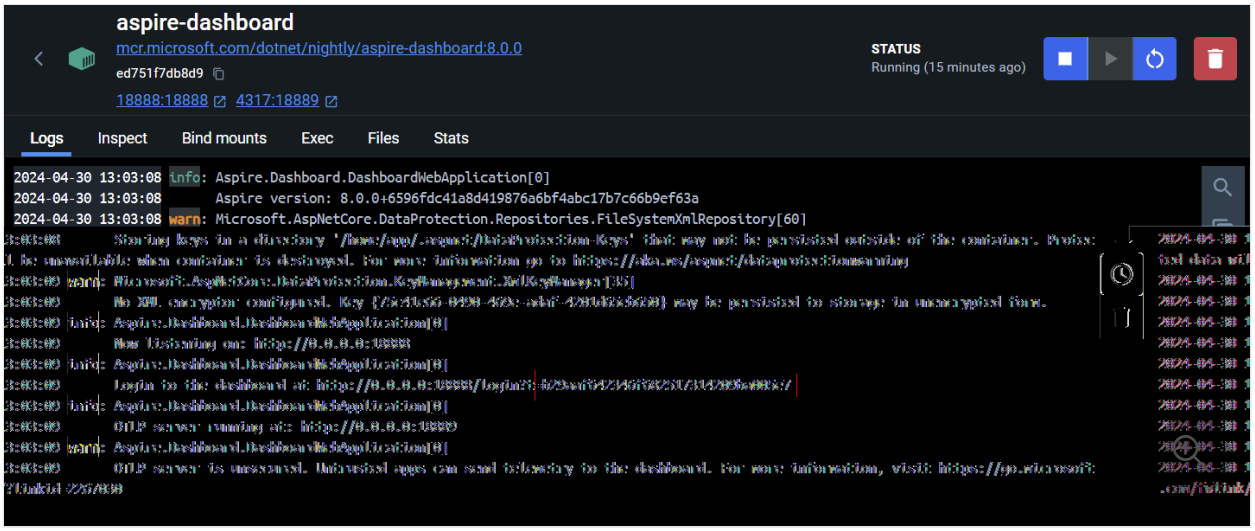
- Starts a container from the `mcr.microsoft.com/dotnet/aspire-dashboard:9.0` image.
- The container expose two ports:
  - Mapping the dashboard's OTLP port `18889` to the host's port `4317`. Port `4317` receives OpenTelemetry data from apps. Apps send data using OpenTelemetry Protocol (OTLP).

- Mapping the dashboard's port `18888` to the host's port `18888`. Port `18888` has the dashboard UI. Navigate to `http://localhost:18888` in the browser to view the dashboard.

Login to the dashboard

Data displayed in the dashboard can be sensitive. By default, the dashboard is secured with authentication that requires a token to login.

When the dashboard is run from a standalone container, the login token is printed to the container logs. After copying the highlighted token into the login page, select the *Login* button.



> **Tip**
>
> To avoid the login, you can disable the authentication requirement by setting the `DOTNET_DASHBOARD_UNSECURED_ALLOW_ANONYMOUS` environment variable to `true`. Additional configuration is available, see **Dashboard configuration**.

For more information about logging into the dashboard, see Dashboard authentication.

Explore the dashboard

The dashboard offers a UI for viewing telemetry. Refer to the documentation to explore the telemetry functionality:

- Structured logs page
- Traces page
- Metrics page

Although there is no restriction on where the dashboard is run, the dashboard is designed as a development and short-term diagnostic tool. The dashboard persists telemetry in-memory which creates some limitations:

- Telemetry is automatically removed if telemetry limits are exceeded.
- No telemetry is persisted when the dashboard is restarted.

The dashboard also has functionality for viewing .NET Aspire resources. The dashboard resource features are disabled when it is run in standalone mode. To enable the resources UI, add configuration for a resource service.

Send telemetry to the dashboard

Apps send telemetry to the dashboard using OpenTelemetry Protocol (OTLP). The dashboard must expose a port for receiving OpenTelemetry data, and apps are configured to send data to that address.

A Docker command was shown earlier to start the dashboard. It configured the container to receive OpenTelemetry data on port `4317`. The OTLP endpoint's full address is `http://localhost:4317`.

Configure OpenTelemetry SDK

Apps collect and send telemetry using their language's OpenTelemetry SDK.

Important OpenTelemetry SDK options to configure:

- OTLP endpoint, which should match the dashboard's configuration, for example, `http://localhost:4317`.
- OTLP protocol, with the dashboard currently supporting only the OTLP/gRPC protocol. Configure applications to use the `grpc` protocol.

To configure applications:

- Use the OpenTelemetry SDK APIs within the application, or
- Start the app with known environment variables:
  - `OTEL_EXPORTER_OTLP_PROTOCOL` with a value of `grpc`.
  - `OTEL_EXPORTER_OTLP_ENDPOINT` with a value of `http://localhost:4317`.

Sample

For a sample of using the standalone dashboard, see the Standalone .NET Aspire dashboard sample app.

Next steps

Tutorial: Use the .NET Aspire dashboard with Python apps

Article • 10/29/2024

The .NET Aspire dashboard provides a great user experience for viewing telemetry, and is available as a standalone container image that can be used with any OpenTelemetry-enabled app. In this article, you'll learn how to:

- ✓ Start the .NET Aspire dashboard in standalone mode.
- ✓ Use the .NET Aspire dashboard with a Python app.

Prerequisites

To complete this tutorial, you need the following:

- Docker or Podman.
  - You can use an alternative container runtime, but the commands in this article are for Docker.
- Python 3.9 or higher locally installed.
- A sample application.

Sample application

This tutorial can be completed using either Flask, Django, or FastAPI. A sample application in each framework is provided to help you follow along with this tutorial. Download or clone the sample application to your local workstation.

Flask

Console

```
git clone https://github.com/Azure-Samples/msdocs-python-flask-webapp-quickstart
```

To run the application locally:

Flask

1. Go to the application folder:

```
Console

cd msdocs-python-flask-webapp-quickstart
```

2. Create a virtual environment for the app:

Windows

```
PowerShell

py -m venv .venv
.\.venv\Scripts\Activate.ps1
```

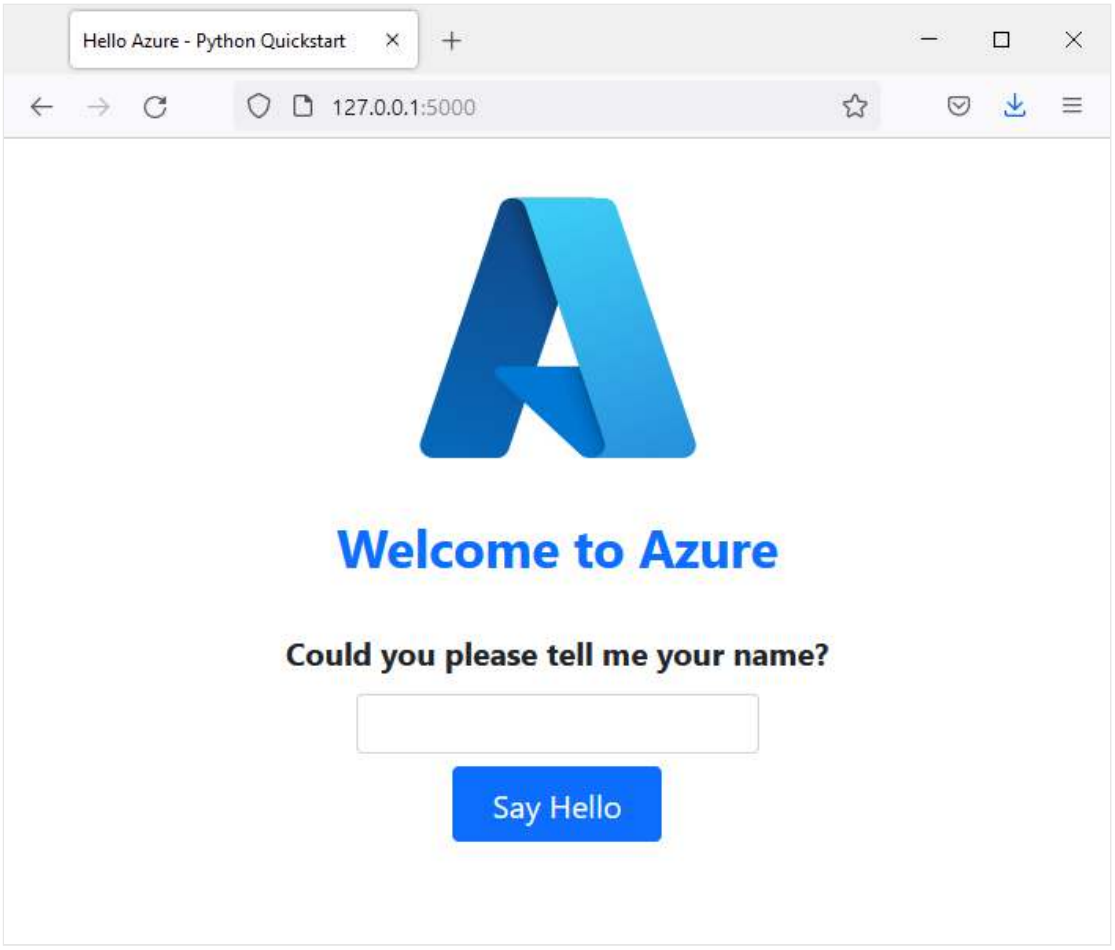
3. Install the dependencies:

```
Console

pip install -r requirements.txt
```

4. Run the app:

```
Console

flask run
```

5. Browse to the sample application at `http://localhost:5000` in a web browser.



Adding OpenTelemetry

To use the .NET Aspire dashboard with your Python app, you need to install the OpenTelemetry SDK and exporter. The OpenTelemetry SDK provides the API for instrumenting your application, and the exporter sends telemetry data to the .NET Aspire dashboard.

1. Install the OpenTelemetry SDK and exporter:

   Console

   ```
   pip install opentelemetry-api opentelemetry-sdk opentelemetry-exporter-otlp-proto-grpc
   ```

2. Add a new file to your application called `otlp_tracing.py` and add the following code:

   Python

   ```
   import logging
   from opentelemetry import metrics, trace
   ```

```
from opentelemetry._logs import set_logger_provider
from opentelemetry.exporter.otlp.proto.grpc._log_exporter import (
    OTLPLogExporter,
)
from opentelemetry.exporter.otlp.proto.grpc.metric_exporter import
OTLPMetricExporter
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import
OTLPSpanExporter
from opentelemetry.sdk._logs import LoggerProvider, LoggingHandler
from opentelemetry.sdk._logs.export import BatchLogRecordProcessor
from opentelemetry.sdk.metrics import MeterProvider
from opentelemetry.sdk.metrics.export import
PeriodicExportingMetricReader
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor

def configure_oltp_grpc_tracing(
    endpoint: str = None
) -> trace.Tracer:
    # Configure Tracing
    traceProvider = TracerProvider()
    processor = BatchSpanProcessor(OTLPSpanExporter(endpoint=endpoint))
    traceProvider.add_span_processor(processor)
    trace.set_tracer_provider(traceProvider)

    # Configure Metrics
    reader =
PeriodicExportingMetricReader(OTLPMetricExporter(endpoint=endpoint))
    meterProvider = MeterProvider(metric_readers=[reader])
    metrics.set_meter_provider(meterProvider)

    # Configure Logging
    logger_provider = LoggerProvider()
    set_logger_provider(logger_provider)

    exporter = OTLPLogExporter(endpoint=endpoint)

logger_provider.add_log_record_processor(BatchLogRecordProcessor(export
er))
    handler = LoggingHandler(level=logging.NOTSET,
logger_provider=logger_provider)
    handler.setFormatter(logging.Formatter("Python: %(message)s"))

    # Attach OTLP handler to root logger
    logging.getLogger().addHandler(handler)

    tracer = trace.get_tracer(__name__)
    return tracer
```

3. Update your application (`app.py` for Flask, `main.py` for FastAPI) to include the imports and call the `configure_oltp_grpc_tracing` function:

Python

```python
import logging
from otlp_tracing import configure_otel_otlp

logging.basicConfig(level=logging.INFO)
tracer = configure_otel_otlp()
logger = logging.getLogger(__name__)
```

4. Replace the `print` calls with `logger.info` calls in your application.

5. Restart your application.

Framework Specific Instrumentation

This instrumentation has only focused on adding OpenTelemetry to our code. For more detailed instrumentation, you can use the OpenTelemetry Instrumentation packages for the specific frameworks that you are using.

Flask

1. Install the Flask instrumentation package:

   Console

   ```
   pip install opentelemetry-instrumentation-flask
   ```

2. Add the following code to your application:

   Python

   ```python
   from opentelemetry.instrumentation.flask import FlaskInstrumentor

   # add this line after configure_otel_otlp() call
   FlaskInstrumentor().instrument()
   ```

Start the Aspire dashboard

To start the Aspire dashboard in standalone mode, run the following Docker command:

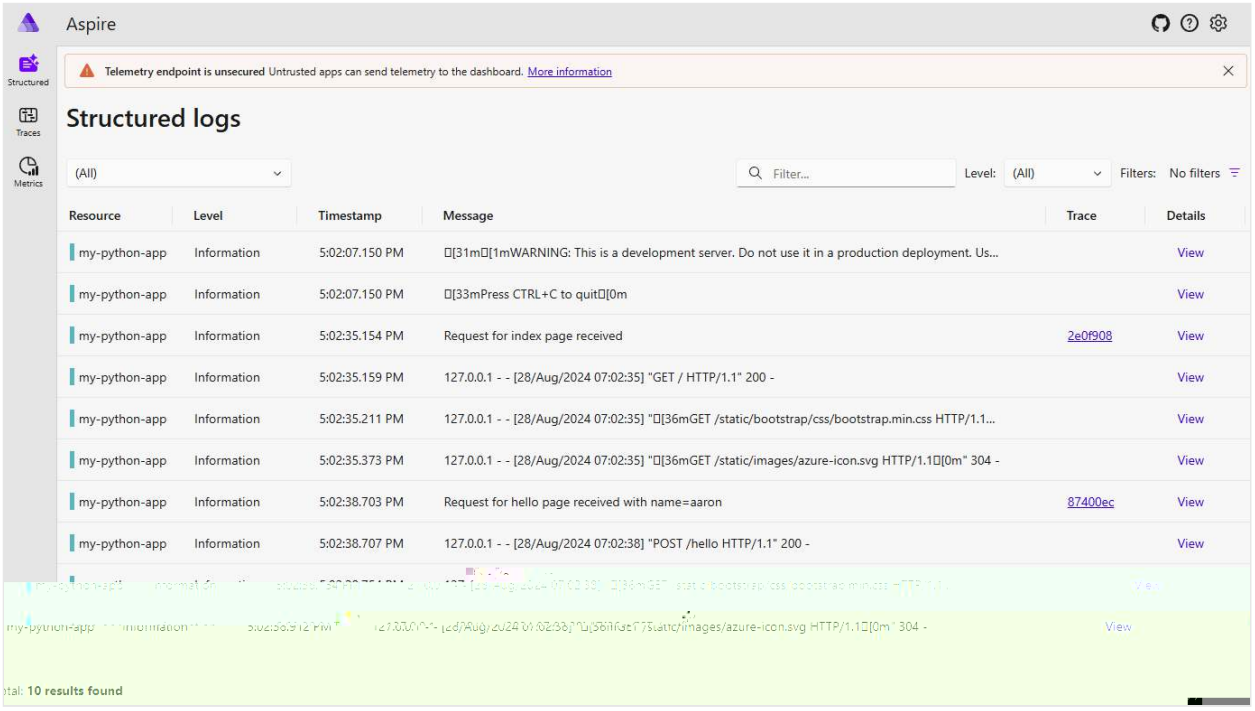
Bash

```
docker run --rm -it -p 18888:18888 -p 4317:18889 --name aspire-dashboard \
    mcr.microsoft.com/dotnet/aspire-dashboard:9.0
```

In the Docker logs, the endpoint and key for the dashboard are displayed. Copy the key and navigate to `http://localhost:18888` in a web browser. Enter the key to log in to the dashboard.

View Structured Logs

Navigate around the Python application, and you'll see structured logs in the Aspire dashboard. The structured logs page displays logs from your application, and you can filter and search the logs.



Next steps

You have successfully used the .NET Aspire dashboard with a Python application. To learn more about the .NET Aspire dashboard, see the Aspire dashboard overview and how to orchestrate a Python application with the .NET Aspire app host.

Dashboard configuration

Article • 02/13/2025

The dashboard is configured when it starts up. Configuration includes frontend and OpenTelemetry Protocol (OTLP) addresses, the resource service endpoint, authentication, telemetry limits, and more.

When the dashboard is launched with the .NET Aspire app host project, it's automatically configured to display the app's resources and telemetry. Configuration is provided when launching the dashboard in standalone mode.

There are many ways to provide configuration:

- Command line arguments.
- Environment variables. The `:` delimiter should be replaced with double underscore (`__`) in environment variable names.
- Optional JSON configuration file. The `DOTNET_DASHBOARD_CONFIG_FILE_PATH` setting can be used to specify a JSON configuration file.

Consider the following example, which shows how to configure the dashboard when started from a Docker container:

Bash

Bash

```bash
docker run --rm -it -p 18888:18888 -p 4317:18889 -d --name aspire-dashboard \
    -e DASHBOARD__TELEMETRYLIMITS__MAXLOGCOUNT='1000' \
    -e DASHBOARD__TELEMETRYLIMITS__MAXTRACECOUNT='1000' \
    -e DASHBOARD__TELEMETRYLIMITS__MAXMETRICSCOUNT='1000' \
    mcr.microsoft.com/dotnet/aspire-dashboard:9.0
```

Alternatively, these same values could be configured using a JSON configuration file that is specified using `DOTNET_DASHBOARD_CONFIG_FILE_PATH`:

JSON

```json
{
  "Dashboard": {
    "TelemetryLimits": {
      "MaxLogCount": 1000,
      "MaxTraceCount": 1000,
      "MaxMetricsCount": 1000
    }
  }
}
```

> **Important**
>
> The dashboard displays information about resources, including their configuration, console logs and in-depth telemetry.
>
> Data displayed in the dashboard can be sensitive. For example, secrets in environment variables, and sensitive runtime data in telemetry. Care should be taken to configure the dashboard to secure access.

For more information, see dashboard security.

> **Note**
>
> Configuration described on this page is for the standalone dashboard. To configure an .NET Aspire app host project, see App host configuration.

Common configuration

Expand table

Option	Default value	Description
`ASPNETCORE_URLS`	`http://localhost:18888`	One or more HTTP endpoints through which the dashboard frontend is served. The frontend endpoint is used to view the dashboard in a browser. When the dashboard is launched by the .NET Aspire app host this address is secured with HTTPS. Securing the dashboard with HTTPS is recommended.
`DOTNET_DASHBOARD_OTLP_ENDPOINT_URL`	`http://localhost:18889`	The OTLP/gRPC endpoint. This endpoint hosts an OTLP service and receives telemetry using gRPC. When the dashboard is launched by the .NET Aspire app host this address is secured with HTTPS. Securing the dashboard with HTTPS is recommended.
`DOTNET_DASHBOARD_OTLP_HTTP_ENDPOINT_URL`	`http://localhost:18890`	The OTLP/HTTP endpoint. This endpoint hosts an OTLP service and receives telemetry using Protobuf over HTTP. When the dashboard is launched by the .NET Aspire app host the OTLP/HTTP endpoint isn't configured by default. To configure an OTLP/HTTP endpoint with the app host, set an `DOTNET_DASHBOARD_OTLP_HTTP_ENDPOINT_URL` env var value in *launchSettings.json*. Securing the dashboard with HTTPS is recommended.
`DOTNET_DASHBOARD_UNSECURED_ALLOW_ANONYMOUS`	`false`	Configures the dashboard to not use authentication and accepts anonymous access. This setting is a shortcut to configuring `Dashboard:Frontend:AuthMode` and `Dashboard:Otlp:AuthMode` to `Unsecured`.
`DOTNET_DASHBOARD_CONFIG_FILE_PATH`	`null`	The path for a JSON configuration file. If the dashboard is being run in a Docker container, then this is the path to the configuration file in a mounted volume. This value is optional.
`DOTNET_DASHBOARD_FILE_CONFIG_DIRECTORY`	`null`	The directory where the dashboard looks for key-per-file configuration. This value is optional.
`DOTNET_RESOURCE_SERVICE_ENDPOINT_URL`	`null`	The gRPC endpoint to which the dashboard connects for its data. If this value is unspecified, the dashboard shows telemetry data but no

Option	Default value	Description
		resource list or console logs. This setting is a shortcut to `Dashboard:ResourceServiceClient:Url`.

Frontend authentication

The dashboard frontend endpoint authentication is configured with `Dashboard:Frontend:AuthMode`. The frontend can be secured with OpenID Connect (OIDC) or browser token authentication.

Browser token authentication works by the frontend asking for a token. The token can either be entered in the UI or provided as a query string value to the login page. For example, `https://localhost:1234/login?t=TheToken`. When the token is successfully authenticated an auth cookie is persisted to the browser, and the browser is redirected to the app.

Expand table

Option	Default value	Description
`Dashboard:Frontend:AuthMode`	`BrowserToken`	Can be set to `BrowserToken`, `OpenIdConnect` or `Unsecured`. `Unsecured` should only be used during local development. It's not recommended when hosting the dashboard publicly or in other settings.
`Dashboard:Frontend:BrowserToken`	`null`	Specifies the browser token. If the browser token isn't specified, then the dashboard generates one. Tooling that wants to automate logging in with browser token authentication can specify a token and open a browser with the token in the query string. A new token should be generated each time the dashboard is launched.
`Dashboard:Frontend:OpenIdConnect:NameClaimType`	`name`	Specifies one or more claim types that should be used to display the authenticated user's full name. Can be a single claim type or a comma-delimited list of claim types.
`Dashboard:Frontend:OpenIdConnect:UsernameClaimType`	`preferred_username`	Specifies one or more claim types that should be used to display the authenticated user's username. Can be a single claim type or a comma-delimited list of claim types.
`Dashboard:Frontend:OpenIdConnect:RequiredClaimType`	`null`	Specifies the claim that must be present for authorized users. Authorization fails without this claim. This value is optional.
`Dashboard:Frontend:OpenIdConnect:RequiredClaimValue`	`null`	Specifies the value of the required claim. Only used if `Dashboard:Frontend:OpenIdConnect:RequireClaimType` is also specified. This value is optional.
`Authentication:Schemes:OpenIdConnect:Authority`	`null`	URL to the identity provider (IdP).
`Authentication:Schemes:OpenIdConnect:ClientId`	`null`	Identity of the relying party (RP).
`Authentication:Schemes:OpenIdConnect:ClientSecret`	`null`	A secret that only the real RP would know.
Other properties of OpenIdConnectOptions	`null`	Values inside configuration section `Authentication:Schemes:OpenIdConnect:*` are bound

Option	Default value	Description
		to `OpenIdConnectOptions`, such as `Scope`.

> [!NOTE]
> Additional configuration may be required when using `OpenIdConnect` as authentication mode behind a reverse-proxy that terminates SSL. Check if you need `ASPNETCORE_FORWARDEDHEADERS_ENABLED` to be set to `true`.
>
> For more information, see Configure ASP.NET Core to work with proxy servers and load balancers.

OTLP authentication

The OTLP endpoint authentication is configured with `Dashboard:Otlp:AuthMode`. The OTLP endpoint can be secured with an API key or client certificate authentication.

API key authentication works by requiring each OTLP request to have a valid `x-otlp-api-key` header value. It must match either the primary or secondary key.

Expand table

Option	Default value	Description
`Dashboard:Otlp:AuthMode`	`Unsecured`	Can be set to `ApiKey`, `Certificate` or `Unsecured`. `Unsecured` should only be used during local development. It's not recommended when hosting the dashboard publicly or in other settings.
`Dashboard:Otlp:PrimaryApiKey`	`null`	Specifies the primary API key. The API key can be any text, but a value with at least 128 bits of entropy is recommended. This value is required if auth mode is API key.
`Dashboard:Otlp:SecondaryApiKey`	`null`	Specifies the secondary API key. The API key can be any text, but a value with at least 128 bits of entropy is recommended. This value is optional. If a second API key is specified, then the incoming `x-otlp-api-key` header value can match either the primary or secondary key.

OTLP CORS

Cross-origin resource sharing (CORS) can be configured to allow browser apps to send telemetry to the dashboard.

By default, browser apps are restricted from making cross domain API calls. This impacts sending telemetry to the dashboard because the dashboard and the browser app are always on different domains. To configure CORS, use the `Dashboard:Otlp:Cors` section and specify the allowed origins and headers:

JSON

```
{
  "Dashboard": {
    "Otlp": {
      "Cors": {
        "AllowedOrigins": "http://localhost:5000,https://localhost:5001"
```

```
      }
    }
  }
}
```

Consider the following configuration options:

Expand table

Option	Default value	Description
`Dashboard:Otlp:Cors:AllowedOrigins`	`null`	Specifies the allowed origins for CORS. It's a comma-delimited string and can include the `*` wildcard to allow any domain. This option is optional and can be set using the `DASHBOARD__OTLP__CORS__ALLOWEDORIGINS` environment variable.
`Dashboard:Otlp:Cors:AllowedHeaders`	`null`	A comma-delimited string representing the allowed headers for CORS. This setting is optional and can be set using the `DASHBOARD__OTLP__CORS__ALLOWEDHEADERS` environment variable.

> **Note**
>
> The dashboard only supports the `POST` method for sending telemetry and doesn't allow configuration of the *allowed methods* (`Access-Control-Allow-Methods`) for CORS.

Resources

The dashboard connects to a resource service to load and display resource information. The client is configured in the dashboard for how to connect to the service.

The resource service client authentication is configured with `Dashboard:ResourceServiceClient:AuthMode`. The client can be configured to support API key or client certificate authentication.

Expand table

Option	Default value	Description
`Dashboard:ResourceServiceClient:Url`	`null`	The gRPC endpoint to which the dashboard connects for its data. If this value is unspecified, the dashboard shows telemetry data but no resource list or console logs.
`Dashboard:ResourceServiceClient:AuthMode`	`null`	Can be set to `ApiKey`, `Certificate` or `Unsecured`. `Unsecured` should only be used during local development. It's not recommended when hosting the dashboard publicly or in other settings. This value is required if a resource service URL is specified.
`Dashboard:ResourceServiceClient:ApiKey`	`null`	The API to send to the resource service in the `x-resource-service-api-key` header.

Option	Default value	Description
		This value is required if auth mode is API key.
`Dashboard:ResourceServiceClient:ClientCertificate:Source`	`null`	Can be set to `File` or `KeyStore`. This value is required if auth mode is client certificate.
`Dashboard:ResourceServiceClient:ClientCertificate:FilePath`	`null`	The certificate file path. This value is required if source is `File`.
`Dashboard:ResourceServiceClient:ClientCertificate:Password`	`null`	The password for the certificate file. This value is optional.
`Dashboard:ResourceServiceClient:ClientCertificate:Subject`	`null`	The certificate subject. This value is required if source is `KeyStore`.
`Dashboard:ResourceServiceClient:ClientCertificate:Store`	`My`	The certificate StoreName.
`Dashboard:ResourceServiceClient:ClientCertificate:Location`	`CurrentUser`	The certificate StoreLocation.

Telemetry limits

Telemetry is stored in memory. To avoid excessive memory usage, the dashboard has limits on the count and size of stored telemetry. When a count limit is reached, new telemetry is added, and the oldest telemetry is removed. When a size limit is reached, data is truncated to the limit.

Telemetry limits have different scopes depending upon the telemetry type:

- `MaxLogCount` and `MaxTraceCount` are shared across resources. For example, a `MaxLogCount` value of 5,000 configures the dashboard to store up to 5,000 total log entries for all resources.
- `MaxMetricsCount` is per-resource. For example, a `MaxMetricsCount` value of 10,000 configures the dashboard to store up to 10,000 metrics data points per-resource.

Expand table

Option	Default value	Description
`Dashboard:TelemetryLimits:MaxLogCount`	10,000	The maximum number of log entries. Limit is shared across resources.
`Dashboard:TelemetryLimits:MaxTraceCount`	10,000	The maximum number of log traces. Limit is shared across resources.
`Dashboard:TelemetryLimits:MaxMetricsCount`	50,000	The maximum number of metric data points. Limit is per-resource.
`Dashboard:TelemetryLimits:MaxAttributeCount`	128	The maximum number of attributes on telemetry.
`Dashboard:TelemetryLimits:MaxAttributeLength`	`null`	The maximum length of attributes.
`Dashboard:TelemetryLimits:MaxSpanEventCount`	`null`	The maximum number of events on span attributes.

Other

Expand table

Option	Default value	Description
`Dashboard:ApplicationName`	`Aspire`	The application name to be displayed in the UI. This applies only when no resource service URL is specified. When a resource service exists, the service specifies the application name.

Next steps

Security considerations for running the .NET Aspire dashboard

Enable browser telemetry

Article • 11/11/2024

The .NET Aspire dashboard can be configured to receive telemetry sent from browser apps. This feature is useful for monitoring client-side performance and user interactions. Browser telemetry requires additional dashboard configuration and that the JavaScript OTEL SDK is added to the browser apps.

This article discusses how to enable browser telemetry in the .NET Aspire dashboard.

Dashboard configuration

Browser telemetry requires the dashboard to enable these features:

- OTLP HTTP endpoint. This endpoint is used by the dashboard to receive telemetry from browser apps.
- Cross-origin resource sharing (CORS). CORS allows browser apps to make requests to the dashboard.

OTLP configuration

The .NET Aspire dashboard receives telemetry through OTLP endpoints. HTTP OTLP endpoints and gRPC OTLP endpoints are supported by the dashboard. Browser apps must use HTTP OLTP to send telemetry to the dashboard because browser apps don't support gRPC.

To configure the gPRC or HTTP endpoints, specify the following environment variables:

- `DOTNET_DASHBOARD_OTLP_ENDPOINT_URL`: The gRPC endpoint to which the dashboard connects for its data.
- `DOTNET_DASHBOARD_OTLP_HTTP_ENDPOINT_URL`: The HTTP endpoint to which the dashboard connects for its data.

Configuration of the HTTP OTLP endpoint depends on whether the dashboard is started by the app host or is run standalone.

Configure OTLP HTTP with app host

If the dashboard and your app are started by the app host, the dashboard OTLP endpoints are configured in the app host's *launchSettings.json* file.

Consider the following example JSON file:

JSON

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:15887;http://localhost:15888",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "DOTNET_ENVIRONMENT": "Development",
        "DOTNET_DASHBOARD_OTLP_HTTP_ENDPOINT_URL":
"https://localhost:16175",
        "DOTNET_RESOURCE_SERVICE_ENDPOINT_URL": "https://localhost:17037",
        "DOTNET_ASPIRE_SHOW_DASHBOARD_RESOURCES": "true"
      }
    },
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:15888",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "DOTNET_ENVIRONMENT": "Development",
        "DOTNET_DASHBOARD_OTLP_HTTP_ENDPOINT_URL": "http://localhost:16175",
        "DOTNET_RESOURCE_SERVICE_ENDPOINT_URL": "http://localhost:17037",
        "DOTNET_ASPIRE_SHOW_DASHBOARD_RESOURCES": "true",
        "ASPIRE_ALLOW_UNSECURED_TRANSPORT": "true"
      }
    },
    "generate-manifest": {
      "commandName": "Project",
      "launchBrowser": true,
      "dotnetRunMessages": true,
      "commandLineArgs": "--publisher manifest --output-path aspire-
manifest.json",
      "applicationUrl": "http://localhost:15888",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "DOTNET_ENVIRONMENT": "Development"
      }
    }
  }
}
```

The preceding launch settings JSON file configures all profiles to include the `DOTNET_DASHBOARD_OTLP_HTTP_ENDPOINT_URL` environment variable.

Configure OTLP HTTP with standalone dashboard

If the dashboard is used standalone, without the rest of .NET Aspire, the OTLP HTTP endpoint is enabled by default on port `18890`. However, the port must be mapped when the dashboard container is started:

Bash

Bash

```
docker run --rm -it -d \
    -p 18888:18888 \
    -p 4317:18889 \
    -p 4318:18890 \
    --name aspire-dashboard \
    mcr.microsoft.com/dotnet/aspire-dashboard:9.0
```

The preceding command runs the dashboard container and maps gRPC OTLP to port `4317` and HTTP OTLP to port `4318`.

CORS configuration

By default, browser apps are restricted from making cross domain API calls. This impacts sending telemetry to the dashboard because the dashboard and the browser app are always on different domains. Configuring CORS in the .NET Aspire dashboard removes the restriction.

If the dashboard and your app are started by the app host, no CORS configuration is required. .NET Aspire automatically configures the dashboard to allow all resource origins.

If the dashboard is used standlone then CORS must be configured manually. The domain used to view the browser app must be configured as an allowed origin by specifing the `DASHBOARD__OTLP__CORS__ALLOWEDORIGINS` environment variable when the dashboard container is started:

Bash

Bash

```
docker run --rm -it -d \
    -p 18888:18888 \
    -p 4317:18889 \
```

```
    -p 4318:18890 \
    -e DASHBOARD__OTLP__CORS__ALLOWEDORIGINS=https://localhost:8080 \
    --name aspire-dashboard \
    mcr.microsoft.com/dotnet/aspire-dashboard:9.0
```

The preceding command runs the dashboard container and configures `https://localhost:8080` as an allowed origin. That means a browser app that is accessed using `https://localhost:8080` has permission to send the dashboard telemetry.

Multiple origins can be allowed with a comma separated value. Or all origins can be allowed with the `*` wildcard. For example, `DASHBOARD__OTLP__CORS__ALLOWEDORIGINS=*`.

For more information, see .NET Aspire dashboard configuration: OTLP CORS.

OTLP endpoint security

Dashboard OTLP endpoints can be secured with API key authentication. When enabled, HTTP OTLP requests to the dashboard must include the API key as the `x-otlp-api-key` header. By default a new API key is generated each time the dashboard is run.

API key authentication is automatically enabled when the dashboard is run from the app host. Dashboard authentication can be disabled by setting `DOTNET_DASHBOARD_UNSECURED_ALLOW_ANONYMOUS` to `true` in the app host's *launchSettings.json* file.

OTLP endpoints are unsecured by default in the standalone dashboard.

Browser app configuration

A browser app uses the JavaScript OTEL SDK to send telemetry to the dashboard. Successfully sending telemetry to the dashboard requires the SDK to be correctly configured.

OTLP exporter

OTLP exporters must be included in the browser app and configured with the SDK. For example, exporting distributed tracing with OTLP uses the @opentelemetry/exporter-trace-otlp-proto package.

When OTLP is added to the SDK, OTLP options must be specified. OTLP options includes:

- `url`: The address that HTTP OTLP requests are made to. The address should be the dashboard HTTP OTLP endpoint and the path to the OTLP HTTP API. For example, `https://localhost:4318/v1/traces` for the trace OTLP exporter. If the browser app is launched by the app host then the HTTP OTLP endpoint is available from the `OTEL_EXPORTER_OTLP_ENDPOINT` environment variable.
- `headers`: The headers sent with requests. If OTLP endpoint API key authentication is enabled the `x-otlp-api-key` header must be sent with OTLP requests. If the browser app is launched by the app host then the API key is available from the `OTEL_EXPORTER_OTLP_HEADERS` environment variable.

Browser metadata

When a browser app is configured to collect distributed traces, the browser app can set the trace parent a browser's spans using the `meta` element in the HTML. The value of the `name="traceparent"` meta element should correspond to the current trace.

In a .NET app, for example, the trace parent value would likely be assigned from the Activity.Current and passing its Activity.Id value as the `content`. For example, consider the following Razor code:

razor

```razor
<head>
    @if (Activity.Current is { } currentActivity)
    {
        <meta name="traceparent" content="@currentActivity.Id" />
    }
    <!-- Other elements omitted for brevity... -->
</head>
```

The preceding code sets the `traceparent` meta element to the current activity ID.

Example browser telemetry code

The following JavaScript code demonstrates the initialization of the OpenTelemetry JavaScript SDK and the sending of telemetry data to the dashboard:

JavaScript

```javascript
import { ConsoleSpanExporter, SimpleSpanProcessor } from
'@opentelemetry/sdk-trace-base';
import { DocumentLoadInstrumentation } from '@opentelemetry/instrumentation-
document-load';
```

```
import { OTLPTraceExporter } from '@opentelemetry/exporter-trace-otlp-
proto';
import { registerInstrumentations } from '@opentelemetry/instrumentation';
import { Resource } from '@opentelemetry/resources';
import { SemanticResourceAttributes } from '@opentelemetry/semantic-
conventions';
import { WebTracerProvider } from '@opentelemetry/sdk-trace-web';
import { ZoneContextManager } from '@opentelemetry/context-zone';

export function initializeTelemetry(otlpUrl, headers, resourceAttributes) {
    const otlpOptions = {
        url: `${otlpUrl}/v1/traces`,
        headers: parseDelimitedValues(headers)
    };

    const attributes = parseDelimitedValues(resourceAttributes);
    attributes[SemanticResourceAttributes.SERVICE_NAME] = 'browser';

    const provider = new WebTracerProvider({
        resource: new Resource(attributes),
    });
    provider.addSpanProcessor(new SimpleSpanProcessor(new
ConsoleSpanExporter()));
    provider.addSpanProcessor(new SimpleSpanProcessor(new
OTLPTraceExporter(otlpOptions)));

    provider.register({
        // Prefer ZoneContextManager: supports asynchronous operations
        contextManager: new ZoneContextManager(),
    });

    // Registering instrumentations
    registerInstrumentations({
        instrumentations: [new
```

The preceding JavaScript code defines an `initializeTelemetry` function that expects the OTLP endpoint URL, the headers, and the resource attributes. These parameters are

provided by the consuming browser app that pulls them from the environment variables set by the app host. Consider the following Razor code:

razor

```
@using System.Diagnostics
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - BrowserTelemetry</title>
    <link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.cs
s" rel="stylesheet" integrity="sha384-
QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+ALEwIH"
crossorigin="anonymous">
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"
/>

    @if (Activity.Current is { } currentActivity)
    {
        <meta name="traceparent" content="@currentActivity.Id" />
    }
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand" asp-area="" asp-
page="/Index">BrowserTelemetry</a>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>
    @await RenderSectionAsync("Scripts", required: false)
    <script src="scripts/bundle.js"></script>
    @if (Environment.GetEnvironmentVariable("OTEL_EXPORTER_OTLP_ENDPOINT")
is { Length: > 0 } endpointUrl)
    {
        var headers =
Environment.GetEnvironmentVariable("OTEL_EXPORTER_OTLP_HEADERS");
        var attributes =
Environment.GetEnvironmentVariable("OTEL_RESOURCE_ATTRIBUTES");
        <script>
            BrowserTelemetry.initializeTelemetry('@endpointUrl', '@headers',
'@attributes');
        </script>
    }
```

```
</body>
</html>
```

> **Tip**
>
> The bundling and minification of the JavaScript code is beyond the scope of this article.

For the complete working example of how to configure the JavaScript OTEL SDK to send telemetry to the dashboard, see the browser telemetry sample.

See also

- .NET Aspire dashboard configuration
- Standalone .NET Aspire dashboard
- Browser telemetry sample

Security considerations for running the .NET Aspire dashboard

Article • 11/20/2024

The .NET Aspire dashboard offers powerful insights to your apps. The dashboard displays information about resources, including their configuration, console logs and in-depth telemetry.

Data displayed in the dashboard can be sensitive. For example, configuration can include secrets in environment variables, and telemetry can include sensitive runtime data. Care should be taken to secure access to the dashboard.

Scenarios for running the dashboard

The dashboard can be run in different scenarios, such as being automatically starting by .NET Aspire tooling, or as a standalone application that is separate from other .NET Aspire integrations. Steps to secure the dashboard depend on how it's being run.

.NET Aspire tooling

The dashboard is automatically started when an .NET Aspire app host is run. The dashboard is secure by default when run from .NET Aspire tooling:

- Transport is secured with HTTPS. Using HTTPS is configured by default in *launchSettings.json*. The launch profile includes `https` addresses in `applicationUrl` and `DOTNET_DASHBOARD_OTLP_ENDPOINT_URL` values.
- Browser frontend authenticated with a browser token.
- Incoming telemetry authenticated with an API key.

HTTPS in the dashboard uses the ASP.NET Core development certificate. The certificate must be trusted for the dashboard to work correctly. The steps required to trust the development cert are different depending on the machine's operating system:

- Trust the ASP.NET Core HTTPS development certificate on Windows and macOS
- Trust HTTPS certificate on Linux

There are scenarios where you might want to allow an unsecured transport. The dashboard can run without HTTPS from the .NET Aspire app host by configuring the `ASPIRE_ALLOW_UNSECURED_TRANSPORT` setting to `true`. For more information, see Allow unsecured transport in .NET Aspire.

Standalone mode

The dashboard is shipped as a Docker image and can be used without the rest of .NET Aspire. When the dashboard is launched in standalone mode, it defaults to a mix of secure and unsecured settings.

- Browser frontend authenticated with a browser token.
- Incoming telemetry is unsecured. Warnings are displayed in the console and dashboard UI.

The telemetry endpoint accepts incoming OTLP data without authentication. When the endpoint is unsecured, the dashboard is open to receiving telemetry from untrusted apps.

For information about securing the telemetry when running the dashboard in standalone mode, see Securing the telemetry endpoint.

Secure telemetry endpoint

The .NET Aspire dashboard provides a variety of ways to view logs, traces, and metrics for your app. This information enables you to track the behavior and performance of your app and to diagnose any issues that arise. It's important that you can trust this information, and a warning is displayed in the dashboard UI if telemetry isn't secured.

The dashboard collects telemetry through an OTLP (OpenTelemetry protocol) endpoint. Apps send telemetry to this endpoint, and the dashboard stores the external information it receives in memory, which is then accessible via the UI.

To prevent untrusted apps from sending telemetry to .NET Aspire, the OTLP endpoint should be secured. The OTLP endpoint is automatically secured with an API key when the dashboard is started by .NET Aspire tooling. Additional configuration is required for standalone mode.

API key authentication can be enabled on the telemetry endpoint with some additional configuration:

Bash

Bash

```
docker run --rm -it -d -p 18888:18888 -p 4317:18889 --name aspire-
dashboard \
    -e DASHBOARD__OTLP__AUTHMODE='ApiKey' \
```

```
    -e DASHBOARD__OTLP__PRIMARYAPIKEY='{MY_APIKEY}' \
    mcr.microsoft.com/dotnet/aspire-dashboard:9.0
```

The preceding Docker command:

- Starts the .NET Aspire dashboard image and exposes OTLP endpoint as port 4317
- Configures the OTLP endpoint to use `ApiKey` authentication. This requires that incoming telemetry has a valid `x-otlp-api-key` header value.
- Configures the expected API key. `{MY_APIKEY}` in the example value should be replaced with a real API key. The API key can be any text, but a value with at least 128 bits of entropy is recommended.

When API key authentication is configured, the dashboard validates incoming telemetry has a required API key. Apps that send the dashboard telemetry must be configured to send the API key. This can be configured in .NET with `OtlpExporterOptions.Headers`:

C#

```
builder.Services.Configure<OtlpExporterOptions>(
    o => o.Headers = $"x-otlp-api-key={MY_APIKEY}");
```

Other languages have different OpenTelmetry APIs. Passing the OTEL_EXPORTER_OTLP_HEADERS environment variable to apps is a universal way to configure the header.

Memory exhaustion

The dashboard stores external information it receives in memory, such as resource details and telemetry. While the number of resources the dashboard tracks are bounded, there isn't a limit to how much telemetry apps send to the dashboard. Limits must be placed on how much information is stored to prevent the dashboard using an excessive amount of memory and exhausting available memory on the current machine.

Telemetry limits

To help prevent memory exhaustion, the dashboard limits how much telemetry it stores by default. For example, there is a maximum of 10,000 structured log entries per resource. Once the limit is reached, each new log entry received causes an old entry to be removed.

Configuration can customize telemetry limits.

.NET Aspire testing overview

Article • 03/17/2025

.NET Aspire supports automated testing of your application through the Aspire.Hosting.Testing NuGet package. This package provides the DistributedApplicationTestingBuilder class, which is used to create a test host for your application. The testing builder launches your app host project in a background thread and manages its lifecycle, allowing you to control and manipulate the application and its resources through DistributedApplicationTestingBuilder or DistributedApplication instances.

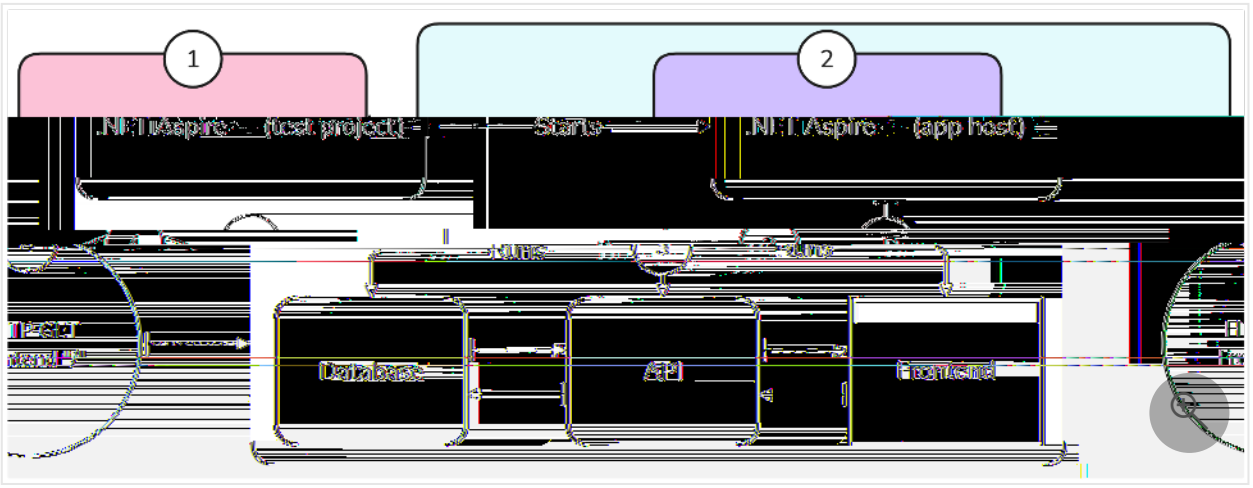
By default, the testing builder disables the dashboard and randomizes the ports of proxied resources to enable multiple instances of your application to run concurrently. Once your test completes, disposing of the application or testing builder cleans up your app resources.

To get started writing your first integration test with .NET Aspire, see the Write your first .NET Aspire test article.

Testing .NET Aspire solutions

.NET Aspire's testing capabilities are designed specifically for closed-box integration testing of your entire distributed application. Unlike unit tests or open-box integration tests, which typically run individual components in isolation, .NET Aspire tests launch your complete solution (the app host and all its resources) as separate processes, closely simulating real-world scenarios.

Consider the following diagram that shows how the .NET Aspire testing project starts the app host, which then starts the application and its resources:



1. The **test project** starts the app host.
2. The **app host** process starts.
3. The **app host** runs the `Database`, `API`, and `Frontend` applications.
4. The **test project** sends an HTTP request to the `Frontend` application.

The diagram illustrates that the **test project** starts the app host, which then orchestrates the all dependent app resources—regardless of their type. The test project is able to send an HTTP request to the `Frontend` app, which depends on an `API` app, and the `API` app depends on a `Database`. A successful request confirms that the `Frontend` app can communicate with the `API` app, and that the `API` app can successfully get data from the `Database`. For more information on seeing this approach in action, see the Write your first .NET Aspire test article.

Use .NET Aspire testing when you want to:

By default, .NET Aspire uses random ports to allow multiple instances of your application to run concurrently without interference. It uses .NET Aspire's service discovery to ensure applications can locate each other's endpoints. To disable port randomization, pass `"DcpPublisher:RandomizePorts=false"` when constructing your testing builder, as shown in the following snippet:

C#

```
var builder = await DistributedApplicationTestingBuilder
    .CreateAsync<Projects.MyAppHost>(
        [
            "DcpPublisher:RandomizePorts=false"
        ]);
```

Enable the dashboard

The testing builder disables the .NET Aspire dashboard by default. To enable it, you can set the `DisableDashboard` property to `false`, when creating your testing builder as shown in the following snippet:

C#

```
var builder = await DistributedApplicationTestingBuilder
    .CreateAsync<Projects.MyAppHost>(
        args: [],
        configureBuilder: (appOptions, hostSettings) =>
        {
            appOptions.DisableDashboard = false;
        });
```

See also

- Write your first .NET Aspire test
- Managing the app host in .NET Aspire tests
- Access resources in .NET Aspire tests

Write your first .NET Aspire test

Article • 02/24/2025

In this article, you learn how to create a test project, write tests, and run them for your .NET Aspire solutions. The tests in this article aren't unit tests, but rather functional or integration tests. .NET Aspire includes several variations of testing project templates that you can use to test your .NET Aspire resource dependencies—and their communications. The testing project templates are available for MSTest, NUnit, and xUnit testing frameworks and include a sample test that you can use as a starting point for your tests.

The .NET Aspire test project templates rely on the Aspire.Hosting.Testing NuGet package. This package exposes the DistributedApplicationTestingBuilder class, which is used to create a test host for your distributed application. The distributed application testing builder launches your app host project with instrumentation hooks so that you can access and manipulate the host at various stages of its lifecyle. In particular, DistributedApplicationTestingBuilder provides you access to IDistributedApplicationBuilder and DistributedApplication class to create and start the app host.

Create a test project

The easiest way to create a .NET Aspire test project is to use the testing project template. If you're starting a new .NET Aspire project and want to include test projects, the Visual Studio tooling supports that option. If you're adding a test project to an existing .NET Aspire project, you can use the `dotnet new` command to create a test project:

.NET CLI

```
dotnet new aspire-xunit
```

For more information, see the .NET CLI dotnet new command documentation.

Explore the test project

The following example test project was created as part of the **.NET Aspire Starter Application** template. If you're unfamiliar with it, see Quickstart: Build your first .NET Aspire project. The .NET Aspire test project takes a project reference dependency on the target app host. Consider the template project:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <IsPackable>false</IsPackable>
    <IsTestProject>true</IsTestProject>
  </PropertyGroup>


  <ItemGroup>
    <PackageReference Include="Aspire.Hosting.Testing" Version="9.1.0" />
    <PackageReference Include="coverlet.collector" Version="6.0.4" />
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.13.0" />
    <PackageReference Include="xunit" Version="2.9.3" />
    <PackageReference Include="xunit.runner.visualstudio" Version="3.0.2" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference
Include="..\AspireApp.AppHost\AspireApp.AppHost.csproj" />
  </ItemGroup>

  <ItemGroup>
    <Using Include="System.Net" />
    <Using Include="Microsoft.Extensions.DependencyInjection" />
    <Using Include="Aspire.Hosting.ApplicationModel" />
    <Using Include="Aspire.Hosting.Testing" />
    <Using Include="Xunit" />
  </ItemGroup>

</Project>
```

The preceding project file is fairly standard. There's a `PackageReference` to the Aspire.Hosting.Testing NuGet package, which includes the required types to write tests for .NET Aspire projects.

The template test project includes a `IntegrationTest1` class with a single test. The test verifies the following scenario:

- The app host is successfully created and started.
- The `webfrontend` resource is available and running.
- An HTTP request can be made to the `webfrontend` resource and returns a successful response (HTTP 200 OK).

Consider the following test class:

C#

```
namespace AspireApp.Tests;

public class IntegrationTest1
{
    [Fact]
    public async Task GetWebResourceRootReturnsOkStatusCode()
    {
        // Arrange
        var builder = await DistributedApplicationTestingBuilder
            .CreateAsync<Projects.AspireApp_AppHost>();

        builder.Services.ConfigureHttpClientDefaults(clientBuilder =>
        {
            clientBuilder.AddStandardResilienceHandler();
        });

        // To output logs to the xUnit.net ITestOutputHelper,
        // consider adding a package from https://www.nuget.org/packages?
q=xunit+logging

        await using var app = await builder.BuildAsync();

        await app.StartAsync();

        // Act
        var httpClient = app.CreateHttpClient("webfrontend");

        using var cts = new
CancellationTokenSource(TimeSpan.FromSeconds(30));
        await app.ResourceNotifications.WaitForResourceHealthyAsync(
            "webfrontend",
            cts.Token);

        var response = await httpClient.GetAsync("/");

        // Assert
        Assert.Equal(HttpStatusCode.OK, response.StatusCode);
    }
}
```

The preceding code:

- Relies on the DistributedApplicationTestingBuilder.CreateAsync API to asynchronously create the app host.
  - The `appHost` is an instance of `IDistributedApplicationTestingBuilder` that represents the app host.
  - The `appHost` instance has its service collection configured with the standard HTTP resilience handler. For more information, see Build resilient HTTP apps: Key development patterns.

- The `appHost` has its IDistributedApplicationTestingBuilder.BuildAsync(CancellationToken) method invoked, which returns the `DistributedApplication` instance as the `app`.
  - The `app` has its service provider get the ResourceNotificationService instance.
  - The `app` is started asynchronously.
- An HttpClient is created for the `webfrontend` resource by calling `app.CreateHttpClient`.
- The `resourceNotificationService` is used to wait for the `webfrontend` resource to be available and running.
- A simple HTTP GET request is made to the root of the `webfrontend` resource.
- The test asserts that the response status code is `OK`.

Test resource environment variables

To further test resources and their expressed dependencies in your .NET Aspire solution, you can assert that environment variables are injected correctly. The following example demonstrates how to test that the `webfrontend` resource has an HTTPS environment variable that resolves to the `apiservice` resource:

C#

```
using Aspire.Hosting;

namespace AspireApp.Tests;

public class EnvVarTests
{
    [Fact]
    public async Task WebResourceEnvVarsResolveToApiService()
    {
        // Arrange
        var builder = await DistributedApplicationTestingBuilder
            .CreateAsync<Projects.AspireApp_AppHost>();

        var frontend = builder.CreateResourceBuilder<ProjectResource>
("webfrontend");

        // Act
        var envVars = await
frontend.Resource.GetEnvironmentVariableValuesAsync(
            DistributedApplicationOperation.Publish);

        // Assert
        Assert.Contains(envVars, static (kvp) =>
        {
            var (key, value) = kvp;
```

```
            return key is "services__apiservice__https__0"
                && value is "{apiservice.bindings.https.url}";
        });
    }
}
```

The preceding code:

- Relies on the DistributedApplicationTestingBuilder.CreateAsync API to asynchronously create the app host.
- The `builder` instance is used to retrieve an IResourceWithEnvironment instance named "webfrontend" from the IDistributedApplicationTestingBuilder.Resources.
- The `webfrontend` resource is used to call GetEnvironmentVariableValuesAsync to retrieve its configured environment variables.
- The DistributedApplicationOperation.Publish argument is passed when calling `GetEnvironmentVariableValuesAsync` to specify environment variables that are published to the resource as binding expressions.
- With the returned environment variables, the test asserts that the `webfrontend` resource has an HTTPS environment variable that resolves to the `apiservice` resource.

Summary

The .NET Aspire testing project template makes it easier to create test projects for .NET Aspire solutions. The template project includes a sample test that you can use as a starting point for your tests. The `DistributedApplicationTestingBuilder` follows a familiar pattern to the WebApplicationFactory<TEntryPoint> in ASP.NET Core. It allows you to create a test host for your distributed application and run tests against it.

Finally, when using the `DistributedApplicationTestingBuilder` all resource logs are redirected to the `DistributedApplication` by default. The redirection of resource logs enables scenarios where you want to assert that a resource is logging correctly.

See also

- Unit testing C# in .NET using dotnet test and xUnit
- MSTest overview
- Unit testing C# with NUnit and .NET Core

Manage the app host in .NET Aspire tests

Article • 02/25/2025

When writing functional or integration tests with .NET Aspire, managing the app host instance efficiently is crucial. The app host represents the full application environment and can be costly to create and tear down. This article explains how to manage the app host instance in your .NET Aspire tests.

For writing tests with .NET Aspire, you use the Aspire.Hosting.Testing NuGet package which contains some helper classes to manage the app host instance in your tests.

Use the `DistributedApplicationTestingBuilder` class

In the tutorial on writing your first test, you were introduced to the DistributedApplicationTestingBuilder class which can be used to create the app host instance:

```
C#

var appHost = await DistributedApplicationTestingBuilder
    .CreateAsync<Projects.AspireApp_AppHost>();
```

The `DistributedApplicationTestingBuilder.CreateAsync<T>` method takes the app host project type as a generic parameter to create the app host instance. While this method is executed at the start of each test, it's more efficient to create the app host instance once and share it across tests as the test suite grows.

With xUnit, you implement the IAsyncLifetime interface on the test class to support asynchronous initialization and disposal of the app host instance. The `InitializeAsync` method is used to create the app host instance before the tests are run and the `DisposeAsync` method disposes the app host once the tests are completed.

```
C#

public class WebTests : IAsyncLifetime
{
    private DistributedApplication _app;
```

```
    public async Task InitializeAsync()
    {
        var appHost = await DistributedApplicationTestingBuilder
            .CreateAsync<Projects.AspireApp_AppHost>();

        _app = await appHost.BuildAsync();
    }

    public async Task DisposeAsync() => await _app.DisposeAsync();

    [Fact]
    public async Task GetWebResourceRootReturnsOkStatusCode()
    {
        // test code here
    }
}
```

By capturing the app host in a field when the test run is started, you can access it in each test without the need to recreate it, decreasing the time it takes to run the tests. Then, when the test run completes, the app host is disposed, which cleans up any resources that were created during the test run, such as containers.

Pass arguments to your app host

You can access the arguments from your app host with the `args` parameter. Arguments are also passed to .NET's configuration system, so you can override many configuration settings this way. In the following example, you override the environment by specifying it as a command line option:

C#

```
var builder = await DistributedApplicationTestingBuilder
    .CreateAsync<Projects.MyAppHost>(
    [
        "--environment=Testing"
    ]);
```

Other arguments can be passed to your app host `Program` and made available in your app host. In the next example, you pass an argument to the app host and use it to control whether you add data volumes to a Postgres instance.

In the app host `Program`, you use configuration to support enabling or disabling volumes:

C#

```
var postgres = builder.AddPostgres("postgres1");
if (builder.Configuration.GetValue("UseVolumes", true))
{
    postgres.WithDataVolume();
}
```

In test code, you pass `"UseVolumes=false"` in the `args` to the app host:

C#

```
public async Task DisableVolumesFromTest()
{
    // Disable volumes in the test builder via arguments:
    using var builder = await DistributedApplicationTestingBuilder
        .CreateAsync<Projects.TestingAppHost1_AppHost>(
        [
            "UseVolumes=false"
        ]);

    // The container will have no volume annotation since we disabled
volumes by passing UseVolumes=false
    var postgres = builder.Resources.Single(r => r.Name == "postgres1");

    Assert.Empty(postgres.Annotations.OfType<ContainerMountAnnotation>());
}
```

Use the `DistributedApplicationFactory` class

While the `DistributedApplicationTestingBuilder` class is useful for many scenarios, there might be situations where you want more control over starting the app host, such as executing code before the builder is created or after the app host is built. In these cases, you implement your own version of the DistributedApplicationFactory class. This is what the `DistributedApplicationTestingBuilder` uses internally.

C#

```
public class TestingAspireAppHost()
    : DistributedApplicationFactory(typeof(Projects.AspireApp_AppHost))
{
    // override methods here
}
```

The constructor requires the type of the app host project reference as a parameter. Optionally, you can provide arguments to the underlying host application builder. These

arguments control how the app host starts and provide values to the args variable used by the *Program.cs* file to start the app host instance.

Lifecycle methods

The `DistributionApplicationFactory` class provides several lifecycle methods that can be overridden to provide custom behavior throughout the preparation and creation of the app host. The available methods are `OnBuilderCreating`, `OnBuilderCreated`, `OnBuilding`, and `OnBuilt`.

For example, we can use the `OnBuilderCreating` method to set configuration, such as the subscription and resource group information for Azure, before the app host is created and any dependent Azure resources are provisioned, resulting in our tests using the correct Azure environment.

C#

```
public class TestingAspireAppHost() :
DistributedApplicationFactory(typeof(Projects.AspireApp_AppHost))
{
    protected override void OnBuilderCreating(DistributedApplicationOptions
applicationOptions, HostApplicationBuilderSettings hostOptions)
    {
        hostOptions.Configuration ??= new();
        hostOptions.Configuration["environment"] = "Development";
        hostOptions.Configuration["AZURE_SUBSCRIPTION_ID"] = "00000000-0000-
0000-0000-000000000000";
        hostOptions.Configuration["AZURE_RESOURCE_GROUP"] = "my-resource-
group";
    }
}
```

Because of the order of precedence in the .NET configuration system, the environment variables will be used over anything in the *appsettings.json* or *secrets.json* file.

Another scenario you might want to use in the lifecycle is to configure the services used by the app host. In the following example, consider a scenario where you override the `OnBuilderCreated` API to add resilience to the `HttpClient`:

C#

```
protected override void OnBuilderCreated(
    DistributedApplicationBuilder applicationBuilder)
{
    applicationBuilder.Services.ConfigureHttpClientDefaults(clientBuilder =>
    {
        clientBuilder.AddStandardResilienceHandler();
```

```
    });
}
```

See also

- Write your first .NET Aspire test

Access resources in .NET Aspire tests

Article • 02/25/2025

In this article, you learn how to access the resources from the .NET Aspire app host in your tests. The app host represents the full application environment and contains all the resources that are available to the application. When writing functional or integration tests with .NET Aspire, you might need to access these resources to verify the behavior of your application.

Access HTTP resources

To access an HTTP resource, use the HttpClient to request and receive responses. The DistributedApplication and the DistributedApplicationFactory both provide a CreateHttpClient method that's used to create an `HttpClient` instance for a specific resource, based on the resource name from the app host. This method also takes an optional `endpointName` parameter, so if the resource has multiple endpoints, you can specify which one to use.

Access other resources

In a test, you might want to access other resources by the connection information they provide, for example, querying a database to verify the state of the data. For this, you use the ConfigurationExtensions.GetConnectionString method to retrieve the connection string for a resource, and then provide that to a client library within the test to interact with the resource.

Ensure resources are available

Starting with .NET Aspire 9, there's support for waiting on dependent resources to be available (via the health check mechanism). This is useful in tests that ensure a resource is available before attempting to access it. The ResourceNotificationService class provides a ResourceNotificationService.WaitForResourceAsync method that's used to wait for a named resource to be available. This method takes the resource name and the desired state of the resource as parameters and returns a Task that yields back when the resource is available. You can access the ResourceNotificationService via DistributedApplication.ResourceNotifications, as in the following example.

> **Note**

It's recommended to provide a time-out when waiting for resources, to prevent the test from hanging indefinitely in situations where a resource never becomes available.

C#

```
using var cts = new CancellationTokenSource(TimeSpan.FromSeconds(30));
await app.ResourceNotifications.WaitForResourceAsync(
    "webfrontend",
    KnownResourceStates.Running,
    cts.Token);
```

A resource enters the KnownResourceStates.Running state as soon as it starts executing, but this doesn't mean that it's ready to serve requests. If you want to wait for the resource to be ready to serve requests, and your resource has health checks, you can wait for the resource to become healthy by using the ResourceNotificationService.WaitForResourceHealthyAsync method.

C#

```
using var cts = new CancellationTokenSource(TimeSpan.FromSeconds(30));

await app.ResourceNotifications.WaitForResourceHealthyAsync(
    "webfrontend",
    cts.Token);
```

This resource-notification pattern ensures that the resources are available before running the tests, avoiding potential issues with the tests failing due to the resources not being ready.

See also

- Write your first .NET Aspire test
- Managing the app host in .NET Aspire tests

.NET Aspire service discovery

Article • 12/31/2024

In this article, you learn how service discovery works within a .NET Aspire project. .NET Aspire includes functionality for configuring service discovery at development and testing time. Service discovery functionality works by providing configuration in the format expected by the *configuration-based endpoint resolver* from the .NET Aspire AppHost project to the individual service projects added to the application model. For more information, see Service discovery in .NET.

Implicit service discovery by reference

Configuration for service discovery is only added for services that are referenced by a given project. For example, consider the following AppHost program:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var catalog = builder.AddProject<Projects.CatalogService>("catalog");
var basket = builder.AddProject<Projects.BasketService>("basket");

var frontend = builder.AddProject<Projects.MyFrontend>("frontend")
                      .WithReference(basket)
                      .WithReference(catalog);
```

In the preceding example, the *frontend* project references the *catalog* project and the *basket* project. The two WithReference calls instruct the .NET Aspire project to pass service discovery information for the referenced projects (*catalog*, and *basket*) into the *frontend* project.

Named endpoints

Some services expose multiple, named endpoints. Named endpoints can be resolved by specifying the endpoint name in the host portion of the HTTP request URI, following the format `scheme://_endpointName.serviceName`. For example, if a service named "basket" exposes an endpoint named "dashboard", then the URI `https+http://_dashboard.basket` can be used to specify this endpoint, for example:

C#

```
builder.Services.AddHttpClient<BasketServiceClient>(
    static client => client.BaseAddress = new("https+http://basket"));

builder.Services.AddHttpClient<BasketServiceDashboardClient>(
    static client => client.BaseAddress =
new("https+http://_dashboard.basket"));
```

In the preceding example, two HttpClient classes are added, one for the core basket service and one for the basket service's dashboard.

Named endpoints using configuration

With the configuration-based endpoint resolver, named endpoints can be specified in configuration by prefixing the endpoint value with `_endpointName.`, where `endpointName` is the endpoint name. For example, consider this *appsettings.json* configuration which defined a default endpoint (with no name) and an endpoint named "dashboard":

JSON

```
{
  "Services": {
    "basket":
      "https": "https://10.2.3.4:8080", /* the https endpoint, requested via
https://basket */
      "dashboard": "https://10.2.3.4:9999" /* the "dashboard" endpoint,
requested via https://_dashboard.basket */
    }
  }
}
```

In the preceding JSON:

- The default endpoint, when resolving `https://basket` is `10.2.3.4:8080`.
- The "dashboard" endpoint, resolved via `https://_dashboard.basket` is `10.2.3.4:9999`.

Named endpoints in .NET Aspire

C#

```
var basket = builder.AddProject<Projects.BasketService>("basket")
    .WithHttpsEndpoint(hostPort: 9999, name: "dashboard");
```

Named endpoints in Kubernetes using DNS SRV

When deploying to Kubernetes, the DNS SRV service endpoint resolver can be used to resolve named endpoints. For example, the following resource definition will result in a DNS SRV record being created for an endpoint named "default" and an endpoint named "dashboard", both on the service named "basket".

yml

```yml
apiVersion: v1
kind: Service
metadata:
  name: basket
spec:
  selector:
    name: basket-service
  clusterIP: None
  ports:
  - name: default
    port: 8080
  - name: dashboard
    port: 9999
```

To configure a service to resolve the "dashboard" endpoint on the "basket" service, add the DNS SRV service endpoint resolver to the host builder as follows:

C#

```
builder.Services.AddServiceDiscoveryCore();
builder.Services.AddDnsSrvServiceEndpointProvider();
```

For more information, see AddServiceDiscoveryCore and AddDnsSrvServiceEndpointProvider.

The special port name "default" is used to specify the default endpoint, resolved using the URI `https://basket`.

As in the previous example, add service discovery to an `HttpClient` for the basket service:

C#

```
builder.Services.AddHttpClient<BasketServiceClient>(
    static client => client.BaseAddress = new("https://basket"));
```

Similarly, the "dashboard" endpoint can be targeted as follows:

C#

```
builder.Services.AddHttpClient<BasketServiceDashboardClient>(
    static client => client.BaseAddress = new("https://_dashboard.basket"));
```

See also

- Service discovery in .NET
- Make HTTP requests with the HttpClient class
- IHttpClientFactory with .NET

.NET Aspire service defaults

Article • 11/04/2024

In this article, you learn about the .NET Aspire service defaults project, a set of extension methods that:

- Connect telemetry, health checks, service discovery to your app.
- Are customizable and extensible.

Cloud-native applications often require extensive configurations to ensure they work across different environments reliably and securely. .NET Aspire provides many helper methods and tools to streamline the management of configurations for OpenTelemetry, health checks, environment variables, and more.

Explore the service defaults project

When you either **Enlist in .NET Aspire orchestration** or create a new .NET Aspire project, the *YourAppName.ServiceDefaults.csproj* project is added to your solution. For example, when building an API, you call the `AddServiceDefaults` method in the *Program.cs* file of your apps:

```
C#

builder.AddServiceDefaults();
```

The `AddServiceDefaults` method handles the following tasks:

- Configures OpenTelemetry metrics and tracing.
- Adds default health check endpoints.
- Adds service discovery functionality.
- Configures HttpClient to work with service discovery.

For more information, see Provided extension methods for details on the `AddServiceDefaults` method.

> **ⓘ Important**
>
> The .NET Aspire service defaults project is specifically designed for sharing the *Extensions.cs* file and its functionality. Don't include other shared functionality or

models in this project. Use a conventional shared class library project for those purposes.

Project characteristics

The *YourAppName.ServiceDefaults* project is a .NET 9.0 library that contains the following XML:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <IsAspireSharedProject>true</IsAspireSharedProject>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />

    <PackageReference Include="Microsoft.Extensions.Http.Resilience"
Version="9.3.0" />
    <PackageReference Include="Microsoft.Extensions.ServiceDiscovery"
Version="9.1.0" />
    <PackageReference Include="OpenTelemetry.Exporter.OpenTelemetryProtocol"
Version="1.11.2" />
    <PackageReference Include="OpenTelemetry.Extensions.Hosting"
Version="1.11.2" />
    <PackageReference Include="OpenTelemetry.Instrumentation.AspNetCore"
Version="1.11.1" />
    <PackageReference Include="OpenTelemetry.Instrumentation.Http"
Version="1.11.1" />
    <PackageReference Include="OpenTelemetry.Instrumentation.Runtime"
Version="1.11.1" />
  </ItemGroup>

</Project>
```

The service defaults project template imposes a `FrameworkReference` dependency on `Microsoft.AspNetCore.App`.

> **Tip**
>
> If you don't want to take a dependency on `Microsoft.AspNetCore.App`, you can create a custom service defaults project. For more information, see Custom service

defaults.

The `IsAspireSharedProject` property is set to `true`, which indicates that this project is a shared project. The .NET Aspire tooling uses this project as a reference for other projects added to a .NET Aspire solution. When you enlist the new project for orchestration, it automatically references the *YourAppName.ServiceDefaults* project and updates the *Program.cs* file to call the `AddServiceDefaults` method.

Provided extension methods

The *YourAppName.ServiceDefaults* project exposes a single *Extensions.cs* file that contains several opinionated extension methods:

- `AddServiceDefaults`: Adds service defaults functionality.
- `ConfigureOpenTelemetry`: Configures OpenTelemetry metrics and tracing.
- `AddDefaultHealthChecks`: Adds default health check endpoints.
- `MapDefaultEndpoints`: Maps the health checks endpoint to `/health` and the liveness endpoint to `/alive`.

Add service defaults functionality

The `AddServiceDefaults` method defines default configurations with the following opinionated functionality:

C#

```
public static IHostApplicationBuilder AddServiceDefaults(
    this IHostApplicationBuilder builder)
{
    builder.ConfigureOpenTelemetry();

    builder.AddDefaultHealthChecks();

    builder.Services.AddServiceDiscovery();

    builder.Services.ConfigureHttpClientDefaults(http =>
    {
        // Turn on resilience by default
        http.AddStandardResilienceHandler();

        // Turn on service discovery by default
        http.AddServiceDiscovery();
    });

    // Uncomment the following to restrict the allowed schemes for service
```

```
discovery.
    // builder.Services.Configure<ServiceDiscoveryOptions>(options =>
    // {
    //     options.AllowedSchemes = ["https"];
    // });

    return builder;
}
```

The preceding code:

- Configures OpenTelemetry metrics and tracing, by calling the `ConfigureOpenTelemetry` method.
- Adds default health check endpoints, by calling the `AddDefaultHealthChecks` method.
- Adds service discovery functionality, by calling the `AddServiceDiscovery` method.
- Configures HttpClient defaults, by calling the `ConfigureHttpClientDefaults` method—which is based on Build resilient HTTP apps: Key development patterns:
  - Adds the standard HTTP resilience handler, by calling the `AddStandardResilienceHandler` method.
  - Specifies that the IHttpClientBuilder should use service discovery, by calling the `UseServiceDiscovery` method.
- Returns the `IHostApplicationBuilder` instance to allow for method chaining.

OpenTelemetry configuration

Telemetry is a critical part of any cloud-native application. .NET Aspire provides a set of opinionated defaults for OpenTelemetry, which are configured with the `ConfigureOpenTelemetry` method:

C#

```
public static IHostApplicationBuilder ConfigureOpenTelemetry(
    this IHostApplicationBuilder builder)
{
    builder.Logging.AddOpenTelemetry(logging =>
    {
        logging.IncludeFormattedMessage = true;
        logging.IncludeScopes = true;
    });

    builder.Services.AddOpenTelemetry()
        .WithMetrics(metrics =>
        {
            metrics.AddAspNetCoreInstrumentation()
                .AddHttpClientInstrumentation()
```

```
                .AddRuntimeInstrumentation();
        })
        .WithTracing(tracing =>
        {
            if (builder.Environment.IsDevelopment())
            {
                // We want to view all traces in development
                tracing.SetSampler(new AlwaysOnSampler());
            }

            tracing.AddAspNetCoreInstrumentation()
                // Uncomment the following line to enable gRPC
instrumentation
                // (requires the OpenTelemetry.Instrumentation.GrpcNetClient
package)
                //.AddGrpcClientInstrumentation()
                .AddHttpClientInstrumentation();
        });

    builder.AddOpenTelemetryExporters();

    return builder;
}
```

The `ConfigureOpenTelemetry` method:

- Adds .NET Aspire telemetry logging to include formatted messages and scopes.
- Adds OpenTelemetry metrics and tracing that include:
  - Runtime instrumentation metrics.
  - ASP.NET Core instrumentation metrics.
  - HttpClient instrumentation metrics.
  - In a development environment, the `AlwaysOnSampler` is used to view all traces.
  - Tracing details for ASP.NET Core, gRPC and HTTP instrumentation.
- Adds OpenTelemetry exporters, by calling `AddOpenTelemetryExporters`.

The `AddOpenTelemetryExporters` method is defined privately as follows:

C#

```
private static IHostApplicationBuilder AddOpenTelemetryExporters(
    this IHostApplicationBuilder builder)
{
    var useOtlpExporter = !string.IsNullOrWhiteSpace(
        builder.Configuration["OTEL_EXPORTER_OTLP_ENDPOINT"]);

    if (useOtlpExporter)
    {
        builder.Services.Configure<OpenTelemetryLoggerOptions>(
            logging => logging.AddOtlpExporter());
        builder.Services.ConfigureOpenTelemetryMeterProvider(
```

```
            metrics => metrics.AddOtlpExporter());
        builder.Services.ConfigureOpenTelemetryTracerProvider(
            tracing => tracing.AddOtlpExporter());
    }

    // Uncomment the following lines to enable the Prometheus exporter
    // (requires the OpenTelemetry.Exporter.Prometheus.AspNetCore package)
    // builder.Services.AddOpenTelemetry()
    //    .WithMetrics(metrics => metrics.AddPrometheusExporter());

    // Uncomment the following lines to enable the Azure Monitor exporter
    // (requires the Azure.Monitor.OpenTelemetry.AspNetCore package)
    //if (!string.IsNullOrEmpty(
    //    builder.Configuration["APPLICATIONINSIGHTS_CONNECTION_STRING"]))
    //{
    //    builder.Services.AddOpenTelemetry()
    //       .UseAzureMonitor();
    //}

    return builder;
}
```

The `AddOpenTelemetryExporters` method adds OpenTelemetry exporters based on the following conditions:

- If the `OTEL_EXPORTER_OTLP_ENDPOINT` environment variable is set, the OpenTelemetry exporter is added.
- Optionally consumers of .NET Aspire service defaults can uncomment some code to enable the Prometheus exporter, or the Azure Monitor exporter.

For more information, see .NET Aspire telemetry.

Health checks configuration

Health checks are used by various tools and systems to assess the readiness of your app. .NET Aspire provides a set of opinionated defaults for health checks, which are configured with the `AddDefaultHealthChecks` method:

C#

```
public static IHostApplicationBuilder AddDefaultHealthChecks(
    this IHostApplicationBuilder builder)
{
    builder.Services.AddHealthChecks()
        // Add a default liveness check to ensure app is responsive
        .AddCheck("self", () => HealthCheckResult.Healthy(), ["live"]);
```

```
    return builder;
}
```

The `AddDefaultHealthChecks` method adds a default liveness check to ensure the app is responsive. The call to AddHealthChecks registers the HealthCheckService. For more information, see .NET Aspire health checks.

Web app health checks configuration

To expose health checks in a web app, .NET Aspire automatically determines the type of project being referenced within the solution, and adds the appropriate call to `MapDefaultEndpoints`:

C#

```
public static WebApplication MapDefaultEndpoints(this WebApplication app)
{
    // Uncomment the following line to enable the Prometheus endpoint
    // (requires the OpenTelemetry.Exporter.Prometheus.AspNetCore package)
    // app.MapPrometheusScrapingEndpoint();

    // Adding health checks endpoints to applications in non-development
    // environments has security implications.
    // See https://aka.ms/dotnet/aspire/healthchecks for details before
    // enabling these endpoints in non-development environments.
    if (app.Environment.IsDevelopment())
    {
        // All health checks must pass for app to be considered ready to
        // accept traffic after starting
        app.MapHealthChecks("/health");

        // Only health checks tagged with the "live" tag must pass for
        // app to be considered alive
        app.MapHealthChecks("/alive", new HealthCheckOptions
        {
            Predicate = r => r.Tags.Contains("live")
        });
    }

    return app;
}
```

The `MapDefaultEndpoints` method:

- Allows consumers to optionally uncomment some code to enable the Prometheus endpoint.
- Maps the health checks endpoint to `/health`.

- Maps the liveness endpoint to `/alive` route where the health check tag contains `live`.

For more information, see .NET Aspire health checks.

Custom service defaults

If the default service configuration provided by the project template is not sufficient for your needs, you have the option to create your own service defaults project. This is especially useful when your consuming project, such as a Worker project or WinForms project, cannot or does not want to have a `FrameworkReference` dependency on `Microsoft.AspNetCore.App`.

To do this, create a new .NET 9.0 class library project and add the necessary dependencies to the project file, consider the following example:

XML

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Library</OutputType>
    <TargetFramework>net9.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" />
    <PackageReference Include="Microsoft.Extensions.ServiceDiscovery" />
    <PackageReference Include="Microsoft.Extensions.Http.Resilience" />
    <PackageReference Include="OpenTelemetry.Exporter.OpenTelemetryProtocol"
/>
    <PackageReference Include="OpenTelemetry.Extensions.Hosting" />
    <PackageReference Include="OpenTelemetry.Instrumentation.Http" />
    <PackageReference Include="OpenTelemetry.Instrumentation.Runtime" />
  </ItemGroup>
</Project>
```

Then create an extensions class that contains the necessary methods to configure the app defaults:

C#

```csharp
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using OpenTelemetry.Logs;
using OpenTelemetry.Metrics;
using OpenTelemetry.Trace;
```

```
namespace Microsoft.Extensions.Hosting;

public static class AppDefaultsExtensions
{
    public static IHostApplicationBuilder AddAppDefaults(
        this IHostApplicationBuilder builder)
    {
        builder.ConfigureAppOpenTelemetry();

        builder.Services.AddServiceDiscovery();

        builder.Services.ConfigureHttpClientDefaults(http =>
        {
            // Turn on resilience by default
            http.AddStandardResilienceHandler();

            // Turn on service discovery by default
            http.AddServiceDiscovery();
        });

        return builder;
    }

    public static IHostApplicationBuilder ConfigureAppOpenTelemetry(
        this IHostApplicationBuilder builder)
    {
        builder.Logging.AddOpenTelemetry(logging =>
        {
            logging.IncludeFormattedMessage = true;
            logging.IncludeScopes = true;
        });

        builder.Services.AddOpenTelemetry()
            .WithMetrics(static metrics =>
            {
                metrics.AddRuntimeInstrumentation();
            })
            .WithTracing(tracing =>
            {
                if (builder.Environment.IsDevelopment())
                {
                    // We want to view all traces in development
                    tracing.SetSampler(new AlwaysOnSampler());
                }

                tracing.AddGrpcClientInstrumentation()
                       .AddHttpClientInstrumentation();
            });

        builder.AddOpenTelemetryExporters();

        return builder;
    }
```

```
    private static IHostApplicationBuilder AddOpenTelemetryExporters(
        this IHostApplicationBuilder builder)
    {
        var useOtlpExporter =
            !string.IsNullOrWhiteSpace(
                builder.Configuration["OTEL_EXPORTER_OTLP_ENDPOINT"]);

        if (useOtlpExporter)
        {
            builder.Services.Configure<OpenTelemetryLoggerOptions>(
                logging => logging.AddOtlpExporter());
            builder.Services.ConfigureOpenTelemetryMeterProvider(
                metrics => metrics.AddOtlpExporter());
            builder.Services.ConfigureOpenTelemetryTracerProvider(
                tracing => tracing.AddOtlpExporter());
        }

        return builder;
    }
}
```

This is only an example, and you can customize the `AppDefaultsExtensions` class to meet your specific needs.

Next steps

This code is derived from the .NET Aspire Starter Application template and is intended as a starting point. You're free to modify this code however you deem necessary to meet your needs. It's important to know that service defaults project and its functionality are automatically applied to all project resources in a .NET Aspire solution.

- Service discovery in .NET Aspire
- .NET Aspire SDK
- .NET Aspire templates
- Health checks in .NET Aspire
- .NET Aspire telemetry
- Build resilient HTTP apps: Key development patterns

.NET Aspire and launch profiles

Article • 11/22/2024

.NET Aspire makes use of *launch profiles* defined in both the app host and service projects to simplify the process of configuring multiple aspects of the debugging and publishing experience for .NET Aspire-based distributed applications.

Launch profile basics

When creating a new .NET application from a template developers will often see a `Properties` directory which contains a file named *launchSettings.json*. The launch settings file contains a list of *launch profiles*. Each launch profile is a collection of related options which defines how you would like `dotnet` to start your application.

The code below is an example of launch profiles in a *launchSettings.json* file for an **ASP.NET Core** application.

JSON

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": false,
      "applicationUrl": "http://localhost:5130",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": false,
      "applicationUrl": "https://localhost:7106;http://localhost:5130",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

The *launchSettings.json* file above defines two *launch profiles*, `http` and `https`. Each has its own set of environment variables, launch URLs and other options. When launching a

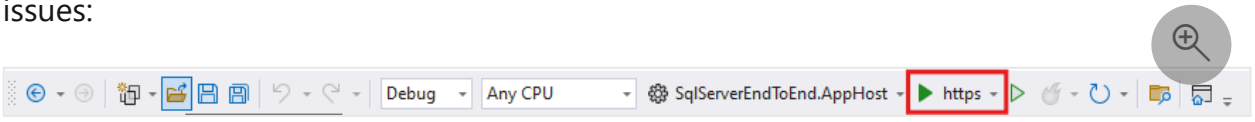
.NET Core application developers can choose which launch profile to use.

.NET CLI

```
dotnet run --launch-profile https
```

If no launch profile is specified, then the first launch profile is selected by default. It is possible to launch a .NET Core application without a launch profile using the `--no-launch-profile` option. Some fields from the *launchSettings.json* file are translated to environment variables. For example, the `applicationUrl` field is converted to the `ASPNETCORE_URLS` environment variable which controls which address and port ASP.NET Core binds to.

In Visual Studio it's possible to select the launch profile when launching the application making it easy to switch between configuration scenarios when manually debugging issues:



When a .NET application is launched with a launch profile a special environment variable called `DOTNET_LAUNCH_PROFILE` is populated with the name of the launch profile that was used when launching the process.

Launch profiles for .NET Aspire app host

In .NET Aspire, the AppHost is just a .NET application. As a result it has a `launchSettings.json` file just like any other application. Here is an example of the `launchSettings.json` file generated when creating a new .NET Aspire project from the starter template (`dotnet new aspire-starter`).

JSON

```
{
  "$schema": "https://json.schemastore.org/launchsettings.json",
  "profiles": {
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:17134;http://localhost:15170",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "DOTNET_ENVIRONMENT": "Development",
        "DOTNET_DASHBOARD_OTLP_ENDPOINT_URL": "https://localhost:21030",
        "DOTNET_RESOURCE_SERVICE_ENDPOINT_URL": "https://localhost:22057"
```

```
      }
    },
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:15170",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "DOTNET_ENVIRONMENT": "Development",
        "DOTNET_DASHBOARD_OTLP_ENDPOINT_URL": "http://localhost:19240",
        "DOTNET_RESOURCE_SERVICE_ENDPOINT_URL": "http://localhost:20154"
      }
    }
  }
}
```

The .NET Aspire templates have a very similar set of *launch profiles* to a regular ASP.NET Core application. When the .NET Aspire app project launches, it starts a DistributedApplication and hosts a web-server which is used by the .NET Aspire Dashboard to fetch information about resources which are being orchestrated by .NET Aspire.

For information about app host configuration options, see .NET Aspire app host configuration.

Relationship between app host launch profiles and service projects

In .NET Aspire the app host is responsible for coordinating the launch of multiple service projects. When you run the app host either via the command line or from Visual Studio (or other development environment) a launch profile is selected for the app host. In turn, the app host will attempt to find a matching launch profile in the service projects it is launching and use those options to control the environment and default networking configuration for the service project.

When the app host launches a service project it doesn't simply launch the service project using the `--launch-profile` option. Therefore, there will be no `DOTNET_LAUNCH_PROFILE` environment variable set for service projects. This is because .NET Aspire modifies the `ASPNETCORE_URLS` environment variable (derived from the `applicationUrl` field in the launch profile) to use a different port. By default, .NET Aspire inserts a reverse proxy in front of the ASP.NET Core application to allow for multiple instances of the application using the WithReplicas method.

Other settings such as options from the `environmentVariables` field are passed through to the application without modification.

Control launch profile selection

Ideally, it's possible to align the launch profile names between the app host and the service projects to make it easy to switch between configuration options on all projects coordinated by the app host at once. However, it may be desirable to control launch profile that a specific project uses. The AddProject extension method provides a mechanism to do this.

C#

```
var builder = DistributedApplication.CreateBuilder(args);
builder.AddProject<Projects.InventoryService>(
    "inventoryservice",
    launchProfileName: "mylaunchprofile");
```

The preceding code shows that the `inventoryservice` resource (a .NET project) is launched using the options from the `mylaunchprofile` launch profile. The launch profile precedence logic is as follows:

1. Use the launch profile specified by `launchProfileName` argument if specified.
2. Use the launch profile with the same name as the AppHost (determined by reading the `DOTNET_LAUNCH_PROFILE` environment variable).
3. Use the default (first) launch profile in *launchSettings.json*.
4. Don't use a launch profile.

To force a service project to launch without a launch profile the `launchProfileName` argument on the AddProject method can be set to null.

Launch profiles and endpoints

When adding an ASP.NET Core project to the app host, .NET Aspire will parse the *launchSettings.json* file selecting the appropriate launch profile and automatically generate endpoints in the application model based on the URL(s) present in the `applicationUrl` field. To modify the endpoints that are automatically injected the WithEndpoint extension method.

C#

```
var builder = DistributedApplication.CreateBuilder(args);
builder.AddProject<Projects.InventoryService>("inventoryservice")
       .WithEndpoint("https", endpoint => endpoint.IsProxied = false);
```

The preceding code shows how to disable the reverse proxy that .NET Aspire deploys in front for the .NET Core application and instead allows the .NET Core application to respond directly on requests over HTTP(S). For more information on networking options within .NET Aspire see .NET Aspire inner loop networking overview.

See also

- Kestrel configured endpoints

Health checks in .NET Aspire

Article • 09/24/2024

Health checks provide availability and state information about an app. Health checks are often exposed as HTTP endpoints, but can also be used internally by the app to write logs or perform other tasks based on the current health. Health checks are typically used in combination with an external monitoring service or container orchestrator to check the status of an app. The data reported by health checks can be used for various scenarios:

- Influence decisions made by container orchestrators, load balancers, API gateways, and other management services. For instance, if the health check for a containerized app fails, it might be skipped by a load balancer routing traffic.
- Verify that underlying dependencies are available, such as a database or cache, and return an appropriate status message.
- Trigger alerts or notifications when an app isn't responding as expected.

.NET Aspire health check endpoints

.NET Aspire exposes two default health check HTTP endpoints in **Development** environments when the `AddServiceDefaults` and `MapDefaultEndpoints` methods are called from the *Program.cs* file:

- The `/health` endpoint indicates if the app is running normally where it's ready to receive requests. All health checks must pass for app to be considered ready to accept traffic after starting.

  ```
  HTTP
  ```

  ```
  GET /health
  ```

  The `/health` endpoint returns an HTTP status code 200 and a `text/plain` value of Healthy when the app is *healthy*.

- The `/alive` indicates if an app is running or has crashed and must be restarted. Only health checks tagged with the *live* tag must pass for app to be considered alive.

  ```
  HTTP
  ```

```
GET /alive
```

The `/alive` endpoint returns an HTTP status code 200 and a `text/plain` value of Healthy when the app is *alive*.

The `AddServiceDefaults` and `MapDefaultEndpoints` methods also apply various configurations to your app beyond just health checks, such as OpenTelemetry and service discovery configurations.

Non-development environments

In non-development environments, the `/health` and `/alive` endpoints are disabled by default. If you need to enable them, its recommended to protect these endpoints with various routing features, such as host filtering and/or authorization. For more information, see Health checks in ASP.NET Core.

Additionally, it may be advantageous to configure request timeouts and output caching for these endpoints to prevent abuse or denial-of-service attacks. To do so, consider the following modified `AddDefaultHealthChecks` method:

C#

```
public static IHostApplicationBuilder AddDefaultHealthChecks(this
IHostApplicationBuilder builder)
{
    builder.Services.AddRequestTimeouts(
        configure: static timeouts =>
            timeouts.AddPolicy("HealthChecks", TimeSpan.FromSeconds(5)));

    builder.Services.AddOutputCache(
        configureOptions: static caching =>
            caching.AddPolicy("HealthChecks",
            build: static policy =>
policy.Expire(TimeSpan.FromSeconds(10))));

    builder.Services.AddHealthChecks()
        // Add a default liveness check to ensure app is responsive
        .AddCheck("self", () => HealthCheckResult.Healthy(), ["live"]);

    return builder;
}
```

The preceding code:

- Adds a timeout of 5 seconds to the health check requests with a policy named `HealthChecks`.
- Adds a 10-second cache to the health check responses with a policy named `HealthChecks`.

Now consider the updated `MapDefaultEndpoints` method:

C#

```
public static WebApplication MapDefaultEndpoints(this WebApplication app)
{
    var healthChecks = app.MapGroup("");

    healthChecks
        .CacheOutput("HealthChecks")
        .WithRequestTimeout("HealthChecks");

    // All health checks must pass for app to be
    // considered ready to accept traffic after starting
    healthChecks.MapHealthChecks("/health");

    // Only health checks tagged with the "live" tag
    // must pass for app to be considered alive
    healthChecks.MapHealthChecks("/alive", new()
    {
        Predicate = static r => r.Tags.Contains("live")
    });

    return app;
}
```

The preceding code:

- Groups the health check endpoints under the `/` path.
- Caches the output and specifies a request time with the corresponding `HealthChecks` policy.

In addition to the updated `AddDefaultHealthChecks` and `MapDefaultEndpoints` methods, you must also add the corresponding services for both request timeouts and output caching.

In the appropriate consuming app's entry point (usually the *Program.cs* file), add the following code:

C#

```
// Wherever your services are being registered.
// Before the call to Build().
```

```
builder.Services.AddRequestTimeouts();
builder.Services.AddOutputCache();

var app = builder.Build();

// Wherever your app has been built, before the call to Run().
app.UseRequestTimeouts();
app.UseOutputCache();

app.Run();
```

For more information, see Request timeouts middleware in ASP.NET Core and Output caching middleware in ASP.NET Core.

Integration health checks

.NET Aspire integrations can also register additional health checks for your app. These health checks contribute to the returned status of the `/health` and `/alive` endpoints. For example, the .NET Aspire PostgreSQL integration automatically adds a health check to verify the following conditions:

- A database connection could be established
- A database query could be executed successfully

If either of these operations fail, the corresponding health check also fails.

Configure health checks

You can disable health checks for a given integration using one of the available configuration options. .NET Aspire integrations support Microsoft.Extensions.Configurations to apply settings through config files such as *appsettings.json*:

JSON

```
{
  "Aspire": {
    "Npgsql": {
      "DisableHealthChecks": true,
    }
  }
}
```

You can also use an inline delegate to configure health checks:

C#

```
builder.AddNpgsqlDbContext<MyDbContext>(
    "postgresdb",
    static settings => settings.DisableHealthChecks  = true);
```

See also

- .NET app health checks in C#
- Health checks in ASP.NET Core

.NET Aspire telemetry

Article • 08/29/2024

One of the primary objectives of .NET Aspire is to ensure that apps are straightforward to debug and diagnose. .NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as the pillars of observability, using the .NET OpenTelemetry SDK.

- **Logging**: Log events describe what's happening as an app runs. A baseline set is enabled for .NET Aspire integrations by default and more extensive logging can be enabled on-demand to diagnose particular problems.
- **Tracing**: Traces correlate log events that are part of the same logical activity (e.g. the handling of a single request), even if they're spread across multiple machines or processes.
- **Metrics**: Metrics expose the performance and health characteristics of an app as simple numerical values. As a result, they have low performance overhead and many services configure them as always-on telemetry. This also makes them suitable for triggering alerts when potential problems are detected.

Together, these types of telemetry allow you to gain insights into your application's behavior and performance using various monitoring and analysis tools. Depending on the backing service, some integrations may only support some of these features.

.NET Aspire OpenTelemetry integration

The .NET OpenTelemetry SDK includes features for gathering data from several .NET APIs, including ILogger, Activity, Meter, and Instrument<T>. These APIs correspond to telemetry features like logging, tracing, and metrics. .NET Aspire projects define OpenTelemetry SDK configurations in the *ServiceDefaults* project. For more information, see .NET Aspire service defaults.

By default, the `ConfigureOpenTelemetry` method enables logging, tracing, and metrics for the app. It also adds exporters for these data points so they can be collected by other monitoring tools.

Export OpenTelemetry data for monitoring

The .NET OpenTelemetry SDK facilitates the export of this telemetry data to a data store or reporting tool. The telemetry export mechanism relies on the OpenTelemetry

protocol (OTLP), which serves as a standardized approach for transmitting telemetry data through REST or gRPC. The `ConfigureOpenTelemetry` method also registers exporters to provide your telemetry data to other monitoring tools, such as Prometheus or Azure Monitor. For more information, see OpenTelemetry configuration.

OpenTelemetry environment variables

OpenTelemetry has a list of known environment variables that configure the most important behavior for collecting and exporting telemetry. OpenTelemetry SDKs, including the .NET SDK, support reading these variables.

.NET Aspire projects launch with environment variables that configure the name and ID of the app in exported telemetry and set the address endpoint of the OTLP server to export data. For example:

- `OTEL_SERVICE_NAME` = myfrontend
- `OTEL_RESOURCE_ATTRIBUTES` = service.instance.id=1a5f9c1e-e5ba-451b-95ee-ced1ee89c168
- `OTEL_EXPORTER_OTLP_ENDPOINT` = `http://localhost:4318`

The environment variables are automatically set in local development.

.NET Aspire local development

When you create a .NET Aspire project, the .NET Aspire dashboard provides a UI for viewing app telemetry by default. Telemetry data is sent to the dashboard using OTLP, and the dashboard implements an OTLP server to receive telemetry data and store it in memory. The .NET Aspire debugging workflow is as follows:

- Developer starts the .NET Aspire project with debugging, presses `F5`.
- .NET Aspire dashboard and developer control plane (DCP) start.
- App configuration is run in the *AppHost* project.
  - OpenTelemetry environment variables are automatically added to .NET projects during app configuration.
  - DCP provides the name (`OTEL_SERVICE_NAME`) and ID (`OTEL_RESOURCE_ATTRIBUTES`) of the app in exported telemetry.
  - The OTLP endpoint is an HTTP/2 port started by the dashboard. This endpoint is set in the `OTEL_EXPORTER_OTLP_ENDPOINT` environment variable on each project. That tells projects to export telemetry back to the dashboard.
  - Small export intervals (`OTEL_BSP_SCHEDULE_DELAY`, `OTEL_BLRP_SCHEDULE_DELAY`, `OTEL_METRIC_EXPORT_INTERVAL`) so data is quickly available in the dashboard.

Small values are used in local development to prioritize dashboard responsiveness over efficiency.

- The DCP starts configured projects, containers, and executables.
- Once started, apps send telemetry to the dashboard.
- Dashboard displays near real-time telemetry of all .NET Aspire projects.

All of these steps happen internally, so in most cases the developer simply needs to run the app to see this process in action.

.NET Aspire deployment

.NET Aspire deployment environments should configure OpenTelemetry environment variables that make sense for their environment. For example, `OTEL_EXPORTER_OTLP_ENDPOINT` should be configured to the environment's local OTLP collector or monitoring service.

.NET Aspire telemetry works best in environments that support OTLP. OTLP exporting is disabled if `OTEL_EXPORTER_OTLP_ENDPOINT` isn't configured.

For more information, see .NET Aspire deployments.

.NET Aspire integrations overview

Article • 02/06/2025

.NET Aspire integrations are a curated suite of NuGet packages selected to facilitate the integration of cloud-native applications with prominent services and platforms, such as Redis and PostgreSQL. Each integration furnishes essential cloud-native functionalities through either automatic provisioning or standardized configuration patterns.

> **Tip**
>
> Always strive to use the latest version of .NET Aspire integrations to take advantage of the latest features, improvements, and security updates.

Integration responsibilities

Most .NET Aspire integrations are made up of two separate libraries, each with a different responsibility. One type represents resources within the *app host* project—known as hosting integrations. The other type of integration represents client libraries that connect to the resources modeled by hosting integrations, and they're known as client integrations.

Hosting integrations

Hosting integrations configure applications by provisioning resources (like containers or cloud resources) or pointing to existing instances (such as a local SQL server). These packages model various services, platforms, or capabilities, including caches, databases, logging, storage, and messaging systems.

Hosting integrations extend the IDistributedApplicationBuilder interface, enabling the *app host* project to express resources within its *app model*. The official hosting integration NuGet packages are tagged with `aspire`, `integration`, and `hosting`. In addition to the official hosting integrations, the community has created hosting integrations for various services and platforms as part of the Community Toolkit.

For information on creating a custom *hosting integration*, see Create custom .NET Aspire hosting integration.

Client integrations

Client integrations wire up client libraries to dependency injection (DI), define configuration schema, and add health checks, resiliency, and telemetry where applicable. .NET Aspire client integration libraries are prefixed with `Aspire.` and then include the full package name that they integrate with, such as `Aspire.StackExchange.Redis`.

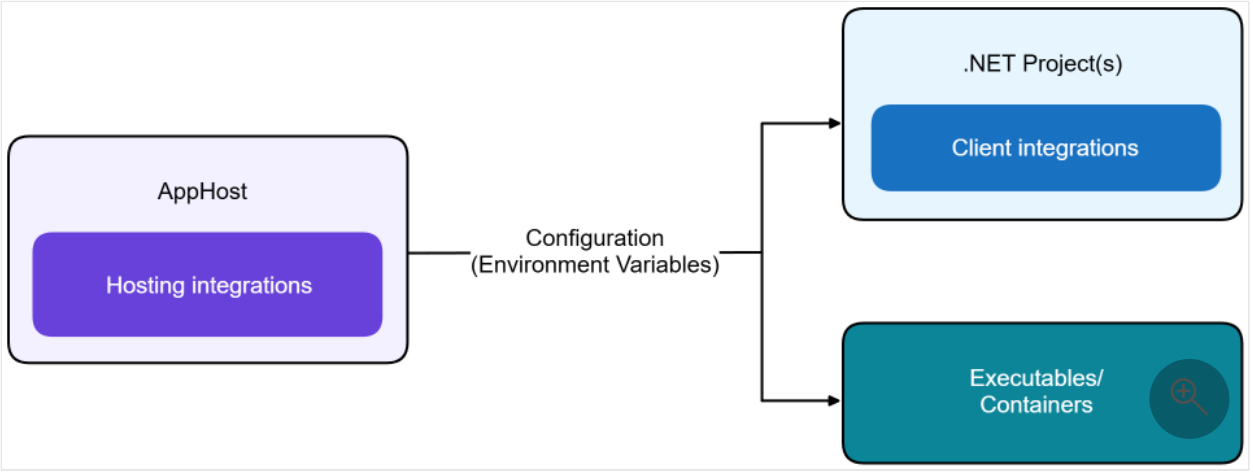
These packages configure existing client libraries to connect to hosting integrations. They extend the IHostApplicationBuilder interface allowing client-consuming projects, such as your web app or API, to use the connected resource. The official client integration NuGet packages are tagged with `aspire`, `integration`, and `client`. In addition to the official client integrations, the community has created client integrations for various services and platforms as part of the Community Toolkit.

For more information on creating a custom client integration, see Create custom .NET Aspire client integrations.

Relationship between hosting and client integrations

Hosting and client integrations are best when used together, but are **not** coupled and can be used separately. Some hosting integrations don't have a corresponding client integration. Configuration is what makes the hosting integration work with the client integration.

Consider the following diagram that depicts the relationship between hosting and client integrations:



The app host project is where hosting integrations are used. Configuration, specifically environment variables, is injected into projects, executables, and containers, allowing client integrations to connect to the hosting integrations.

Integration features

When you add a client integration to a project within your .NET Aspire solution, service defaults are automatically applied to that project; meaning the Service Defaults project is referenced and the `AddServiceDefaults` extension method is called. These defaults are designed to work well in most scenarios and can be customized as needed. The following service defaults are applied:

- **Observability and telemetry**: Automatically sets up logging, tracing, and metrics configurations:
  - **Logging**: A technique where code is instrumented to produce logs of interesting events that occurred while the program was running.
  - **Tracing**: A specialized form of logging that helps you localize failures and performance issues within applications distributed across multiple machines or processes.
  - **Metrics**: Numerical measurements recorded over time to monitor application performance and health. Metrics are often used to generate alerts when potential problems are detected.

- **Health checks**: Exposes HTTP endpoints to provide basic availability and state information about an app. Health checks are used to influence decisions made by container orchestrators, load balancers, API gateways, and other management services.

- **Resiliency**: The ability of your system to react to failure and still remain functional. Resiliency extends beyond preventing failures to include recovering and reconstructing your cloud-native environment back to a healthy state.

Versioning considerations

Hosting and client integrations are updated each release to target the latest stable versions of dependent resources. When container images are updated with new image versions, the hosting integrations update to these new versions. Similarly, when a new NuGet version is available for a dependent client library, the corresponding client integration updates to the new version. This ensures the latest features and security updates are available to applications. The .NET Aspire update type (major, minor, patch) doesn't necessarily indicate the type of update in dependent resources. For example, a new major version of a dependent resource may be updated in a .NET Aspire patch release, if necessary.

When major breaking changes happen in dependent resources, integrations may temporarily split into version-dependent packages to ease updating across the breaking change. For more information, see the first example of such a breaking change.

Official integrations

.NET Aspire provides many integrations to help you build cloud-native applications. These integrations are designed to work seamlessly with the .NET Aspire app host and client libraries. The following sections detail cloud-agnostic, Azure-specific, Amazon Web Services (AWS), and Community Toolkit integrations.

Cloud-agnostic integrations

The following section details cloud-agnostic .NET Aspire integrations with links to their respective docs and NuGet packages, and provides a brief description of each integration.

Expand table

Integration docs and NuGet packages	Description
- **Learn more**: Apache Kafka - **Hosting**: Aspire.Hosting.Kafka - **Client**: Aspire.Confluent.Kafka	A library for producing and consuming messages from an Apache Kafka broker.
- **Learn more**: Dapr - **Hosting**: Aspire.Hosting.Dapr - **Client**: N/A	A library for modeling Dapr as a .NET Aspire resource.
- **Learn more**: Elasticsearch - **Hosting**: Aspire.Hosting.Elasticsearch - **Client**: Aspire.Elastic.Clients.Elasticsearch	A library for accessing Elasticsearch databases.
- **Learn more**: Keycloak - **Hosting**: Aspire.Hosting.Keycloak - **Client**: Aspire.Keycloak.Authentication	A library for accessing Keycloak authentication.
- **Learn more**: Milvus - **Hosting**: Aspire.Hosting.Milvus - **Client**: Aspire.Milvus.Client	A library for accessing Milvus databases.
- **Learn more**: MongoDB Driver - **Hosting**: Aspire.Hosting.MongoDB - **Client**: Aspire.MongoDB.Driver	A library for accessing MongoDB databases.
- **Learn more**: MySqlConnector - **Hosting**: Aspire.Hosting.MySql - **Client**: Aspire.MySqlConnector	A library for accessing MySqlConnector databases.
- **Learn more**: NATS - **Hosting**: Aspire.Hosting.Nats	A library for accessing NATS messaging.

Integration docs and NuGet packages	Description
- **Client**: Aspire.NATS.Net	
- **Learn more**: Oracle - EF Core - **Hosting**: Aspire.Hosting.Oracle - **Client**: Aspire.Oracle.EntityFrameworkCore	A library for accessing Oracle databases with Entity Framework Core.
- **Learn more**: Orleans - **Hosting**: Aspire.Hosting.Orleans - **Client**: N/A	A library for modeling Orleans as a .NET Aspire resource.
- **Learn more**: Pomelo MySQL - EF Core - **Hosting**: Aspire.Hosting.MySql - **Client**: Aspire.Pomelo.EntityFrameworkCore.MySql	A library for accessing MySql databases with Entity Framework Core.
- **Learn more**: PostgreSQL - EF Core - **Hosting**: Aspire.Hosting.PostgreSQL - **Client**: Aspire.Npgsql.EntityFrameworkCore.PostgreSQL	A library for accessing PostgreSQL databases using Entity Framework Core.
- **Learn more**: PostgreSQL - **Hosting**: Aspire.Hosting.PostgreSQL - **Client**: Aspire.Npgsql	A library for accessing PostgreSQL databases.
- **Learn more**: Qdrant - **Hosting**: Aspire.Hosting.Qdrant - **Client**: Aspire.Qdrant.Client	A library for accessing Qdrant databases.
- **Learn more**: RabbitMQ - **Hosting**: Aspire.Hosting.RabbitMQ - **Client**: Aspire.RabbitMQ.Client	A library for accessing RabbitMQ.
- **Learn more**: Redis Distributed Caching - **Hosting**: Aspire.Hosting.Redis, Aspire.Hosting.Garnet, or Aspire.Hosting.Valkey - **Client**: Aspire.StackExchange.Redis.DistributedCaching	A library for accessing Redis caches for distributed caching.
- **Learn more**: Redis Output Caching - **Hosting**: Aspire.Hosting.Redis, Aspire.Hosting.Garnet, or Aspire.Hosting.Valkey - **Client**: Aspire.StackExchange.Redis.OutputCaching	A library for accessing Redis caches for output caching.
- **Learn more**: Redis - **Hosting**: Aspire.Hosting.Redis, Aspire.Hosting.Garnet, or Aspire.Hosting.Valkey - **Client**: Aspire.StackExchange.Redis	A library for accessing Redis caches.
- **Learn more**: Seq - **Hosting**: Aspire.Hosting.Seq - **Client**: Aspire.Seq	A library for logging to Seq.

Integration docs and NuGet packages	Description
- **Learn more**: SQL Server - EF Core - **Hosting**: Aspire.Hosting.SqlServer - **Client**: Aspire.Microsoft.EntityFrameworkCore.SqlServer	A library for accessing SQL Server databases using EF Core.
- **Learn more**: SQL Server - **Hosting**: Aspire.Hosting.SqlServer - **Client**: Aspire.Microsoft.Data.SqlClient	A library for accessing SQL Server databases.

For more information on working with .NET Aspire integrations in Visual Studio, see Visual Studio tooling.

Azure integrations

Azure integrations configure applications to use Azure resources. These hosting integrations are available in the `Aspire.Hosting.Azure.*` NuGet packages, while their client integrations are available in the `Aspire.*` NuGet packages:

Expand table

Integration	Docs and NuGet packages	Description
	- **Learn more**: Azure App Configuration - **Hosting**: Aspire.Hosting.Azure.AppConfiguration - **Client**: N/A	A library for interacting with Azure App Configuration.
	- **Learn more**: Azure Application Insights - **Hosting**: Aspire.Hosting.Azure.ApplicationInsights - **Client**: N/A	A library for interacting with Azure Application Insights.
	- **Learn more**: Azure Cache for Redis - **Hosting**: Aspire.Hosting.Azure.Redis - **Client**: Aspire.StackExchange.Redis or Aspire.StackExchange.Redis.DistributedCaching or Aspire.StackExchange.Redis.OutputCaching	A library for accessing Azure Cache for Redis.
	- **Learn more**: Azure Cosmos DB - EF Core - **Hosting**: Aspire.Hosting.Azure.CosmosDB - **Client**: Aspire.Microsoft.EntityFrameworkCore.Cosmos	A library for accessing Azure Cosmos DB databases with Entity Framework Core.

Integration	Docs and NuGet packages	Description
	- **Learn more:** Azure Cosmos DB - **Hosting:** Aspire.Hosting.Azure.CosmosDB - **Client:** Aspire.Microsoft.Azure.Cosmos	A library for accessing Azure Cosmos DB databases.
	- **Learn more:** Azure Event Hubs - **Hosting:** Aspire.Hosting.Azure.EventHubs - **Client:** Aspire.Azure.Messaging.EventHubs	A library for accessing Azure Event Hubs.
	- **Learn more:** Azure Functions - **Hosting:** Aspire.Hosting.Azure.Functions - **Client:** N/A	A library for integrating with Azure Functions.
	- **Learn more:** Azure Key Vault - **Hosting:** Aspire.Hosting.Azure.KeyVault - **Client:** Aspire.Azure.Security.KeyVault	A library for accessing Azure Key Vault.
	- **Learn more:** Azure Operational Insights - **Hosting:** Aspire.Hosting.Azure.OperationalInsights - **Client:** N/A	A library for interacting with Azure Operational Insights.
	- **Learn more:** Azure AI OpenAI - **Hosting:** Aspire.Hosting.Azure.CognitiveServices - **Client:** Aspire.Azure.AI.OpenAI	A library for accessing Azure AI OpenAI or OpenAI functionality.
	- **Learn more:** Azure PostgreSQL - **Hosting:** Aspire.Hosting.Azure.PostgreSQL - **Client:** N/A	A library for interacting with Azure Database for PostgreSQL.
	- **Learn more:** Azure AI Search - **Hosting:** Aspire.Hosting.Azure.Search - **Client:** Aspire.Azure.Search.Documents	A library for accessing Azure AI Search functionality.
	- **Learn more:** Azure Service Bus - **Hosting:** Aspire.Hosting.Azure.ServiceBus - **Client:** Aspire.Azure.Messaging.ServiceBus	A library for accessing Azure Service Bus.
	- **Learn more:** Azure SignalR Service - **Hosting:** Aspire.Hosting.Azure.SignalR - **Client:** Microsoft.Azure.SignalR	A library for accessing Azure SignalR Service.

Integration	Docs and NuGet packages	Description
	- **Learn more:** Azure Blob Storage - **Hosting:** Aspire.Hosting.Azure.Storage - **Client:** Aspire.Azure.Storage.Blobs	A library for accessing Azure Blob Storage.
	- **Learn more:** Azure Storage Queues - **Hosting:** Aspire.Hosting.Azure.Storage - **Client:** Aspire.Azure.Storage.Queues	A library for accessing Azure Storage Queues.
	- **Learn more:** Azure Table Storage - **Hosting:** Aspire.Hosting.Azure.Storage - **Client:** Aspire.Azure.Data.Tables	A library for accessing the Azure Table service.
	- **Learn more:** Azure Web PubSub - **Hosting:** Aspire.Hosting.Azure.WebPubSub - **Client:** Aspire.Azure.Messaging.WebPubSub	A library for accessing the Azure Web PubSub service.

Amazon Web Services (AWS) hosting integrations

Expand table

Integration docs and NuGet packages	Description
- **Learn more:** AWS Hosting - **Hosting:** Aspire.Hosting.AWS - **Client:** N/A	A library for modeling AWS resources.

For more information, see GitHub: Aspire.Hosting.AWS library.

Community Toolkit integrations

> **Note**
>
> The Community Toolkit integrations are community-driven and maintained by the .NET Aspire community. These integrations are not officially supported by the .NET Aspire team.

Expand table

Integration docs and NuGet packages	Description
- **Learn More**: Azure Static Web Apps emulator - **Hosting**: CommunityToolkit.Aspire.Hosting.Azure.StaticWebApps - **Client**: N/A	A hosting integration for the Azure Static Web Apps emulator (Note: this does not support deployment of a project to Azure Static Web Apps).
- **Learn More**: Bun hosting - **Hosting**: CommunityToolkit.Aspire.Hosting.Bun - **Client**: N/A	A hosting integration for Bun apps.
- **Learn More**: Deno hosting - **Hosting**: CommunityToolkit.Aspire.Hosting.Deno - **Client**: N/A	A hosting integration for Deno apps.
- **Learn More**: Go hosting - **Hosting**: CommunityToolkit.Aspire.Hosting.Golang - **Client**: N/A	A hosting integration for Go apps.
- **Learn More**: Java/Spring hosting - **Hosting**: CommunityToolkit.Aspire.Hosting.Java - **Client**: N/A	A integration for running Java code in .NET Aspire either using the local JDK or using a container.
- **Learn More**: Node.js hosting extensions - **Hosting**: CommunityToolkit.Aspire.Hosting.NodeJs.Extensions - **Client**: N/A	An integration that contains some additional extensions for running Node.js applications
- **Learn More**: Ollama - **Hosting**: CommunityToolkit.Aspire.Hosting.Ollama - **Client**: Aspire.CommunitToolkit.OllamaSharp	An Aspire component leveraging the Ollama container with support for downloading a model on startup.
- **Learn More**: Meilisearch hosting - **Hosting**: CommunityToolkit.Aspire.Hosting.Meilisearch - **Client**: Aspire.CommunitToolkit.Meilisearch	An Aspire component leveraging the Meilisearch container.
- **Learn More**: Rust hosting - **Hosting**: CommunityToolkit.Aspire.Hosting.Rust - **Client**: N/A	A hosting integration for Rust apps.
- **Learn More**: SQL Database projects hosting - **Hosting**: CommunityToolkit.Aspire.Hosting.SqlDatabaseProjects - **Client**: N/A	An Aspire hosting integration for SQL Database Projects.

For more information, see .NET Aspire Community Toolkit.

Tutorial: Implement caching with .NET Aspire integrations

Article • 02/05/2025

Cloud-native apps often require various types of scalable caching solutions to improve performance. .NET Aspire integrations simplify the process of connecting to popular caching services such as Redis. In this article, you'll learn how to:

- ✓ Create a basic ASP.NET core app that is set up to use .NET Aspire.
- ✓ Add .NET Aspire integrations to connect to Redis and implement caching.
- ✓ Configure the .NET Aspire integrations to meet specific requirements.

This article explores how to use two different types of ASP.NET Core caching using .NET Aspire and Redis:

- **Output caching**: A configurable, extensible caching method for storing entire HTTP responses for future requests.
- **Distributed caching**: A cache shared by multiple app servers that allows you to cache specific pieces of data. A distributed cache is typically maintained as an external service to the app servers that access it and can improve the performance and scalability of an ASP.NET Core app.

Prerequisites

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0
- An OCI compliant container runtime, such as:
  - Docker Desktop or Podman. For more information, see Container runtime.
- An Integrated Developer Environment (IDE) or code editor, such as:
  - Visual Studio 2022 version 17.9 or higher (Optional)
  - Visual Studio Code (Optional)
    - C# Dev Kit: Extension (Optional)
  - JetBrains Rider with .NET Aspire plugin (Optional)

For more information, see .NET Aspire setup and tooling, and .NET Aspire SDK.

Create the project

1. At the top of Visual Studio, navigate to **File** > **New** > **Project...**.

2. In the dialog window, enter **.NET Aspire** into the project template search box and select **.NET Aspire Starter Application**. Choose **Next**.
3. On the **Configure your new project** screen:

    - Enter a **Project name** of **AspireRedis**.
    - Leave the rest of the values at their defaults and select **Next**.

4. On the **Additional information** screen:

    - Make sure **.NET 9.0** is selected.
    - Uncheck **Use Redis for caching**. You will implement your own caching setup.
    - Select **Create**.

Visual Studio creates a new .NET Aspire solution that consists of the following projects:

- **AspireRedis.Web** - A Blazor UI project with default .NET Aspire configurations.

```
var builder = DistributedApplication.CreateBuilder(args);

var redis = builder.AddRedis("cache");

var apiservice = builder.AddProject<Projects.AspireRedis_ApiService>
("apiservice")
    .WithReference(redis);

builder.AddProject<Projects.AspireRedis_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(apiservice)
    .WithReference(redis);

builder.Build().Run();
```

The preceding code creates a local Redis container instance and configures the UI and API to use the instance automatically for both output and distributed caching. The code also configures communication between the frontend UI and the backend API using service discovery. With .NET Aspire's implicit service discovery, setting up and managing service connections is streamlined for developer productivity. In the context of this tutorial, the feature simplifies how you connect to Redis.

Traditionally, you'd manually specify the Redis connection string in each project's *appsettings.json* file:

JSON

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

Configuring connection string with this method, while functional, requires duplicating the connection string across multiple projects, which can be cumbersome and error-prone.

Configure the UI with output caching

1. Add the .NET Aspire Stack Exchange Redis output caching integration packages to your `AspireRedis.Web` app:

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.OutputCaching
```

2. In the *Program.cs* file of the `AspireRedis.Web` Blazor project, immediately after the line `var builder = WebApplication.CreateBuilder(args);`, add a call to the AddRedisOutputCache extension method:

```
C#

builder.AddRedisOutputCache("cache");
```

This method accomplishes the following tasks:

- Configures ASP.NET Core output caching to use a Redis instance with the specified connection name.
- Automatically enables corresponding health checks, logging, and telemetry.

3. Replace the contents of the *Home.razor* file of the `AspireRedis.Web` Blazor project with the following:

```
razor

@page "/"
@attribute [OutputCache(Duration = 10)]

<PageTitle>Home</PageTitle>

<h1>Hello, world!</h1>

Welcome to your new app on @DateTime.Now
```

The integration include the `[OutputCache]` attribute, which caches the entire rendered response. The page also include a call to `@DateTime.Now` to help verify that the response is cached.

Configure the API with distributed caching

1. Add the .NET Aspire Stack Exchange Redis distributed caching integration packages to your `AspireRedis.ApiService` app:

```
.NET CLI

dotnet add package Aspire.StackExchange.Redis.DistributedCaching
```

2. Towards the top of the *Program.cs* file, add a call to AddRedisDistributedCache:

```
C#
```

```
builder.AddRedisDistributedCache("cache");
```

3. In the *Program.cs* file, add the following `using` statements:

```C#
using System.Text;
using System.Text.Json;
using Microsoft.Extensions.Caching.Distributed;
```

4. In the *Program.cs* file, replace the existing `/weatherforecast` endpoint code with the following:

```C#
app.MapGet("/weatherforecast", async (IDistributedCache cache) =>
{
    var cachedForecast = await cache.GetAsync("forecast");

    if (cachedForecast is null)
    {
        var summaries = new[] { "Freezing", "Bracing", "Chilly",
"Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching" };
        var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        (
            DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            Random.Shared.Next(-20, 55),
            summaries[Random.Shared.Next(summaries.Length)]
        ))
        .ToArray();

        await cache.SetAsync("forecast",
Encoding.UTF8.GetBytes(JsonSerializer.Serialize(forecast)), new ()
        {
            AbsoluteExpiration = DateTime.Now.AddSeconds(10)
        });

        return forecast;
    }

    return JsonSerializer.Deserialize<IEnumerable<WeatherForecast>>
(cachedForecast);
})
.WithName("GetWeatherForecast");
```

Test the app locally

Test the caching behavior of your app using the following steps:

1. Run the app using Visual Studio by pressing `F5`.
2. If the **Start Docker Desktop** dialog appears, select **Yes** to start the service.
3. The .NET Aspire Dashboard loads in the browser and lists the UI and API projects.

Test the output cache:

1. On the projects page, in the **webfrontend** row, click the `localhost` link in the **Endpoints** column to open the UI of your app.
2. The application will display the current time on the home page.
3. Refresh the browser every few seconds to see the same page returned by output caching. After 10 seconds the cache expires and the page reloads with an updated time.

Test the distributed cache:

1. Navigate to the **Weather** page on the Blazor UI to load a table of randomized weather data.
2. Refresh the browser every few seconds to see the same weather data returned by output caching. After 10 seconds the cache expires and the page reloads with updated weather data.

Congratulations! You configured a ASP.NET Core app to use output and distributed caching with .NET Aspire.

Tutorial: Connect an ASP.NET Core app to SQL Server using .NET Aspire and Entity Framework Core

Article • 03/04/2025

In this tutorial, you create an ASP.NET Core app that uses a .NET Aspire Entity Framework Core SQL Server integration to connect to SQL Server to read and write support ticket data. Entity Framework Core is a lightweight, extensible, open source object-relational mapper that enables .NET developers to work with databases using .NET objects. You'll learn how to:

- ✓ Create a basic .NET app that is set up to use .NET Aspire integrations
- ✓ Add a .NET Aspire integration to connect to SQL Server
- ✓ Configure and use .NET Aspire Component features to read and write from the database

Prerequisites

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0
- An OCI compliant container runtime, such as:
  - Docker Desktop or Podman. For more information, see Container runtime.
- An Integrated Developer Environment (IDE) or code editor, such as:
  - Visual Studio 2022 version 17.9 or higher (Optional)
  - Visual Studio Code (Optional)
    - C# Dev Kit: Extension (Optional)
  - JetBrains Rider with .NET Aspire plugin (Optional)

For more information, see .NET Aspire setup and tooling, and .NET Aspire SDK.

Create the sample solution

1. At the top of Visual Studio, navigate to **File** > **New** > **Project**.
2. In the dialog window, search for *Blazor* and select **Blazor Web App**. Choose **Next**.
3. On the **Configure your new project** screen:
   - Enter a **Project Name** of **AspireSQLEFCore**.
   - Leave the rest of the values at their defaults and select **Next**.

4. On the **Additional information** screen:

    - Make sure **.NET 9.0** is selected.
    - Ensure the **Interactive render mode** is set to **None**.
    - Check the **Enlist in .NET Aspire orchestration** option and select **Create**.

Visual Studio creates a new ASP.NET Core solution that is structured to use .NET Aspire. The solution consists of the following projects:

- **AspireSQLEFCore**: A Blazor project that depends on service defaults.
- **AspireSQLEFCore.AppHost**: An orchestrator project designed to connect and configure the different projects and services of your app. The orchestrator should be set as the startup project.
- **AspireSQLEFCore.ServiceDefaults**: A shared class library to hold configurations that can be reused across the projects in your solution.

Create the database model and context classes

To represent a user submitted support request, add the following `SupportTicket` model class at the root of the *AspireSQLEFCore* project.

C#

```
using System.ComponentModel.DataAnnotations;

namespace AspireSQLEFCore;

public sealed class SupportTicket
{
    public int Id { get; set; }
    [Required]
    public string Title { get; set; } = string.Empty;
    [Required]
    public string Description { get; set; } = string.Empty;
}
```

Add the following `TicketDbContext` data context class at the root of the **AspireSQLEFCore** project. The class inherits System.Data.Entity.DbContext to work with Entity Framework and represent your database.

C#

```
using Microsoft.EntityFrameworkCore;
using System.Reflection.Metadata;

namespace AspireSQLEFCore;
```

```
public class TicketContext(DbContextOptions options) : DbContext(options)
{
    public DbSet<SupportTicket> Tickets => Set<SupportTicket>();
}
```

Add the .NET Aspire integration to the Blazor app

Add the .NET Aspire Entity Framework Core Sql Server library package to your *AspireSQLEFCore* project:

.NET CLI

```
dotnet add package Aspire.Microsoft.EntityFrameworkCore.SqlServer
```

Your *AspireSQLEFCore* project is now set up to use .NET Aspire integrations. Here's the updated *AspireSQLEFCore.csproj* file:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference
Include="Aspire.Microsoft.EntityFrameworkCore.SqlServer" Version="9.1.0" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference
Include="..\AspireSQLEFCore.ServiceDefaults\AspireSQLEFCore.ServiceDefaults.
csproj" />
  </ItemGroup>

</Project>
```

Configure the .NET Aspire integration

In the *Program.cs* file of the *AspireSQLEFCore* project, add a call to the AddSqlServerDbContext extension method after the creation of the `builder` but before the call to `AddServiceDefaults`. For more information, see .NET Aspire service defaults. Provide the name of your connection string as a parameter.

```
C#

using AspireSQLEFCore;
using AspireSQLEFCore.Components;

var builder = WebApplication.CreateBuilder(args);
builder.AddSqlServerDbContext<TicketContext>("sqldata");

builder.AddServiceDefaults();

// Add services to the container.
builder.Services.AddRazorComponents().AddInteractiveServerComponents();

var app = builder.Build();

app.MapDefaultEndpoints();
```

This method accomplishes the following tasks:

- Registers a `TicketContext` with the DI container for connecting to the containerized Azure SQL Database.
- Automatically enable corresponding health checks, logging, and telemetry.

Create the database

While developing locally, you need to create a database inside the SQL Server container. Update the *Program.cs* file with the following code:

```
C#

using AspireSQLEFCore;
using AspireSQLEFCore.Components;

var builder = WebApplication.CreateBuilder(args);
builder.AddSqlServerDbContext<TicketContext>("sqldata");

builder.AddServiceDefaults();

// Add services to the container.
builder.Services.AddRazorComponents().AddInteractiveServerComponents();

var app = builder.Build();
```

```
app.MapDefaultEndpoints();

if (app.Environment.IsDevelopment())
{
    using (var scope = app.Services.CreateScope())
    {
        var context =
scope.ServiceProvider.GetRequiredService<TicketContext>();
        context.Database.EnsureCreated();
    }
}
else
{
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    // The default HSTS value is 30 days.
    // You may want to change this for production scenarios, see
https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}
```

The preceding code:

- Checks if the app is running in a development environment.
- If it is, it retrieves the `TicketContext` service from the DI container and calls `Database.EnsureCreated()` to create the database if it doesn't already exist.

> **Note**
>
> Note that `EnsureCreated()` is not suitable for production environments, and it only creates the database as defined in the context. It doesn't apply any migrations. For more information on Entity Framework Core migrations in .NET Aspire, see Apply Entity Framework Core migrations in .NET Aspire.

Create the form

The app requires a form for the user to be able to submit support ticket information and save the entry to the database.

Use the following Razor markup to create a basic form, replacing the contents of the *Home.razor* file in the *AspireSQLEFCore/Components/Pages* directory:

razor

```
@page "/"
@inject TicketContext context
@using Microsoft.EntityFrameworkCore
```

```
<div class="row">
    <div class="col-md-6">
        <div>
            <h1 class="display-4">Request Support</h1>
        </div>
        <EditForm Model="@Ticket" FormName="Tickets" method="post"
                  OnValidSubmit="@HandleValidSubmit" class="mb-4">
            <DataAnnotationsValidator />
            <div class="mb-4">
                <label>Issue Title</label>
                <InputText class="form-control" @bind-Value="@Ticket.Title"
/>
                <ValidationMessage For="() => Ticket.Title" />
            </div>
            <div class="mb-4">
                <label>Issue Description</label>
                <InputText class="form-control" @bind-
Value="@Ticket.Description" />
                <ValidationMessage For="() => Ticket.Description" />
            </div>
            <button class="btn btn-primary" type="submit">Submit</button>
            <button class="btn btn-danger mx-2" type="reset"
@onclick=@ClearForm>Clear</button>
        </EditForm>

        <table class="table table-striped">
            @foreach (var ticket in Tickets)
            {
                <tr>
                    <td>@ticket.Id</td>
                    <td>@ticket.Title</td>
                    <td>@ticket.Description</td>
                </tr>
            }
        </table>
    </div>
</div>

@code {
    [SupplyParameterFromForm(FormName = "Tickets")]
    private SupportTicket Ticket { get; set; } = new();

    private List<SupportTicket> Tickets = [];

    private void ClearForm() => Ticket = new();

    protected override async Task OnInitializedAsync()
    {
        Tickets = await context.Tickets.ToListAsync();
    }

    private async Task HandleValidSubmit()
    {
        context.Tickets.Add(Ticket);
```

```
        await context.SaveChangesAsync();

        Tickets = await context.Tickets.ToListAsync();

        ClearForm();
    }
}
```

For more information about creating forms in Blazor, see ASP.NET Core Blazor forms overview.

Configure the AppHost

The *AspireSQLEFCore.AppHost* project is the orchestrator for your app. It's responsible for connecting and configuring the different projects and services of your app. The orchestrator should be set as the startup project.

Add the .NET Aspire Hosting Sql Server NuGet package to your *AspireStorage.AppHost* project:

.NET CLI

```
dotnet add package Aspire.Hosting.SqlServer
```

Replace the contents of the *Program.cs* file in the *AspireSQLEFCore.AppHost* project with the following code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var
```

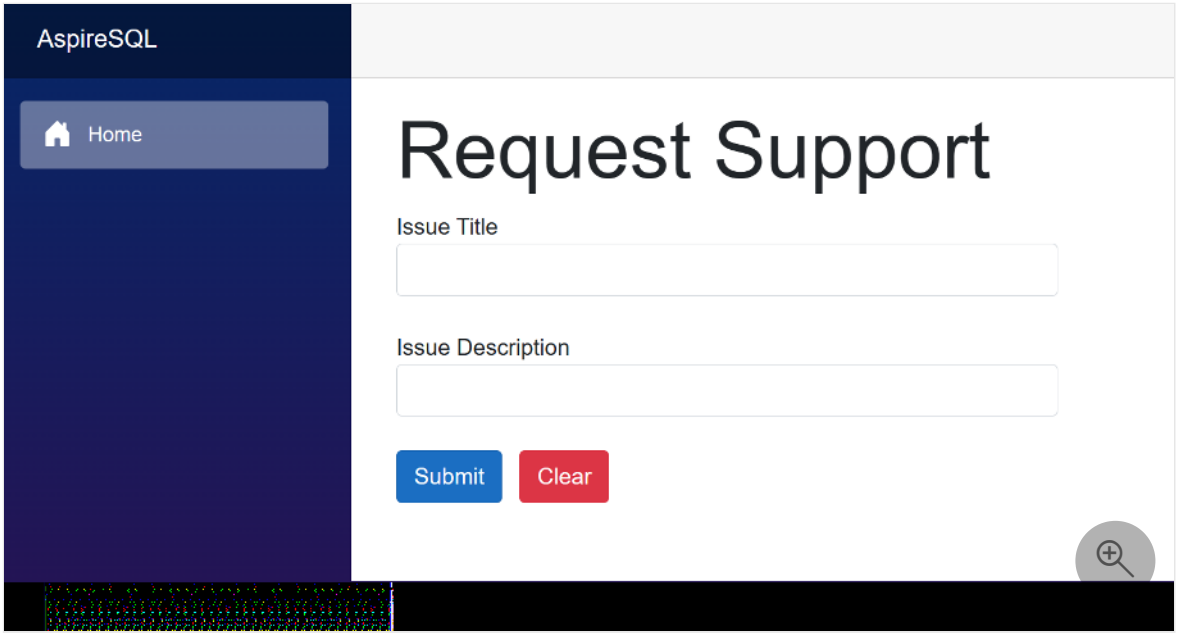
The preceding code adds a SQL Server Container resource to your app and configures a connection to a database called `sqldata`. The Entity Framework classes you configured earlier will automatically use this connection when migrating and connecting to the database.

Run and test the app locally

The sample app is now ready for testing. Verify that the submitted form data is persisted to the database by completing the following steps:

1. Select the run button at the top of Visual Studio (or `F5`) to launch your .NET Aspire project dashboard in the browser.

2. On the projects page, in the **AspireSQLEFCore** row, click the link in the **Endpoints** column to open the UI of your app.



3. Enter sample data into the `Title` and `Description` form fields.

4. Select the **Submit** button, and the form submits the support ticket for processing — (then select **Clear** to clear the form).

5. The data you submitted displays in the table at the bottom of the page when the page reloads.

6. Close the web browser tabs that display the **AspireSQL** web app and the .NET Aspire dashboard.

7. Switch to Visual Studio and, to stop debugging, select the stop button or press `Shift + F5`.

8. To start debugging a second time, select the run button at the top of Visual Studio (or `F5`).

9. In the .NET Aspire dashboard, on the projects page, in the **AspireSQLEFCore** row, click the link in the **Endpoints** column to open the UI of your app.

10. Notice that the page doesn't display the ticket you created in the previous run.
11. Close the web browser tabs that display the **AspireSQL** web app and the .NET Aspire dashboard.
12. Switch to Visual Studio and, to stop debugging, select the stop button or press `Shift + F5`.

Persist data across restarts

Developers often prefer their data to persist across restarts in the development environment for a more realistic database to run code against. To implement persistence in .NET Aspire, use the WithDataVolume method. This methods adds a Docker volume to your database container, which won't be destroyed every time you restart debugging.

1. In Visual Studio, in the *AspireSQLEFCore.AppHost* project, double-click the *Program.cs* code file.
2. Locate the following code:

```
C#

var sql = builder.AddSqlServer("sql")
                 .AddDatabase("sqldata");
```

3. Modify that code to match the following:

```
C#

var sql = builder.AddSqlServer("sql")
                 .WithDataVolume()
                 .AddDatabase("sqldata");
```

Run and test the data persistence

Let's examine how the data volume changes the behavior of the solution:

1. Select the run button at the top of Visual Studio (or `F5`) to launch your .NET Aspire project dashboard in the browser.
2. On the projects page, in the **AspireSQLEFCore** row, click the link in the **Endpoints** column to open the UI of your app.
3. Enter sample data into the `Title` and `Description` form fields.

4. Select the **Submit** button, and the form submits the support ticket for processing — (then select **Clear** to clear the form).
5. The data you submitted displays in the table at the bottom of the page when the page reloads.
6. Close the web browser tabs that display the **AspireSQL** web app and the .NET Aspire dashboard.
7. Switch to Visual Studio and, to stop debugging, select the stop button or press `Shift + F5`.
8. To start debugging a second time, select the run button at the top of Visual Studio (or `F5`).
9. In the .NET Aspire dashboard, on the projects page, in the **AspireSQLEFCore** row, click the link in the **Endpoints** column to open the UI of your app.
10. Notice that the page now displays the ticket you created in the previous run.

See also

- .NET Aspire with SQL Database deployment
- .NET Aspire deployment via Azure Container Apps
- Deploy a .NET Aspire project using GitHub Actions

Tutorial: Connect a .NET Aspire microservice to an existing database

Article • 03/28/2025

.NET Aspire is designed to make it easy and quick to develop cloud-native solutions. It uses containers to host the services, such as databases, that underpin each microservice. However, if you want your microservice to query a database that already exists, you must connect your microservice to it instead of creating a database container whenever you run the solution.

In this tutorial, you create a .NET Aspire solution with an API that connects to an existing database. You'll learn how to:

- ✓ Create an API microservice that interacts with a database.
- ✓ Configure the .NET Aspire App Host project with a connection string for the existing database.
- ✓ Pass the connection string to the API and use it to connect to the database.

Prerequisites

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0
- An OCI compliant container runtime, such as:
  - Docker Desktop or Podman. For more information, see Container runtime.
- An Integrated Developer Environment (IDE) or code editor, such as:
  - Visual Studio 2022 version 17.9 or higher (Optional)
  - Visual Studio Code (Optional)
    - C# Dev Kit: Extension (Optional)
  - JetBrains Rider with .NET Aspire plugin (Optional)

For more information, see .NET Aspire setup and tooling, and .NET Aspire SDK.

> **ⓘ Important**
>
> This tutorial also assumes you have a Microsoft SQL Server instance running on your local machine. You can connect to a database elsewhere by providing an appropriate connection string instead of the one suggested in this article. To create a new local instance, download and install SQL Server Developer Edition.

> **Tip**
>
> In this tutorial, you use .NET Aspire EF Core integrations to access the database. Other database integrations, which don't use EF Core, can use the same approach to connect to an existing database.

Create a new .NET Aspire solution

Let's start by creating a new solution with the .NET Aspire Starter template.

1. In Visual Studio, select **File** > **New** > **Project**.
2. In the **Create a new project** dialog window, search for *.NET Aspire*, select **.NET Aspire Starter App**, and then select **Next**.
3. In the **Configure your new project** page:
   - Enter a **Solution name** of **AspireExistingDB**.
   - Leave the other values at their defaults and then select **Next**.
4. In the **Additional information** page:
   - Make sure that **.NET 9.0** is selected.
   - Leave the other values at their defaults and then select **Create**.

Visual Studio creates a new .NET Aspire solution with an API and a web front end. The solution consists of the following projects:

- **AspireExistingDB.ApiService**: A web API project that returns weather forecasts.
- **AspireExistingDB.AppHost**: An orchestrator project designed to connect and configure the different projects and services of your app. The orchestrator should be set as the startup project.
- **AspireExistingDB.ServiceDefaults**: A shared class library to hold configurations that can be reused across the projects in your solution.
- **AspireExistingDB.Web**: A Blazor app that implements a web user interface for the solution.

Create the database model and context classes

First, install EF Core in the *AspireExistingDB.ApiService* project.

1. In **Solution Explorer**, right-click the *AspireExistingDB.ApiService* project, and then select **Manage NuGet Packages**.

1. Select the **Browse** tab, and then search for **Aspire.Microsoft.EntityFrameworkCore**.
2. Select the **Aspire.Microsoft.EntityFrameworkCore.SqlServer** package, and then select **Install**.

1. If the **Preview Changes** dialog appears, select **Apply**.
2. In the **License Acceptance** dialog, select **I Accept**.

To represent a weather report, add the following `WeatherReport` model class at the root of the *AspireExistingDB.ApiService* project:

C#

```
using System.ComponentModel.DataAnnotations;

namespace AspireExistingDB.ApiService;

public sealed class WeatherReport
{
    public int Id { get; set; }
    [Required]
    public DateTime Date { get; set; } = DateTime.Now;
    [Required]
    public int TemperatureC { get; set; } = 10;
    public string? Summary { get; set; }
}
```

Add the following `WeatherDbContext` data context class at the root of the *AspireExistingDB.ApiService* project. The class inherits System.Data.Entity.DbContext to work with EF Core and represent your database.

C#

```
using Microsoft.EntityFrameworkCore;

namespace AspireExistingDB.ApiService;

public class WeatherDbContext(DbContextOptions options) : DbContext(options)
{
    public DbSet<WeatherReport> Forecasts => Set<WeatherReport>();
}
```

Add the Scalar user interface to the API project

You'll use the Scalar UI to test the *AspireExistingDB.ApiService* project. Let's install and configure it:

1. In Visual Studio, in the **Solution Explorer**, right-click the *AspireExistingDB.ApiService* project, and then select **Manage NuGet Packages**.
2. Select the **Browse** tab, and then search for **Scalar**.
3. Select the **Scalar.AspNetCore** package, and then select **Install**.
4. If the **Preview Changes** dialog appears, select **Apply**.
5. In the **License Acceptance** dialog, select **I Accept**.
6. In the *AspireExistingDB.ApiService* project, open the *Program.cs* file.
7. Locate the following lines of code:

```
C#

if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
}
```

8. Modify that code to match the following lines:

```
C#

if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
    app.MapScalarApiReference(_ => _.Servers = [ ]);
}
```

Configure a connection string in the App Host project

Usually, when you create a cloud-native solution with .NET Aspire, you call the AddSqlServer method to initiate a container that runs the SQL Server instance. You pass that resource to other projects in your solution that need access to the database.

In this case, however, you want to work with an existing database outside of any container. There are three differences in the App Host project:

- You don't need to install the `Aspire.Hosting.SqlServer` hosting integration.
- You add a connection string in a configuration file, such as **appsetting.json**.
- You call AddConnectionString to create a resource that you pass to other projects. Those projects use this resource to connect to the existing database.

Let's implement that configuration:

1. In Visual Studio, in the *AspireExistingDB.AppHost* project, open the *appsetting.json* file.

2. Replace the entire contents of the file with the following code:

   JSON

   ```
   {
       "ConnectionStrings": {
           "sql":
   "Server=localhost;Trusted_Connection=True;TrustServerCertificate=True;I
   nitial Catalog=WeatherForecasts;"
       },
       "Logging": {
           "LogLevel": {
               "Default": "Information",
               "Microsoft.AspNetCore": "Warning",
               "Aspire.Hosting.Dcp": "Warning"
           }
       }
   }
   ```

3. In the *AspireExistingDB.AppHost* project, open the *Program.cs* file.

4. Locate the following line of code:

   C#

   ```
   var builder = DistributedApplication.CreateBuilder(args);
   ```

5. Immediately after that line, add this line of code, which obtains the connection string from the configuration file:

   C#

   ```
   var connectionString = builder.AddConnectionString("sql");
   ```

1. Locate the following line of code, which creates a resource for the *AspireExistingDB.ApiService* project:

```C#
var apiService =
builder.AddProject<Projects.AspireExistingDB_ApiService>("apiservice");
```

2. Modify that line to match the following, which creates the resource and passes the connection string to it:

```C#
var apiService =
builder.AddProject<Projects.AspireExistingDB_ApiService>("apiservice")
                        .WithReference(connectionString);
```

3. To save your changes, select **File** > **Save All**.

Use the database in the API project

Returning to the *AspireExistingDB.ApiService* project, you must obtain the connection string resource from the App Host, and then use it to create the database:

1. In Visual Studio, in the *AspireExistingDB.ApiService* project, open the *Program.cs* file.

2. Locate the following line of code:

```C#
var builder = WebApplication.CreateBuilder(args);
```

1. Immediately after that line, add this line of code:

```C#
builder.AddSqlServerDbContext<WeatherDbContext>("sql");
```

1. Locate the following line of code:

```C#
app.MapDefaultEndpoints();
```

mediately after that line, add this code, which connects to the database and
eates it if it doesn't already exist:

```
C#

if (app.Environment.IsDevelopment())
{
    using (var scope = app.Services.CreateScope())
    {
        var context =
scope.ServiceProvider.GetRequiredService<WeatherDbContext>();
        context.Database.EnsureCreated();
    }
}
else
{
    app.UseExceptionHandler("/Error", createScopeForErr
    app.UseHsts();
}
```

eceding code:

hecks if the app is running in a development
it is, it retrieves the `WeatherDbContext` ser
`atabase.EnsureCreated()` to create the

lote

Note tha

.NET Aspire starter solution template, the API creates five random weather
sts and returns them when another project requests them. Let's replace that with
hat queries the database:

n Visual Studio, in the *AspireExistingDB.ApiService* project, open the *Program.cs*
ile.

t the top of the file, add the following lines of code:

```
C#

using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
```

3. Locate the following code:

```
C#

string[] summaries = ["Freezing", "Bracing", "Chilly", "Cool", "Mild",
"Warm", "Balmy", "Hot", "Sweltering", "Scorching"];

app.MapGet("/weatherforecast", () =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        (
            DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            Random.Shared.Next(-20, 55),
            summaries[Random.Shared.Next(summaries.Length)]
        ))
        .ToArray();
    return forecast;
})
.WithName("GetWeatherForecast");
```

4. Replace that code with the following lines:

```
C#

app.MapGet("/weatherforecast", async ([FromServices] WeatherDbContext
context) =>
{
    var forecast = await context.Forecasts.ToArrayAsync();
    return forecast;
})
.WithName("GetWeatherForecast");
```

5. Locate and remove the following code:

```
C#

record WeatherForecast(DateOnly Date, int TemperatureC, string?
Summary)
{
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
}
```

Add code to insert a new forecast into the database

Finally, let's add a POST method to the API, which will add records to the database:

1. In Visual Studio, in the *Program.cs* file for the *AspireExistingDB.ApiService* project, locate the following code:

   ```
   C#

   app.MapDefaultEndpoints();
   ```

2. Immediately *before* that line, add the following code, which creates and saves a new forecast:

   ```
   C#

   app.MapPost("/weatherforecast", async ([FromBody] WeatherReport
   forecast, [FromServices] WeatherDbContext context, HttpResponse
   response) =>
   {
       context.Forecasts.Add(forecast);
       await context.SaveChangesAsync();
       response.StatusCode = 200;
       response.Headers.Location = $"weatherforecast/{forecast.Id}";
   })
   .Accepts<WeatherReport>("application/json")
   .Produces<WeatherReport>(StatusCodes.Status201Created)
   .WithName("PostWeatherForecast").WithTags("Setters");
   ```

Run and test the app locally

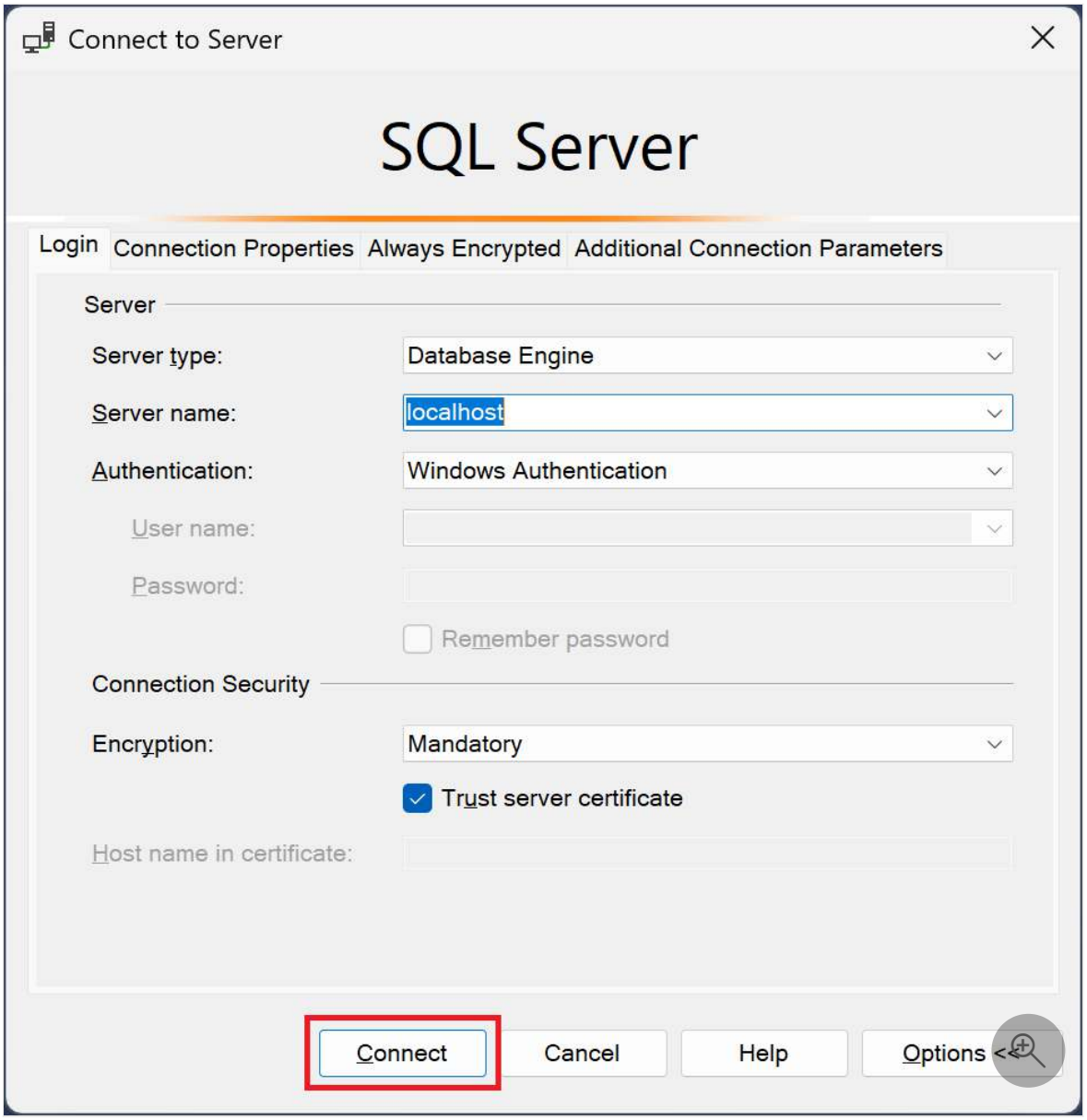
The sample app is ready to test. Before you start debugging, make sure that:

- Docker Desktop or Podman is running to host containers for the solution.
- SQL Server is running to host the database.

Let's connect to the SQL Server instance and check the databases that exist.

1. Start **Microsoft SQL Server Management Studio**.

2. Connect to the SQL Server instance on the local machine. Ensure that **Trust server certificate** is selected.



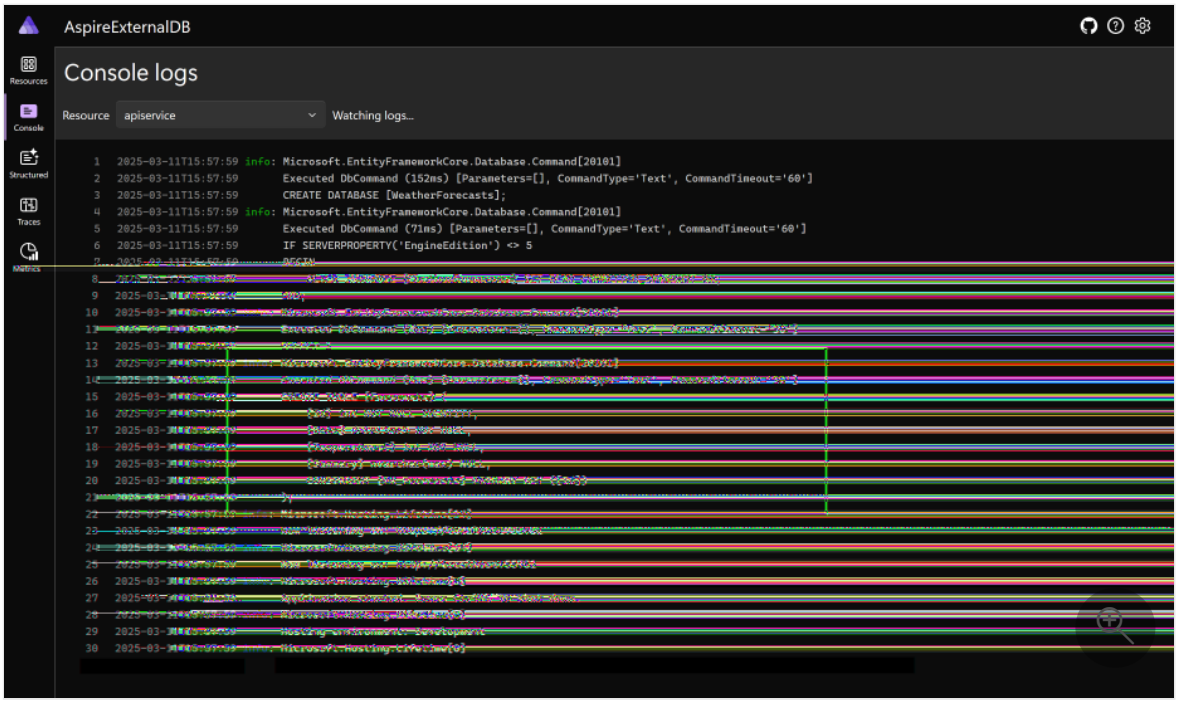
3. In the **Object Explorer** expand **Databases**. There is no database named **WeatherForecasts**.

Now, let's test the solution:

1. In Visual Studio, select the run button (or press `F5`) to launch your .NET Aspire project dashboard in the browser.

2. In the navigation on the left, select **Console**.

3. In the **Resource** drop down list, select **apiservice**. Notice the `CREATE TABLE` SQL command, which has created the **Forecasts** table in the **WeatherForecasts** database.



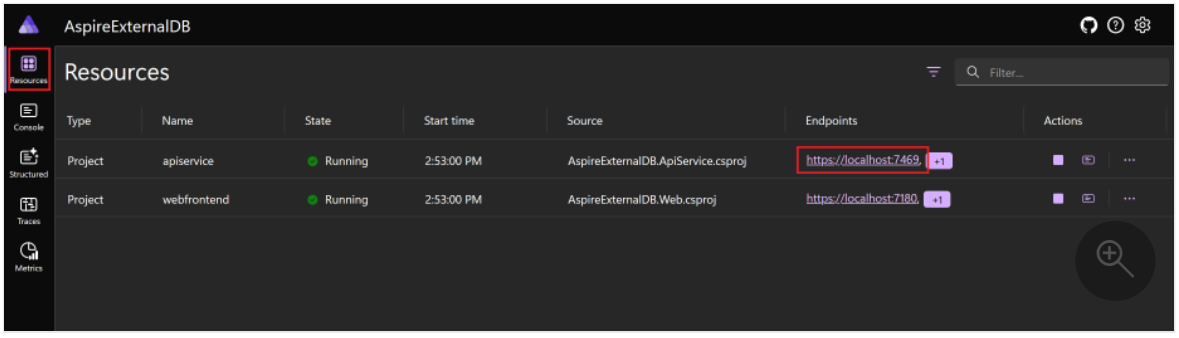
4. Switch to SQL Server Management Studio. In the **Object Explorer**, right-click **Databases** and then select **Refresh**.

5. Expand the new **WeatherForecasts** database and then expand **Tables**. Notice the new **dbo.Forecasts** table.

6. Right-click the **dbo.Forecasts** table and then select **Select Top 1000 Rows**. The query runs but returns no results because the table is empty.

7. In the .NET Aspire dashboard, in the navigation on the left, select **Resources**.

8. Select one of the endpoints for the **apiservice** resource.



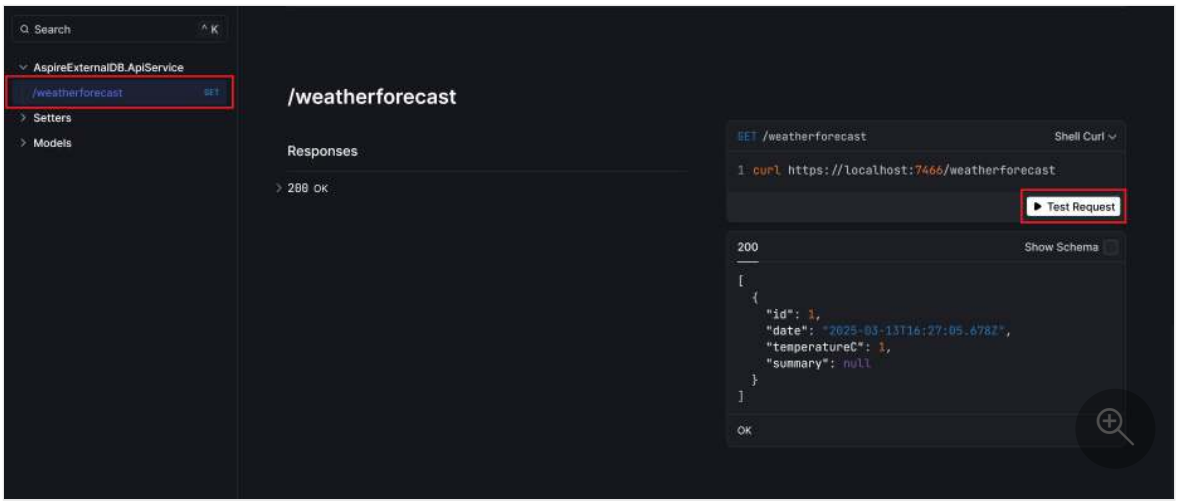
9. In the browser window, append **/scalar** to the web address and then press `Enter`.

10. In the navigation on the left, expand **Setters** and then select **/weatherforecast POST**.

11. Select **Test Request**. Under **Body**, in the **JSON** window, delete the **id** and **date** lines and fill in your own values for **temperatureC** and **summary**.

12. Select **Send**.

13. In the navigation on the left, select **/weatherforecast GET** and then select **Test Request**.



14. Select **Send**. The call should return JSON with the weather reported you posted.

15. Switch to SQL Server Management Studio. In the query window for the **Forecasts** table, select **Execute** or press `F5`. Your weather forecast is displayed.

See also

- .NET Aspire SQL Server Entity Framework Core integration
- Tutorial: Connect an ASP.NET Core app to SQL Server using .NET Aspire and Entity Framework Core
- Use openAPI documents

Tutorial: Connect an ASP.NET Core app to .NET Aspire storage integrations

Article • 11/08/2024

Cloud-native apps often require scalable storage solutions that provide capabilities like blob storage, queues, or semi-structured NoSQL databases. .NET Aspire integrations simplify connections to various storage services, such as Azure Blob Storage. In this tutorial, you'll create an ASP.NET Core app that uses .NET Aspire integrations to connect to Azure Blob Storage and Azure Queue Storage to submit support tickets. The app sends the tickets to a queue for processing and uploads an attachment to storage. You'll learn how to:

- ✓ Create a basic .NET app that is set up to use .NET Aspire integrations
- ✓ Add .NET Aspire integrations to connect to multiple storage services
- ✓ Configure and use .NET Aspire Component features to send and receive data

Prerequisites

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0
- An OCI compliant container runtime, such as:
  - Docker Desktop or Podman. For more information, see Container runtime.
- An Integrated Developer Environment (IDE) or code editor, such as:
  - Visual Studio 2022 version 17.9 or higher (Optional)
  - Visual Studio Code (Optional)
    - C# Dev Kit: Extension (Optional)
  - JetBrains Rider with .NET Aspire plugin (Optional)

For more information, see .NET Aspire setup and tooling, and .NET Aspire SDK.

Explore the completed sample app

A completed version of the sample app from this tutorial is available on GitHub. The project is also structured as a template for the Azure Developer CLI, meaning you can use the `azd up` command to automate Azure resource provisioning if you have the tool installed.

Bash

```
git clone https://github.com/Azure-Samples/dotnet-aspire-connect-storage.git
```

Set up the Azure Storage resources

For this article, you'll need to create a blob container and storage queue resource in your local development environment using an emulator. To do so, use Azurite. Azurite is a free, open source, cross-platform Azure Storage API compatible server (emulator) that runs in a Docker container.

To use the emulator you need to install Azurite.

Create the sample solution

Create a .NET Aspire project using either Visual Studio or the .NET CLI.

Visual Studio

1. At the top of Visual Studio, navigate to **File** > **New** > **Project**.
2. In the dialog window, search for *Aspire* and select **.NET Aspire Starter Application**. Choose **Next**.
3. On the **Configure your new project** screen:
    - Enter a **Solution Name** of **AspireStorage** and select **Next**.
4. On the **Additional information** screen:
    - Uncheck **Use Redis for caching** (not required for this tutorial).
    - Select **Create**.

Visual Studio creates a new ASP.NET Core solution that is structured to use .NET Aspire.

The solution consists of the following projects:

- **AspireStorage.ApiService** - An API project with default .NET Aspire service configurations.
- **AspireStorage.AppHost** - An orchestrator project designed to connect and configure the different projects and services of your app. The orchestrator should be set as the startup project.
- **AspireStorage.ServiceDefaults** - A shared class library to hold code that can be reused across the projects in your solution.

- **AspireStorage.Web** - A Blazor Server project that serves as the front end of your app.

Add the Worker Service project

Next, add a Worker Service project to the solution to retrieve and process messages as they are added to the Azure Storage queue.

Visual Studio

1. In the solution explorer, right click on the top level *AspireStorage* solution node and select **Add** > **New project**.
2. Search for and select the **Worker Service** template and choose **Next**.
3. For the **Project name**, enter *AspireStorage.WorkerService* and select **Next**.
4. On the **Additional information** screen:
    - Make sure **.NET 9.0** is selected.
    - Make sure **Enlist in .NET Aspire orchestration** is checked and select **Create**.

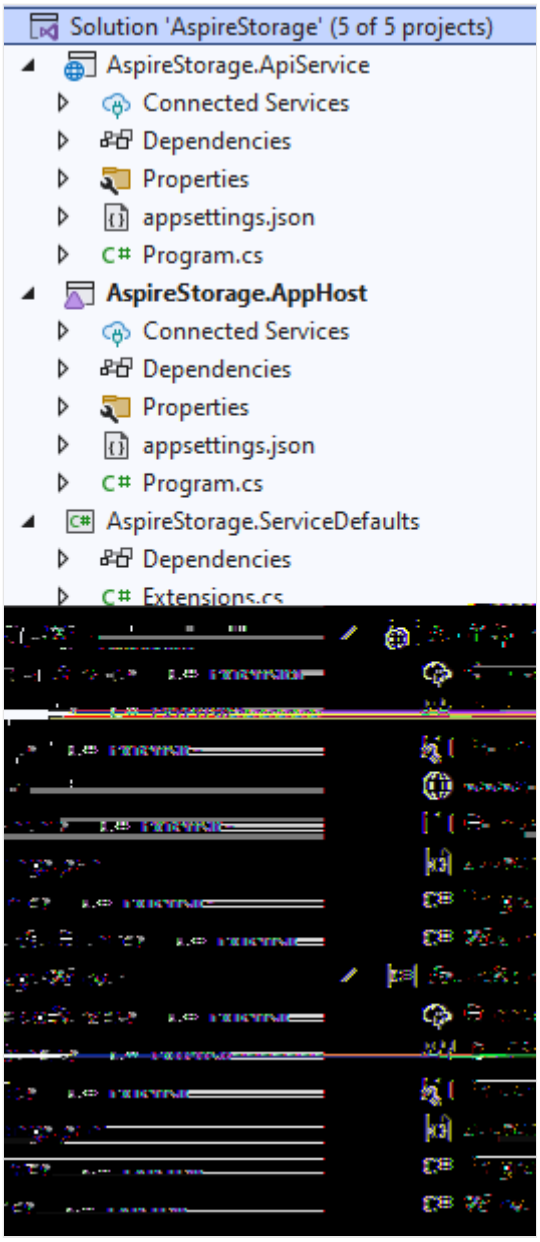
Visual Studio adds the project to your solution and updates the *Program.cs* file of the *AspireStorage.AppHost* project with a new line of code:

C#

```
builder.AddProject<Projects.AspireStorage_WorkerService>(
    "aspirestorage-workerservice");
```

Visual Studio tooling added this line of code to register your new project with the IDistributedApplicationBuilder object, which enables orchestration features. For more information, see .NET Aspire orchestration overview.

The completed solution structure should resemble the following:



Add the .NET Aspire integrations to the Blazor app

Add the .NET Aspire Azure Blob Storage integration and .NET Aspire Azure Queue Storage integration packages to your *AspireStorage.Web* project:

.NET CLI

```
dotnet add package Aspire.Azure.Storage.Blobs
dotnet add package Aspire.Azure.Storage.Queues
```

Your **AspireStorage.Web** project is now set up to use .NET Aspire integrations. Here's the updated *AspireStorage.Web.csproj* file:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference
Include="..\AspireStorage.ServiceDefaults\AspireStorage.ServiceDefaults.cspr
oj" />
  </ItemGroup>

  <ItemGroup>
    <PackageReference Include="Aspire.Azure.Storage.Blobs" Version="9.1.0"
/>
    <PackageReference Include="Aspire.Azure.Storage.Queues" Version="9.1.0"
/>
  </ItemGroup>

</Project>
```

The next step is to add the integrations to the app.

In the *Program.cs* file of the *AspireStorage.Web* project, add calls to the AddAzureBlobClient and AddAzureQueueClient extension methods after the creation of the `builder` but before the call to `AddServiceDefaults`. For more information, see .NET Aspire service defaults. Provide the name of your connection string as a parameter.

C#

```
using AspireStorage.Web;
using AspireStorage.Web.Components;

using Azure.Storage.Blobs;
using Azure.Storage.Queues;

var builder = WebApplication.CreateBuilder(args);

builder.AddAzureBlobClient("BlobConnection");
builder.AddAzureQueueClient("QueueConnection");

// Add service defaults & Aspire components.
builder.AddServiceDefaults();

// Add services to the container.
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();

builder.Services.AddOutputCache();
```

```
builder.Services.AddHttpClient<WeatherApiClient>(client =>
    {
        // This URL uses "https+http://" to indicate HTTPS is preferred over
HTTP.
        // Learn more about service discovery scheme resolution at
https://aka.ms/dotnet/sdschemes.
        client.BaseAddress = new("https+http://apiservice");
    });

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}
else
{
    // In development, create the blob container and queue if they don't
exist.
    var blobService = app.Services.GetRequiredService<BlobServiceClient>();
    var docsContainer = blobService.GetBlobContainerClient("fileuploads");

    await docsContainer.CreateIfNotExistsAsync();

    var queueService = app.Services.GetRequiredService<QueueServiceClient>
();
    var queueClient = queueService.GetQueueClient("tickets");

    await queueClient.CreateIfNotExistsAsync();
}

app.UseHttpsRedirection();

app.UseStaticFiles();
app.UseAntiforgery();

app.UseOutputCache();

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();

app.MapDefaultEndpoints();

app.Run();
```

With the additional `using` statements, these methods accomplish the following tasks:

- Register a Azure.Storage.Blobs.BlobServiceClient and a Azure.Storage.Queues.QueueServiceClient with the DI container for connecting to

Azure Storage.
- Automatically enable corresponding health checks, logging, and telemetry for the respective services.

When the *AspireStorage.Web* project starts, it will create a `fileuploads` container in Azurite Blob Storage and a `tickets` queue in Azurite Queue Storage. This is conditional when the app is running in a development environment. When the app is running in a production environment, the container and queue are assumed to have already been created.

Add the .NET Aspire integration to the Worker Service

The worker service handles pulling messages off of the Azure Storage queue for processing. Add the .NET Aspire Azure Queue Storage integration integration package to your *AspireStorage.WorkerService* app:

.NET CLI

```
dotnet add package Aspire.Azure.Storage.Queues
```

In the *Program.cs* file of the *AspireStorage.WorkerService* project, add a call to the AddAzureQueueClient extension method after the creation of the `builder` but before the call to `AddServiceDefaults`:

C#

```
using AspireStorage.WorkerService;

var builder = Host.CreateApplicationBuilder(args);

builder.AddAzureQueueClient("QueueConnection");

builder.AddServiceDefaults();
builder.Services.AddHostedService<WorkerService>();

var host = builder.Build();
host.Run();
```

This method handles the following tasks:

- Register a QueueServiceClient with the DI container for connecting to Azure Storage Queues.

- Automatically enable corresponding health checks, logging, and telemetry for the respective services.

Create the form

The app requires a form for the user to be able to submit support ticket information and upload an attachment. The app uploads the attached file on the `Document` (IFormFile) property to Azure Blob Storage using the injected BlobServiceClient. The QueueServiceClient sends a message composed of the `Title` and `Description` to the Azure Storage Queue.

Use the following Razor markup to create a basic form, replacing the contents of the *Home.razor* file in the *AspireStorage.Web/Components/Pages* directory:

razor

```razor
@page "/"

@using System.ComponentModel.DataAnnotations
@using Azure.Storage.Blobs
@using Azure.Storage.Queues

@inject BlobServiceClient BlobClient
@inject QueueServiceClient QueueServiceClient

<PageTitle>Home</PageTitle>

<div class="text-center">
    <h1 class="display-4">Request Support</h1>
</div>

<EditForm Model="@Ticket" FormName="Tickets" method="post"
          OnValidSubmit="@HandleValidSubmit" enctype="multipart/form-data">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <div class="mb-4">
        <label>Issue Title</label>
        <InputText class="form-control" @bind-Value="@Ticket.Title" />
        <ValidationMessage For="() => Ticket.Title" />
    </div>
    <div class="mb-4">
        <label>Issue Description</label>
        <InputText class="form-control" @bind-Value="@Ticket.Description" />
        <ValidationMessage For="() => Ticket.Description" />
    </div>
    <div class="mb-4">
        <label>Attachment</label>
        <InputFile class="form-control" name="Ticket.Document" />
        <ValidationMessage For="() => Ticket.Document" />
```

```
    </div>
    <button class="btn btn-primary" type="submit">Submit</button>
    <button class="btn btn-danger mx-2" type="reset"
@onclick=@ClearForm>Clear</button>
</EditForm>

@code {
    [SupplyParameterFromForm(FormName = "Tickets")]
    private SupportTicket Ticket { get; set; } = new();

    private async Task HandleValidSubmit()
    {
        var docsContainer =
BlobClient.GetBlobContainerClient("fileuploads");

        // Upload file to blob storage
        await docsContainer.UploadBlobAsync(
            Ticket.Document.FileName,
            Ticket.Document.OpenReadStream());

        // Send message to queue
        var queueClient = QueueServiceClient.GetQueueClient("tickets");

        await queueClient.SendMessageAsync(
             $"{Ticket.Title} - {Ticket.Description}");

        ClearForm();
    }

    private void ClearForm() => Ticket = new();

    private class SupportTicket()
    {
        [Required] public string Title { get; set; } = default!;
        [Required] public string Description { get; set; } = default!;
        [Required] public IFormFile Document { get; set; } = default!;
    }
}
```

For more information about creating forms in Blazor, see ASP.NET Core Blazor forms overview.

Update the AppHost

The *AspireStorage.AppHost* project is the orchestrator for your app. It's responsible for connecting and configuring the different projects and services of your app. The orchestrator should be set as the startup project.

To add Azure Storage hosting support to your IDistributedApplicationBuilder, install the Aspire.Hosting.Azure.Storage NuGet package.

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Storage
```

Replace the contents of the *Program.cs* file in the *AspireStorage.AppHost* project with the following code:

C#

```
using Microsoft.Extensions.Hosting;

var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("Storage");

if (builder.Environment.IsDevelopment())
{
    storage.RunAsEmulator();
}

var blobs = storage.AddBlobs("BlobConnection");
var queues = storage.AddQueues("QueueConnection");

var apiService = builder.AddProject<Projects.AspireStorage_ApiService>
("apiservice");

builder.AddProject<Projects.AspireStorage_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(apiService)
    .WithReference(blobs)
    .WithReference(queues);

builder.AddProject<Projects.AspireStorage_WorkerService>("aspirestorage-
workerservice")
    .WithReference(queues);

builder.Build().Run();
```

The preceding code adds Azure storage, blobs, and queues, and when in development mode, it uses the emulator. Each project defines references for these resources that they depend on.

Process the items in the queue

When a new message is placed on the `tickets` queue, the worker service should retrieve, process, and delete the message. Update the *Worker.cs* class, replacing the contents with the following code:

C#

```
using Azure.Storage.Queues;
using Azure.Storage.Queues.Models;

namespace AspireStorage.WorkerService;

public sealed class WorkerService(
    QueueServiceClient client,
    ILogger<WorkerService> logger) : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        var queueClient = client.GetQueueClient("tickets");
        await queueClient.CreateIfNotExistsAsync(cancellationToken:
stoppingToken);

        while (!stoppingToken.IsCancellationRequested)
        {
            QueueMessage[] messages =
                await queueClient.ReceiveMessagesAsync(
                    maxMessages: 25, cancellationToken: stoppingToken);

            foreach (var message in messages)
            {
                logger.LogInformation(
                    "Message from queue: {Message}", message.MessageText);

                await queueClient.DeleteMessageAsync(
                    message.MessageId,
                    message.PopReceipt,
                    cancellationToken: stoppingToken);
            }

            // TODO: Determine an appropriate time to wait
            // before checking for more messages.
            await Task.Delay(TimeSpan.FromSeconds(15), stoppingToken);
        }
    }
}
```

Before the worker service can process messages, it needs to be able to connect to the Azure Storage queue. With Azurite, you need to ensure that the queue is available before the worker service starts executing message queue processing.

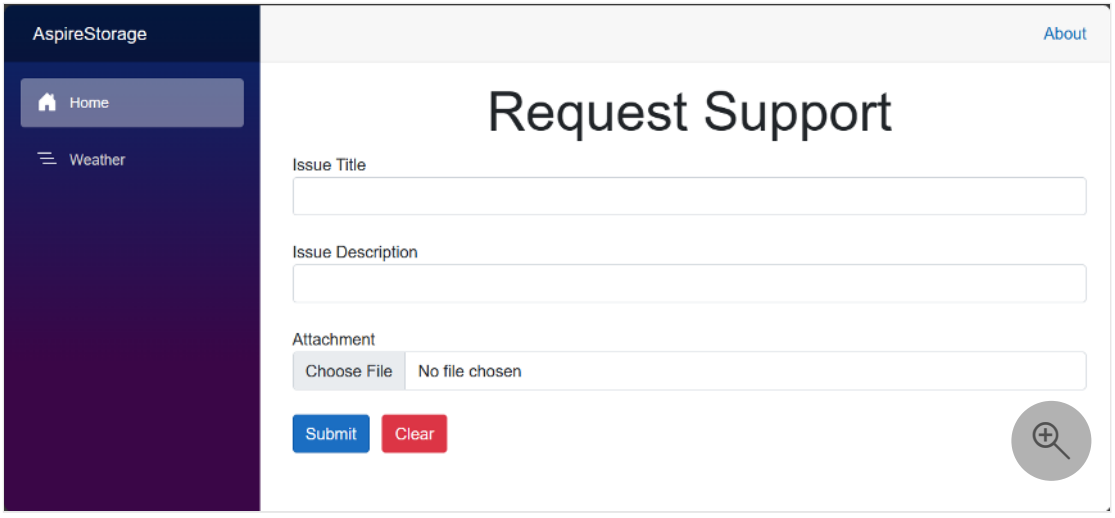
The worker service processes message in the queue and deletes them when they've been processed.

Run and test the app locally

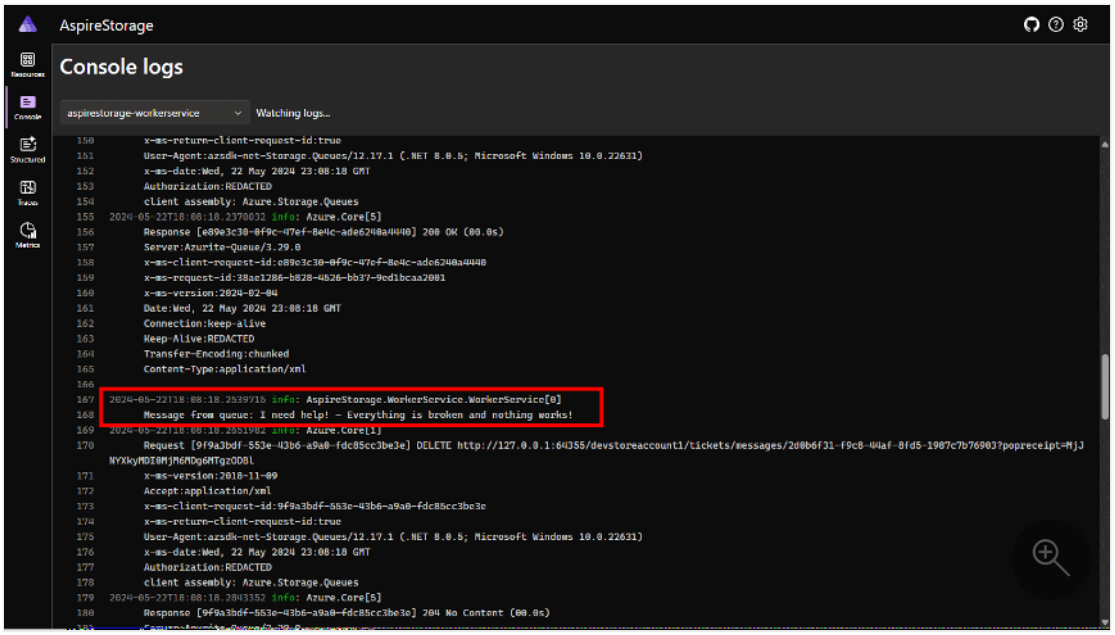
The sample app is now ready for testing. Verify that the submitted form data is sent to Azure Blob Storage and Azure Queue Storage by completing the following steps:

Visual Studio

1. Press the run button at the top of Visual Studio to launch your .NET Aspire project dashboard in the browser.
2. On the resources page, in the **aspirestorage.web** row, click the link in the **Endpoints** column to open the UI of your app.



3. Enter sample data into the `Title` and `Description` form fields and select a simple file to upload.
4. Select the **Submit** button, and the form submits the support ticket for processing — and clears the form.
5. In a separate browser tab, use the Azure portal to navigate to the **Storage browser** in your Azure Storage Account.
6. Select **Containers** and then navigate into the **Documents** container to see the uploaded file.
7. You can verify the message on the queue was processed by looking at the **Project logs** of the .NET Aspire dashboard, and selecting the **aspirestorage.workerservice** from the dropdown.



Summary

The example app that you built demonstrates persisting blobs from an ASP.NET Core Blazor Web App and processing queues in a .NET Worker Service. Your app connects to Azure Storage using .NET Aspire integrations. The app sends the support tickets to a queue for processing and uploads an attachment to storage.

Since you choose to use Azurite, there's no need to clean up these resources when you're done testing them, as you created them locally in the context of an emulator. The emulator enabled you to test your app locally without incurring any costs, as no Azure resources were provisioned or created.

Tutorial: Use .NET Aspire messaging integrations in ASP.NET Core

Article • 03/20/2025

Cloud-native apps often require scalable messaging solutions that provide capabilities such as messaging queues and topics and subscriptions. .NET Aspire integrations simplify the process of connecting to various messaging providers, such as Azure Service Bus. In this tutorial, you'll create an ASP.NET Core app that uses .NET Aspire integrations to connect to Azure Service Bus to create a notification system. Submitted messages will be sent to a Service Bus topic for consumption by subscribers. You'll learn how to:

- ✓ Create a basic .NET app that is set up to use .NET Aspire integrations
- ✓ Add an .NET Aspire integration to connect to Azure Service Bus
- ✓ Configure and use .NET Aspire integration features to send and receive data

Prerequisites

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0
- An OCI compliant container runtime, such as:
  - Docker Desktop or Podman. For more information, see Container runtime.
- An Integrated Developer Environment (IDE) or code editor, such as:
  - Visual Studio 2022 version 17.9 or higher (Optional)
  - Visual Studio Code (Optional)
    - C# Dev Kit: Extension (Optional)
  - JetBrains Rider with .NET Aspire plugin (Optional)

For more information, see .NET Aspire setup and tooling, and .NET Aspire SDK.

In addition to the preceding prerequisites, you also need to install the Azure CLI. To install the Azure CLI, follow the instructions in the Azure CLI installation guide.

Set up the Azure Service Bus account

For this tutorial, you'll need access to an Azure Service Bus namespace with a topic and subscription configured. Use one of the following options to set up the require resources:

- **Azure portal**: Create a service bus account with a topic and subscription.

Alternatively:

- **Azure CLI**: Run the following commands in the Azure CLI or CloudShell to set up the required Azure Service Bus resources:

  Azure CLI

  ```
  az group create -n <your-resource-group-name> --location eastus
  az servicebus namespace create -g <your-resource-group-name> --name
  <your-namespace-name> --location eastus
  az servicebus topic create -g <your-resource-group-name> --namespace-
  name <your-namespace-name> --name notifications
  az servicebus topic subscription create -g <your-resource-group-name> -
  -namespace-name <your-namespace-name> --topic-name notifications --name
  mobile
  ```

  > **Note**
  >
  > Replace the **your-resource-group-name** and **your-namespace-name** placeholders with your own values. Service Bus namespace names must be globally unique across Azure.

Azure authentication

This quickstart can be completed using either passwordless authentication or a connection string. Passwordless connections use Azure Active Directory and role-based access control (RBAC) to connect to a Service Bus namespace. You don't need to worry about having hard-coded connection string in your code, a configuration file, or in secure storage such as Azure Key Vault.

You can also use a connection string to connect to a Service Bus namespace, but the passwordless approach is recommended for real-world applications and production environments. For more information, read about Authentication and authorization or visit the passwordless overview page.

Passwordless (Recommended)

On your Service Bus namespace, assign the following role to the user account you logged into Visual Studio or the Azure CLI with:

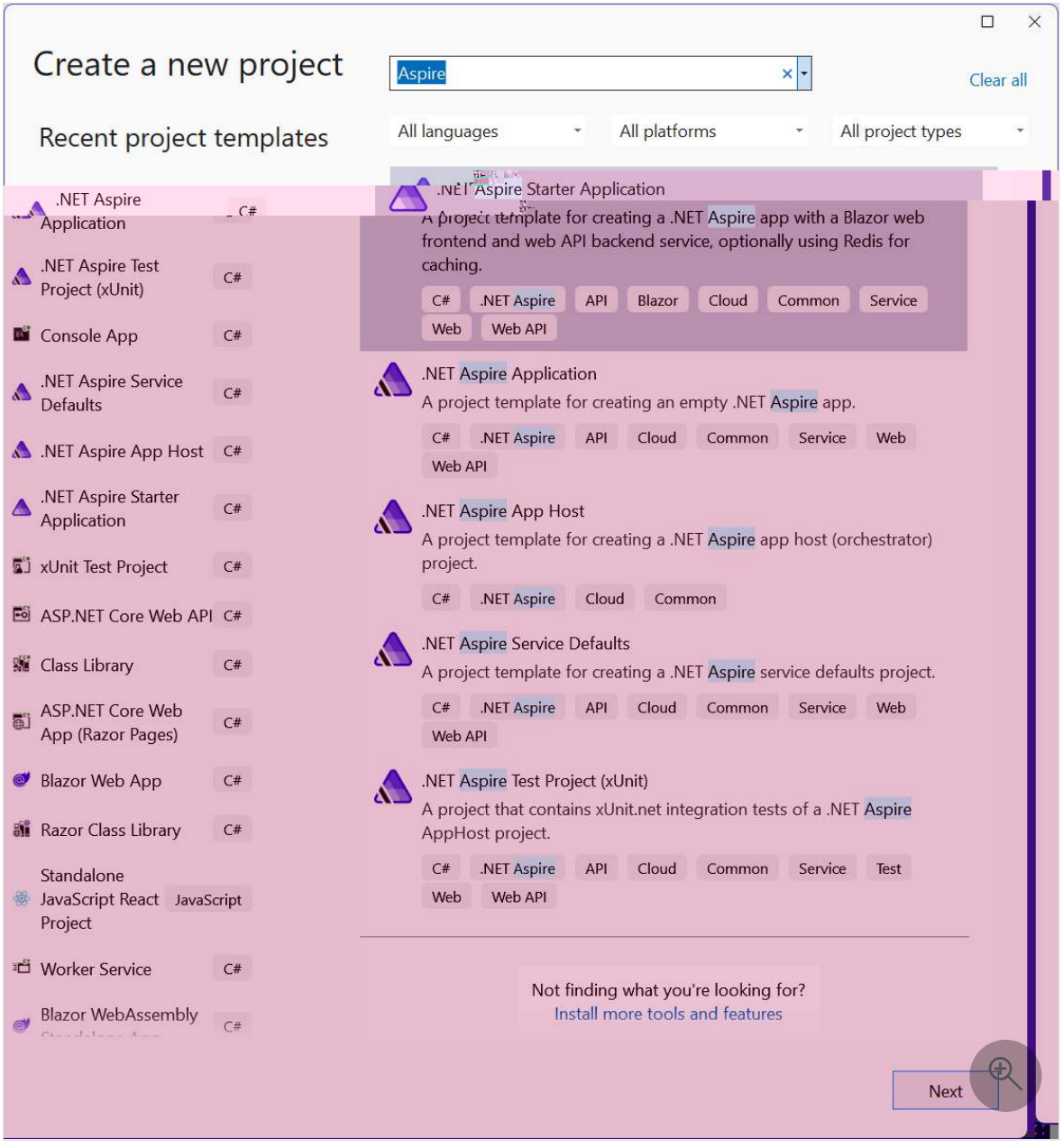
- Service Bus Data Owner: Assign an Azure RBAC role

Create the sample solution

To create a new .NET Aspire Starter Application, you can use either Visual Studio, Visual Studio Code, or the .NET CLI.

Visual Studio provides .NET Aspire templates that handle some initial setup configurations for you. Complete the following steps to create a project for this quickstart:

1. At the top of Visual Studio, navigate to **File** > **New** > **Project**.
2. In the dialog window, search for *Aspire* and select **.NET Aspire Starter App**. Select **Next**.



3. On the **Configure your new project** screen:

    - Enter a **Project Name** of *AspireSample*.
    - Leave the rest of the values at their defaults and select **Next**.

4. On the **Additional information** screen:

    - Make sure **.NET 9.0 (Standard Term Support)** is selected.
    - Ensure that **Use Redis for caching (requires a supported container runtime)** is checked and select **Create**.
    - Optionally, you can select **Create a tests project**. For more information, see Write your first .NET Aspire test.

Visual Studio creates a new solution that is structured to use .NET Aspire.

Add the Worker Service project

Next, add a Worker Service project to the solution to retrieve and process messages to and from Azure Service Bus.

1. In the solution explorer, right click on the top level `AspireSample` solution node and select **Add** > **New project**.
2. Search for and select the **Worker Service** template and choose **Next**.
3. For the **Project name**, enter *AspireSample.WorkerService* and select **Next**.
4. On the **Additional information** screen:

    - Make sure **.NET 9.0** is selected.
    - Make sure **Enlist in .NET Aspire orchestration** is checked and select **Create**.

Visual Studio adds the project to your solution and updates the *Program.cs* file of the `AspireSample.AppHost` project with a new line of code:

C#

```
builder.AddProject<Projects.AspireSample_WorkerService>(
    "aspiresample-workerservice");
```

Visual Studio tooling added this line of code to register your new project with the IDistributedApplicationBuilder object, which enables orchestration features you'll explore later.

The completed solution structure should resemble the following, assuming the top-level directory is named *aspire-messaging*:

Directory

```
└── aspire-messaging
    ├── AspireSample.WorkerService
    │   ├── Properties
    │   │   └── launchSettings.json
    │   ├── appsettings.Development.json
    │   ├── appsettings.json
    │   ├── AspireSample.WorkerService.csproj
    │   ├── Program.cs
    │   └── Worker.cs
    ├── AspireSample.ApiService
    │   ├── Properties
    │   │   └── launchSettings.json
    │   ├── appsettings.Development.json
    │   ├── appsettings.json
    │   ├── AspireSample.ApiService.csproj
    │   └── Program.cs
    ├── AspireSample.AppHost
    │   ├── Properties
    │   │   └── launchSettings.json
    │   ├── appsettings.Development.json
    │   ├── appsettings.json
    │   ├── AspireSample.AppHost.csproj
    │   └── Program.cs
    ├── AspireSample.ServiceDefaults
    │   ├── AspireSample.ServiceDefaults.csproj
    │   └── Extensions.cs
    ├── AspireSample.Web
    │   ├── Components
    │   │   ├── Layout
    │   │   │   ├── MainLayout.razor
    │   │   │   ├── MainLayout.razor.css
    │   │   │   ├── NavMenu.razor
    │   │   │   └── NavMenu.razor.css
    │   │   ├── Pages
    │   │   │   ├── Counter.razor
    │   │   │   ├── Error.razor
    │   │   │   ├── Home.razor
    │   │   │   └── Weather.razor
    │   │   ├── _Imports.razor
    │   │   ├── App.razor
    │   │   └── Routes.razor
    │   ├── Properties
    │   │   └── launchSettings.json
    │   ├── wwwroot
    │   │   ├── bootstrap
    │   │   │   ├── bootstrap.min.css
    │   │   │   └── bootstrap.min.css.map
    │   │   ├── app.css
    │   │   └── favicon.png
    │   ├── appsettings.Development.json
    │   ├── appsettings.json
    │   └── AspireSample.Web.csproj
```

```
│   ├── Program.cs
│   └── WeatherApiClient.cs
└── AspireSample.sln
```

Add the .NET Aspire integration to the API

Add the .NET Aspire Azure Service Bus integration to your **AspireSample.ApiService** app:

1. In the **Solution Explorer**, double-click the **AspireSample.ApiService.csproj** file to open its XML file.
2. Add the following `<PackageReference>` item to the `<ItemGroup>` element:

   XML

   ```xml
   <ItemGroup>
       <PackageReference Include="Aspire.Azure.Messaging.ServiceBus"
                         Version="9.1.0" />
   </ItemGroup>
   ```

In the *Program.cs* file of the **AspireSample.ApiService** project, add a call to the `AddAzureServiceBusClient` extension method immediately after the existing call to `AddServiceDefaults`:

C#

```csharp
// Add service defaults & Aspire integrations.
builder.AddServiceDefaults();
builder.AddAzureServiceBusClient("serviceBusConnection");
```

For more information, see AddAzureServiceBusClient.

This method accomplishes the following tasks:

- Registers a ServiceBusClient with the DI container for connecting to Azure Service Bus.
- Automatically enables corresponding health checks, logging, and telemetry for the respective services.

In the *appsettings.json* file of the same project, add the corresponding connection information:

Passwordless (Recommended)

JSON

```
{
  // Existing configuration is omitted for brevity.
  "ConnectionStrings": {
    "serviceBusConnection": "{your_namespace}.servicebus.windows.net"
  }
}
```

> ⓘ **Note**
>
> Make sure to replace **{your_namespace}** in the service URIs with the name of your own Service Bus namespace.

Create the API endpoint

The API must provide an endpoint to receive data and publish it to the Service Bus topic and broadcast to subscribers. Add the following endpoint to the **AspireSample.ApiService** project to send a message to the Service Bus topic. Replace all of the contents of the *Program.cs* file with the following C# code:

C#

```
using Azure.Messaging.ServiceBus;

var builder = WebApplication.CreateBuilder(args);

// Add service defaults & Aspire integrations.
builder.AddServiceDefaults();
builder.AddAzureServiceBusClient("serviceBusConnection");

// Add services to the container.
builder.Services.AddProblemDetails();

var app = builder.Build();

// Configure the HTTP request pipeline.
app.UseExceptionHandler();

app.MapPost("/notify", static async (ServiceBusClient client, string
message) =>
{
    var sender = client.CreateSender("notifications");

    // Create a batch
    using ServiceBusMessageBatch messageBatch =
```

- Registers a ServiceBusClient with the DI container for connecting to Azure Service Bus.
- Automatically enables corresponding health checks, logging, and telemetry for the respective services.

In the *appsettings.json* file of the **AspireSample.WorkerService** project, add the corresponding connection information:

Passwordless (Recommended)

JSON

```
{
  // Existing configuration is omitted for brevity.
  "ConnectionStrings": {
    "serviceBusConnection": "{your_namespace}.servicebus.windows.net"
  }
}
```

> **Note**
>
> Make sure to replace **{your_namespace}** in the Service URIs with the name of your own Service Bus namespace.

Process the message from the subscriber

When a new message is placed on the `messages` queue, the worker service should retrieve, process, and delete the message. Update the *Worker.cs* class to match the following code:

C#

```
using Azure.Messaging.ServiceBus;

namespace AspireSample.WorkerService;

public sealed class Worker(
    ILogger<Worker> logger,
    ServiceBusClient client) : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
```

```
        {
            var processor = client.CreateProcessor(
                "notifications",
                "mobile",
                new ServiceBusProcessorOptions());

            // Add handler to process messages
            processor.ProcessMessageAsync += MessageHandler;

            // Add handler to process any errors
            processor.ProcessErrorAsync += ErrorHandler;

            // Start processing
            await processor.StartProcessingAsync();

            logger.LogInformation("""
                Wait for a minute and then press any key to end the
processing
                """);

            Console.ReadKey();

            // Stop processing
            logger.LogInformation("""

                Stopping the receiver...
                """);

            await processor.StopProcessingAsync();

            logger.LogInformation("Stopped receiving messages");
        }
    }

    async Task MessageHandler(ProcessMessageEventArgs args)
    {
        string body = args.Message.Body.ToString();

        logger.LogInformation("Received: {Body} from subscription.", body);

        // Complete the message. messages is deleted from the subscription.
        await args.CompleteMessageAsync(args.Message);
    }

    // Handle any errors when receiving messages
    Task ErrorHandler(ProcessErrorEventArgs args)
    {
        logger.LogError(args.Exception, "{Error}", args.Exception.Message);

        return Task.CompletedTask;
    }
}
```

Run and test the app locally

The sample app is now ready for testing. Verify that the data submitted to the API is sent to the Azure Service Bus topic and consumed by the subscriber worker service:

1. Launch the .NET Aspire project by selecting the **Start** debugging button, or by pressing `F5`. The .NET Aspire dashboard app should open in the browser.
2. On the resources page, in the **apiservice** row, find the link in the **Endpoints** that opens the `weatherforecast` endpoint. Note the HTTPS port number.
3. On the .NET Aspire dashboard, navigate to the logs for the **aspiresample-workerservice** project.
4. In a terminal window, use the `curl` command to send a test message to the API:

   Bash

   ```
   curl -X POST -H "Content-Type: application/json" https://localhost:{port}/notify?message=hello%20aspire
   ```

   Be sure to replace **{port}** with the port number you noted earlier.
5. Switch back to the **aspiresample-workerservice** logs. You should see the test message printed in the output logs.

Congratulations! You created and configured an ASP.NET Core API that connects to Azure Service Bus using Aspire integrations.

Clean up resources

Run the following Azure CLI command to delete the resource group when you no longer need the Azure resources you created. Deleting the resource group also deletes the resources contained inside of it.

Azure CLI

```
az group delete --name <your-resource-group-name>
```

For more information, see Clean up resources in Azure.

.NET Aspire Apache Kafka integration

Article • 10/15/2024

Includes: ✅ Hosting integration and ✅ Client integration

Apache Kafka is an open-source distributed event streaming platform. It's useful for building real-time data pipelines and streaming applications. The .NET Aspire Apache Kafka integration enables you to connect to existing Kafka instances, or create new instances from .NET with the docker.io/confluentinc/confluent-local container image.

Hosting integration

The Apache Kafka hosting integration models a Kafka server as the KafkaServerResource type. To access this type, install the 📦 Aspire.Hosting.Kafka NuGet package in the app host project, then add it with the builder.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Kafka
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Kafka server resource

In your app host project, call AddKafka on the `builder` instance to add a Kafka server resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var kafka = builder.AddKafka("kafka");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(kafka);

// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/confluentinc/confluent-local` image, it creates a new Kafka server instance on your local machine. A reference to your Kafka server (the `kafka` variable) is added to the `ExampleProject`. The Kafka server resource includes default ports

The WithReference method configures a connection in the `ExampleProject` named `"kafka"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing Kafka server, call **AddConnectionString** instead. For more information, see **Reference existing resources**.

Add Kafka UI

To add the Kafka UI to the Kafka server resource, call the WithKafkaUI method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var kafka = builder.AddKafka("kafka")
                   .WithKafkaUI();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(kafka);

// After adding all resources, run the app...
```

The Kafka UI is a free, open-source web UI to monitor and manage Apache Kafka clusters. .NET Aspire adds another container image docker.io/provectuslabs/kafka-ui to the app host that runs the Kafka UI.

Change the Kafka UI host port

To change the Kafka UI host port, chain a call to the WithHostPort method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var kafka = builder.AddKafka("kafka")
                   .WithKafkaUI(kafkaUI => kafkaUI.WithHostPort(9100));
```

```
builder.AddProject<Projects.ExampleProject>()
       .WithReference(kafka);

// After adding all resources, run the app...
```

The Kafka UI is accessible at `http://localhost:9100` in the preceding example.

Add Kafka server resource with data volume

To add a data volume to the Kafka server resource, call the WithDataVolume method on the Kafka server resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var kafka = builder.AddKafka("kafka")
                   .WithDataVolume(isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(kafka);

// After adding all resources, run the app...
```

The data volume is used to persist the Kafka server data outside the lifecycle of its container. The data volume is mounted at the `/var/lib/kafka/data` path in the Kafka server container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add Kafka server resource with data bind mount

To add a data bind mount to the Kafka server resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var kafka = builder.AddKafka("kafka")
                   .WithDataBindMount(
                       source: @"C:\Kafka\Data",
                       isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
```

```
        .WithReference(kafka);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Kafka server data across container restarts. The data bind mount is mounted at the `C:\Kafka\Data` on Windows (or `/Kafka/Data` on Unix) path on the host machine in the Kafka server container. For more information on data bind mounts, see Docker docs: Bind mounts.

Hosting integration health checks

The Kafka hosting integration automatically adds a health check for the Kafka server resource. The health check verifies that a Kafka producer with the specified connection name is able to connect and persist a topic to the Kafka server.

The hosting integration relies on the AspNetCore.HealthChecks.Kafka NuGet package.

Client integration

To get started with the .NET Aspire Apache Kafka integration, install the Aspire.Confluent.Kafka NuGet package in the client-consuming project, that is, the project for the application that uses the Apache Kafka client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Confluent.Kafka
```

Add Kafka producer

In the *Program.cs* file of your client-consuming project, call the AddKafkaProducer extension method to register an `IProducer<TKey, TValue>` for use via the dependency injection container. The method takes two generic parameters corresponding to the type of the key and the type of the message to send to the broker. These generic parameters are used by `AddKafkaProducer` to create an instance of `ProducerBuilder<TKey, TValue>`. This method also takes connection name parameter.

```
C#

builder.AddKafkaProducer<string, string>("messaging");
```

You can then retrieve the `IProducer<TKey, TValue>` instance using dependency injection. For example, to retrieve the producer from an `IHostedService`:

```
C#

internal sealed class Worker(IProducer<string, string> producer) :
BackgroundService
{
    // Use producer...
}
```

For more information on workers, see Worker services in .NET.

Add Kafka consumer

To register an `IConsumer<TKey, TValue>` for use via the dependency injection container, call the AddKafkaConsumer extension method in the *Program.cs* file of your client-consuming project. The method takes two generic parameters corresponding to the type of the key and the type of the message to receive from the broker. These generic parameters are used by `AddKafkaConsumer` to create an instance of `ConsumerBuilder<TKey, TValue>`. This method also takes connection name parameter.

```
C#

builder.AddKafkaConsumer<string, string>("messaging");
```

You can then retrieve the `IConsumer<TKey, TValue>` instance using dependency injection. For example, to retrieve the consumer from an `IHostedService`:

```
C#
```

```
internal sealed class Worker(IConsumer<string, string> consumer) :
BackgroundService
{
    // Use consumer...
}
```

Add keyed Kafka producers or consumers

There might be situations where you want to register multiple producer or consumer instances with different connection names. To register keyed Kafka producers or consumers, call the appropriate API:

- AddKeyedKafkaProducer: Registers a keyed Kafka producer.
- AddKeyedKafkaConsumer: Registers a keyed Kafka consumer.

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire Apache Kafka integration provides multiple options to configure the connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddKafkaProducer()` or `builder.AddKafkaProducer()`:

C#

```
builder.AddKafkaProducer<string, string>("kafka-producer");
```

Then the connection string is retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "kafka-producer": "broker:9092"
  }
}
```

The connection string value is set to the `BootstrapServers` property of the produced `IProducer<TKey, TValue>` or `IConsumer<TKey, TValue>` instance. For more information, see BootstrapServers.

Use configuration providers

The .NET Aspire Apache Kafka integration supports Microsoft.Extensions.Configuration. It loads the KafkaProducerSettings or KafkaConsumerSettings from configuration by respectively using the `Aspire:Confluent:Kafka:Producer` and `Aspire.Confluent:Kafka:Consumer` keys. The following snippet is an example of a *appsettings.json* file that configures some of the options:

JSON

```
{
  "Aspire": {
    "Confluent": {
      "Kafka": {
        "Producer": {
          "DisableHealthChecks": false,
          "Config": {
            "Acks": "All"
          }
        }
      }
    }
  }
}
```

The `Config` properties of both `Aspire:Confluent:Kafka:Producer` and `Aspire.Confluent:Kafka:Consumer` configuration sections respectively bind to instances of ProducerConfig and ConsumerConfig.

`Confluent.Kafka.Consumer<TKey, TValue>` requires the `ClientId` property to be set to let the broker track consumed message offsets.

For the complete Kafka client integration JSON schema, see Aspire.Confluent.Kafka/ConfigurationSchema.json.

Use inline delegates

There are several inline delegates available to configure various options.

Configure

You can pass the `Action<KafkaProducerSettings> configureSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

```
C#

builder.AddKafkaProducer<string, string>(
    "messaging",
    static settings => settings.DisableHealthChecks = true);
```

You can configure inline a consumer from code:

```
C#

builder.AddKafkaConsumer<string, string>(
    "messaging",
    static settings => settings.DisableHealthChecks = true);
```

Configure `ProducerBuilder<TKey, TValue>` and `ConsumerBuilder<TKey, TValue>`

To configure `Confluent.Kafka` builders, pass an `Action<ProducerBuilder<TKey, TValue>>` (or `Action<ConsumerBuilder<TKey, TValue>>`):

```
C#

builder.AddKafkaProducer<string, MyMessage>(
    "messaging",
    static producerBuilder =>
    {
        var messageSerializer = new MyMessageSerializer();
        producerBuilder.SetValueSerializer(messageSerializer);
    })
```

When registering producers and consumers, if you need to access a service registered in the DI container, you can pass an `Action<IServiceProvider, ProducerBuilder<TKey, TValue>>` or `Action<IServiceProvider, ConsumerBuilder<TKey, TValue>>` respectively:

- AddKafkaProducer<TKey,TValue>(IHostApplicationBuilder, String, Action<IServiceProvider,ProducerBuilder<TKey,TValue>>)
- AddKafkaConsumer<TKey,TValue>(IHostApplicationBuilder, String, Action<IServiceProvider,ConsumerBuilder<TKey,TValue>>)

Consider the following producer registration example:

```
C#
```

```
builder.AddKafkaProducer<string, MyMessage>(
    "messaging",
    static (serviceProvider, producerBuilder) =>
    {
        var messageSerializer =
serviceProvider.GetRequiredServices<MyMessageSerializer>();
        producerBuilder.SetValueSerializer(messageSerializer);
    })
```

For more information, see ProducerBuilder<TKey, TValue> and ConsumerBuilder<TKey, TValue> API documentation.

Client integration health checks

By default, .NET Aspire integrations enable health checks for all services. For more information, see .NET Aspire integrations overview.

The .NET Aspire Apache Kafka integration handles the following health check scenarios:

- Adds the `Aspire.Confluent.Kafka.Producer` health check when KafkaProducerSettings.DisableHealthChecks is `false`.
- Adds the `Aspire.Confluent.Kafka.Consumer` health check when KafkaConsumerSettings.DisableHealthChecks is `false`.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Apache Kafka integration uses the following log categories:

- `Aspire.Confluent.Kafka`

Tracing

The .NET Aspire Apache Kafka integration dos not emit distributed traces.

Metrics

The .NET Aspire Apache Kafka integration emits the following metrics using OpenTelemetry:

- `Aspire.Confluent.Kafka`
  - `messaging.kafka.network.tx`
  - `messaging.kafka.network.transmitted`
  - `messaging.kafka.network.rx`
  - `messaging.kafka.network.received`
  - `messaging.publish.messages`
  - `messaging.kafka.message.transmitted`
  - `messaging.receive.messages`
  - `messaging.kafka.message.received`

See also

- Apache Kafka
- Confluent
- Confluent Kafka .NET client docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire Azure integrations overview

Article • 03/10/2025

Azure is the most popular cloud platform for building and deploying .NET applications. The Azure SDK for .NET allows for easy management and use of Azure services. .NET Aspire provides a set of integrations with Azure services, where you're free to add new resources or connect to existing ones. This article details some common aspects of all Azure integrations in .NET Aspire and aims to help you understand how to use them.

All .NET Aspire Azure hosting integrations expose Azure resources and by convention are added using `AddAzure*` APIs. When you add these resources to your .NET Aspire app host, they represent an Azure service. The

Hosting integration	Description
Azure Cosmos DB	Call AzureCosmosExtensions.RunAsEmulator on the `IResourceBuilder<AzureCosmosDBResource>` to configure the Cosmos DB resource to be emulated with the NoSQL API.
Azure Event Hubs	Call AzureEventHubsExtensions.RunAsEmulator on the `IResourceBuilder<AzureEventHubsResource>` to configure the Event Hubs resource to be emulated.
Azure Service Bus	Call AzureServiceBusExtensions.RunAsEmulator on the `IResourceBuilder<AzureServiceBusResource>` to configure the Service Bus resource to be emulated with Service Bus emulator.
Azure SignalR Service	Call AzureSignalRExtensions.RunAsEmulator on the `IResourceBuilder<AzureSignalRResource>` to configure the SignalR resource to be emulated with Azure SignalR emulator.
Azure Storage	Call AzureStorageExtensions.RunAsEmulator on the `IResourceBuilder<AzureStorageResource>` to configure the Storage resource to be emulated with Azurite.

To have your Azure resources use the local emulators, chain a call the `RunAsEmulator` method on the Azure resource builder. This method configures the Azure resource to use the local emulator instead of the actual Azure service.

> **Important**
>
> Calling any of the available `RunAsEmulator` APIs on an Azure resource builder doesn't effect the **publishing manifest**. When you publish your app, **generated Bicep file** reflects the actual Azure service, not the local emulator.

Local containers

Some Azure resources can be substituted locally using open-source or on-premises containers. To substitute an Azure resource locally in a container, chain a call to the `RunAsContainer` method on the Azure resource builder. This method configures the Azure resource to use a containerized version of the service for local development and testing, rather than the actual Azure service.

Currently, .NET Aspire supports the following Azure services as containers:

Expand table

Hosting integration	Details
Azure Cache for Redis	Call AzureRedisExtensions.RunAsContainer on the `IResourceBuilder<AzureRedisCacheResource>` to configure it to run locally in a container,

Hosting integration	Details
	based on the `docker.io/library/redis` image.
Azure PostgreSQL Flexible Server	Call AzurePostgresExtensions.RunAsContainer on the `IResourceBuilder<AzurePostgresFlexibleServerResource>` to configure it to run locally in a container, based on the `docker.io/library/postgres` image.
Azure SQL Server	Call AzureSqlExtensions.RunAsContainer on the `IResourceBuilder<AzureSqlServerResource>` to configure it to run locally in a container, based on the `mcr.microsoft.com/mssql/server` image.

> **Note**
>
> Like emulators, calling `RunAsContainer` on an Azure resource builder doesn't effect the publishing manifest. When you publish your app, the generated Bicep file reflects the actual Azure service, not the local container.

Understand Azure integration APIs

.NET Aspire's strength lies in its ability to provide an amazing developer inner-loop. The Azure integrations are no different. They provide a set of common APIs and patterns that are shared across all Azure resources. These APIs and patterns are designed to make it easy to work with Azure resources in a consistent manner.

In the preceding containers section, you saw how to run Azure services locally in containers. If you're familiar with .NET Aspire, you might wonder how calling `AddAzureRedis("redis").RunAsContainer()` to get a local `docker.io/library/redis` container differs from `AddRedis("redis")`—as they both result in the same local container.

The answer is that there's no difference when running locally. However, when they're published you get different resources:

Expand table

API	Run mode	Publish mode
AddAzureRedis("redis").RunAsContainer()	Local Redis container	Azure Cache for Redis
AddRedis("redis")	Local Redis container	Azure Container App with Redis image

The same is true for SQL and PostgreSQL services:

Expand table

API	Run mode	Publish mode

Operation	API	Description
		is deployed instead of creating a new one.
Run	AzureSqlExtensions.RunAsContainer(IResourceBuilder<AzureSqlServerResource>, Action<IResourceBuilder<SqlServerServerResource>>) AzureRedisExtensions.RunAsContainer(IResourceBuilder<AzureRedisCacheResource>, Action<IResourceBuilder<RedisResource>>) RunAsContainer(IResourceBuilder<AzurePostgresFlexibleServerResource>, Action<IResourceBuilder<PostgresServerResource>>)	Configures an equivalent container to run locally. For more information, see Local containers.
Run	AzureCosmosExtensions.RunAsEmulator(IResourceBuilder<AzureCosmosDBResource>, Action<IResourceBuilder<AzureCosmosDBEmulatorResource>>) AzureSignalRExtensions.RunAsEmulator(IResourceBuilder<AzureSignalRResource>, Action<IResourceBuilder<AzureSignalREmulatorResource>>) AzureStorageExtensions.RunAsEmulator(IResourceBuilder<AzureStorageResource>, Action<IResourceBuilder<AzureStorageEmulatorResource>>) AzureEventHubsExtensions.RunAsEmulator(IResourceBuilder<AzureEventHubsResource>, Action<IResourceBuilder<AzureEventHubsEmulatorResource>>) AzureServiceBusExtensions.RunAsEmulator(IResourceBuilder<AzureServiceBusResource>, Action<IResourceBuilder<AzureServiceBusEmulatorResource>>)	Configures the Azure resource to be emulated. For more information, see Local emulators.
Run	RunAsExisting	Uses an existing resource when the application is running instead of creating a new one.
Publish and Run	AsExisting<T>(IResourceBuilder<T>, IResourceBuilder<ParameterResource>, IResourceBuilder<ParameterResource>)	Uses an existing resource regardless of the operation.

> **Note**
>
> Not all APIs are available on all Azure resources. For example, some Azure resources can be containerized or emulated, while others can't.

For more information on execution modes, see Execution context.

General run mode API use cases

Use RunAsExisting when you need to dynamically interact with an existing resource during runtime without needing to deploy or update it. Use PublishAsExisting when declaring existing resources as part of a deployment configuration, ensuring the correct scopes and permissions are applied. Finally, use AsExisting<T>(IResourceBuilder<T>, IResourceBuilder<ParameterResource>, IResourceBuilder<ParameterResource>) when declaring existing resources in both configurations, with a requirement to parameterize the references.

You can query whether a resource is marked as an existing resource, by calling the IsExisting(IResource) extension method on the IResource. For more information, see Use existing Azure resources.

Use existing Azure resources

.NET Aspire provides support for referencing existing Azure resources. You mark an existing resource through the `PublishAsExisting`, `RunAsExisting`, and `AsExisting` APIs. These APIs allow developers to reference already-deployed Azure resources, configure them, and generate appropriate deployment manifests using Bicep templates.

Existing resources referenced with these APIs can be enhanced with role assignments and other customizations that are available with .NET Aspire's infrastructure as code capabilities. These APIs are limited to Azure resources that can be deployed with Bicep templates.

Configure existing Azure resources for run mode

The RunAsExisting method is used when a distributed application is executing in "run" mode. In this mode, it assumes that the referenced Azure resource already exists and integrates with it during execution without provisioning the resource. To mark an Azure resource as existing, call the `RunAsExisting` method on the resource builder. Consider the following example:

C#

```
var builder = DistributedApplication.CreateBuilder();

var existingServiceBusName = builder.AddParameter("existingServiceBusName");
var existingServiceBusResourceGroup =
builder.AddParameter("existingServiceBusResourceGroup");

var serviceBus = builder.AddAzureServiceBus("messaging")
                        .RunAsExisting(existingServiceBusName,
existingServiceBusResourceGroup);

serviceBus.AddServiceBusQueue("queue");
```

The preceding code:

- Creates a new `builder` instance.

- Adds a parameter named `existingServiceBusName` to the builder.
- Adds an Azure Service Bus resource named `messaging` to the builder.
- Calls the `RunAsExisting` method on the `serviceBus` resource builder, passing the `existingServiceBusName` parameter—alternatively, you can use the `string` parameter overload.
- Adds a queue named `queue` to the `serviceBus` resource.

By default, the Service Bus parameter reference is assumed to be in the same Azure resource group. However, if it's in a different resource group, you can pass the resource group explicitly as a parameter to correctly specify the appropriate resource grouping.

The PublishAsExisting method is used in "publish" mode when the intent is to declare and reference an already-existing Azure resource during publish mode. This API facilitates the creation of manifests and templates that include resource definitions that map to existing resources in Bicep.

To mark an Azure resource as existing in for the "publish" mode, call the `PublishAsExisting` method on the resource builder. Consider the following example:

```
C#
```

The preceding code:

JSON

```
"messaging": {
  "type": "azure.bicep.v0",
  "connectionString": "{messaging.outputs.serviceBusEndpoint}",
  "path": "messaging.module.bicep",
  "params": {
    "existingServiceBusName": "{existingServiceBusName.value}",
    "principalType": "",
    "principalId": ""
  }
},
"queue": {
  "type": "value.v0",
  "connectionString": "{messaging.outputs.serviceBusEndpoint}"
}
```

For more information on the manifest file, see .NET Aspire manifest format for deployment tool builders.

Additionally, the generated Bicep template includes the `existingResourceName` parameter, which can be used to reference the existing Azure resource. Consider the following generated Bicep template:

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location

param existingServiceBusName string

param principalType string

param principalId string

resource messaging 'Microsoft.ServiceBus/namespaces@2024-01-01' existing = {
  name: existingServiceBusName
}

resource messaging_AzureServiceBusDataOwner
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(messaging.id, principalId,
subscriptionResourceId('Microsoft.Authorization/roleDefinitions',
```

```
}

output serviceBusEndpoint string = messaging.properties.serviceBusEndpoint
```

For more information on the generated Bicep templates, see Infrastructure as code and consider other publishing APIs.

The AsExisting<T>(IResourceBuilder<T>, IResourceBuilder<ParameterResource>, IResourceBuilder<ParameterResource>) method is used when the distributed application is running in "run" or "publish" mode. Because the `AsExisting` method operates in both scenarios, it only supports a parameterized reference to the resource name or resource group name. This approach helps prevent the use of the same resource in both testing and production environments.

To mark an Azure resource as existing, call the `AsExisting` method on the resource builder. Consider the following example:

C#

The preceding code:

- Creates a new `builder` instance.
- Adds a parameter named `existingServiceBusName` to the builder.
- Adds an Azure Service Bus resource named `messaging` to the builder.

- Calls the `AsExisting` method on the `serviceBus` resource builder, passing the `existingServiceBusName` parameter.
- Adds a queue named `queue` to the `serviceBus` resource.

Add existing Azure resources with connection strings

.NET Aspire provides the ability to connect to existing resources, including Azure resources. Expressing connection strings is useful when you have existing Azure resources that you want to use in your .NET Aspire app. The AddConnectionString API is used with the app host's execution context to conditionally add a connection string to the app model.

> ⓘ **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

Consider the following example, where in *publish* mode you add an Azure Storage resource while in *run* mode you add a connection string to an existing Azure Storage:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.ExecutionContext.IsPublishMode
    ? builder.AddAzureStorage("storage")
    : builder.AddConnectionString("storage");

builder.AddProject<Projects.Api>("api")
       .WithReference(storage);

// After adding all resources, run the app...
```

The preceding code:

- Creates a new `builder` instance.
- Adds an Azure Storage resource named `storage` in "publish" mode.
- Adds a connection string to an existing Azure Storage named `storage` in "run" mode.
- Adds a project named `api` to the builder.
- The `api` project references the `storage` resource regardless of the mode.

The consuming API project uses the connection string information with no knowledge of how the app host configured it. In "publish" mode, the code adds a new Azure Storage resource—which would be reflected in the deployment manifest accordingly. When in "run" mode the connection

string corresponds to a configuration value visible to the app host. It's assumed that all role assignments for the target resource are configured. This means, you'd likely configure an environment variable or a user secret to store the connection string. The configuration is resolved from the `ConnectionStrings__storage` (or `ConnectionStrings:storage`) configuration key. These configuration values can be viewed when the app runs. For more information, see Resource details.

Unlike existing resources modeled with the first-class AsExisting API, existing resource modeled as connection strings can't be enhanced with additional role assignments or infrastructure customizations.

Publish as Azure Container App

.NET Aspire allows you to publish primitive resources as Azure Container Apps, a serverless platform that reduces infrastructure management. Supported resource types include:

- ContainerResource: Represents a specified container.
- ExecutableResource: Represents a specified executable process.
- ProjectResource: Represents a specified .NET project.

To publish these resources, use the following APIs:

- AzureContainerAppContainerExtensions.PublishAsAzureContainerApp<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure,ContainerApp>)
- AzureContainerAppExecutableExtensions.PublishAsAzureContainerApp<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure,ContainerApp>)
- AzureContainerAppProjectExtensions.PublishAsAzureContainerApp<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure,ContainerApp>)

These APIs configure the resource to be published as an Azure Container App and implicitly call AddAzureContainerAppsInfrastructure(IDistributedApplicationBuilder) to add the necessary infrastructure and Bicep files to your app host. As an example, consider the following code:

C#

```
var builder = DistributedApplication.CreateBuilder();

var env = builder.AddParameter("env");

var api = builder.AddProject<Projects.AspireApi>("api")
                 .PublishAsAzureContainerApp<Projects.AspireApi>((infra, app) =>
                 {
                     app.Template.Containers[0].Value!.Env.Add(new
ContainerAppEnvironmentVariable
                     {
                         Name = "Hello",
                         Value = env.AsProvisioningParameter(infra)
                     });
                 });
```

The preceding code:

- Creates a new `builder` instance.
- Adds a parameter named `env` to the builder.
- Adds a project named `api` to the builder.
- Calls the `PublishAsAzureContainerApp` method on the `api` resource builder, passing a lambda expression that configures the Azure Container App infrastructure—where `infra` is the AzureResourceInfrastructure and `app` is the ContainerApp.
  - Adds an environment variable named `Hello` to the container app, using the `env` parameter.
  - The `AsProvisioningParameter` method is used to treat `env` as either a new ProvisioningParameter in infrastructure, or reuses an existing bicep parameter if one with the same name already exists.

For more information, see ContainerApp and AsProvisioningParameter.

The Azure SDK for .NET provides the  Azure.Provisioning NuGet package and a suite of service-specific Azure provisioning packages . These Azure provisioning libraries make it easy to declaratively specify Azure infrastructure natively in .NET. Their APIs enable you to write object-oriented infrastructure in C#, resulting in Bicep. Bicep is a domain-specific language (DSL) for deploying Azure resources declaratively.

While it's possible to provision Azure resources manually, .NET Aspire simplifies the process by providing a set of APIs to express Azure resources. These APIs are available as extension methods in .NET Aspire Azure hosting libraries, extending the IDistributedApplicationBuilder interface. When you add Azure resources to your app host, they add the appropriate provisioning functionality implicitly. In other words, you don't need to call any provisioning APIs directly.

Since .NET Aspire models Azure resources within Azure hosting integrations, the Azure SDK is used to provision these resources. Bicep files are generated that define the Azure resources you need. The generated Bicep files are output alongside the manifest file when you publish your app.

There are several ways to influence the generated Bicep files:

- Azure.Provisioning customization:
  - Configure infrastructure: Customize Azure resource infrastructure.
  - Add Azure infrastructure: Manually add Azure infrastructure to your app host.
- Use custom Bicep templates:
  - Reference Bicep files: Add a reference to a Bicep file on disk.

To avoid conflating terms and to help disambiguate "provisioning," it's important to understand the distinction between *local provisioning* and *Azure provisioning*:

- ***Local provisioning:***

  By default, when you call any of the Azure hosting integration APIs to add Azure resources, the AddAzureProvisioning(IDistributedApplicationBuilder) API is called implicitly. This API registers services in the dependency injection (DI) container that are used to provision Azure resources when the app host starts. This concept is known as *local provisioning*. For more information, see Local Azure provisioning.

- ***`Azure.Provisioning`:***

  `Azure.Provisioning` refers to the NuGet package, and is a set of libraries that lets you use C# to generate Bicep. The Azure hosting integrations in .NET Aspire use these libraries under the covers to generate Bicep files that define the Azure resources you need. For more information, see Azure.Provisioning customization.

`Azure.Provisioning` customization

All .NET Aspire Azure hosting integrations expose various Azure resources, and they're all subclasses of the AzureProvisioningResource type—which itself inherits the AzureBicepResource. This enables extensions that are generically type-constrained to this type, allowing for a fluent API to customize the infrastructure to your liking. While .NET Aspire provides defaults, you're free to influence the generated Bicep using these APIs.

Configure infrastructure

Regardless of the Azure resource you're working with, to configure its underlying infrastructure, you chain a call to the ConfigureInfrastructure extension method. This method allows you to customize the infrastructure of the Azure resource by passing a `configure` delegate of type `Action<AzureResourceInfrastructure>`. The AzureResourceInfrastructure type is a subclass of the Azure.Provisioning.Infrastructure. This type exposes a massive API surface area for configuring the underlying infrastructure of the Azure resource.

Consider the following example:

C#

```
var sku = builder.AddParameter("storage-sku");

var storage = builder.AddAzureStorage("storage")
    .ConfigureInfrastructure(infra =>
    {
        var resources = infra.GetProvisionableResources();

        var storageAccount = resources.OfType<StorageAccount>().Single();

        storageAccount.Sku = new StorageSku
```

```
        {
            Name = sku.AsProvisioningParameter(infra)
        };
    });
```

The preceding code:

- Adds a parameter named `storage-sku`.
- Adds Azure Storage with the AddAzureStorage API named `storage`.
- Chains a call to `ConfigureInfrastructure` to customize the Azure Storage infrastructure:
  - Gets the provisionable resources.
  - Filters to a single StorageAccount.
  - Assigns the `storage-sku` parameter to the StorageAccount.Sku property:
    - A new instance of the StorageSku has its `Name` property assigned from the result of the AsProvisioningParameter API.

This exemplifies flowing an external parameter into the Azure Storage infrastructure, resulting in the generated Bicep file reflecting the desired configuration.

Add Azure infrastructure

Not all Azure services are exposed as .NET Aspire integrations. While they might be at a later time, you can still provision services that are available in `Azure.Provisioning.*` libraries. Imagine a scenario where you have worker service that's responsible for managing an Azure Container Registry. Now imagine that an app host project takes a dependency on the  Azure.Provisioning.ContainerRegistry NuGet package.

You can use the `AddAzureInfrastructure` API to add the Azure Container Registry infrastructure to your app host:

C#

```
var acr = builder.AddAzureInfrastructure("acr", infra =>
{
    var registry = new ContainerRegistryService("acr")
    {
        Sku = new()
        {
            Name = ContainerRegistrySkuName.Standard
        },
    };
    infra.Add(registry);

    var output = new ProvisioningOutput("registryName", typeof(string))
    {
        Value = registry.Name
    };
    infra.Add(output);
});

builder.AddProject<Projects.WorkerService>("worker")
```

```
        .WithEnvironment(
            "ACR_REGISTRY_NAME",
            new BicepOutputReference("registryName", acr.Resource));
```

The preceding code:

- Calls AddAzureInfrastructure with a name of `acr`.
- Provides a `configureInfrastructure` delegate to customize the Azure Container Registry infrastructure:
  - Instantiates a ContainerRegistryService with the name `acr` and a standard SKU.
  - Adds the Azure Container Registry service to the `infra` variable.
  - Instantiates a ProvisioningOutput with the name `registryName`, a type of `string`, and a value that corresponds to the name of the Azure Container Registry.
  - Adds the output to the `infra` variable.
- Adds a project named `worker` to the builder.
- Chains a call to WithEnvironment to set the `ACR_REGISTRY_NAME` environment variable in the project to the value of the `registryName` output.

The functionality demonstrates how to add Azure infrastructure to your app host project, even if the Azure service isn't directly exposed as a .NET Aspire integration. It further shows how to flow the output of the Azure Container Registry into the environment of a dependent project.

Consider the resulting Bicep file:

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location

resource acr 'Microsoft.ContainerRegistry/registries@2023-07-01' = {
  name: take('acr${uniqueString(resourceGroup().id)}', 50)
  location: location
  sku: {
    name: 'Standard'
  }
}

output registryName string = acr.name
```

The Bicep file reflects the desired configuration of the Azure Container Registry, as defined by the `AddAzureInfrastructure` API.

Use custom Bicep templates

When you're targeting Azure as your desired cloud provider, you can use Bicep to define your infrastructure as code. It aims to drastically simplify the authoring experience with a cleaner syntax and better support for modularity and code reuse.

While .NET Aspire provides a set of prebuilt Bicep templates, there might be times when you either want to customize the templates or create your own. This section explains the concepts and corresponding APIs that you can use to customize the Bicep templates.

> **Important**
>
> This section isn't intended to teach you Bicep, but rather to provide guidance on how to create custom Bicep templates for use with .NET Aspire.

As part of the Azure deployment story for .NET Aspire, the Azure Developer CLI (`azd`) provides an understanding of your .NET Aspire project and the ability to deploy it to Azure. The `azd` CLI uses the Bicep templates to deploy the application to Azure.

Install `Aspire.Hosting.Azure` package

When you want to reference Bicep files, it's possible that you're not using any of the Azure hosting integrations. In this case, you can still reference Bicep files by installing the `Aspire.Hosting.Azure` package. This package provides the necessary APIs to reference Bicep files and customize the Azure resources.

> **Tip**
>
> If you're using any of the Azure hosting integrations, you don't need to install the `Aspire.Hosting.Azure` package, as it's a transitive dependency.

To use any of this functionality, the Aspire.Hosting.Azure NuGet package must be installed:

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

What to expect from the examples

All the examples in this section assume that you're using the Aspire.Hosting.Azure namespace. Additionally, the examples assume you have an IDistributedApplicationBuilder instance:

C#

```
using Aspire.Hosting.Azure;

var builder = DistributedApplication.CreateBuilder(args);

// Examples go here...

builder.Build().Run();
```

By default, when you call any of the Bicep-related APIs, a call is also made to AddAzureProvisioning that adds support for generating Azure resources dynamically during application startup. For more information, see Local provisioning and Azure.Provisioning.

Reference Bicep files

Imagine that you have a Bicep template in a file named `storage.bicep` that provisions an Azure Storage Account:

Bicep

```
param location string = resourceGroup().location
param storageAccountName string = 'toylaunch${uniqueString(resourceGroup().id)}'

resource storageAccount 'Microsoft.Storage/storageAccounts@2021-06-01' = {
  name: storageAccountName
  location: location
  sku: {
    name: 'Standard_LRS'
  }
  kind: 'StorageV2'
  properties: {
    accessTier: 'Hot'
  }
}
```

To add a reference to the Bicep file on disk, call the AddBicepTemplate method. Consider the following example:

C#

```
builder.AddBicepTemplate(
    name: "storage",
    bicepFile: "../infra/storage.bicep");
```

The preceding code adds a reference to a Bicep file located at `../infra/storage.bicep`. The file paths should be relative to the *app host* project. This reference results in an AzureBicepResource being added to the application's resources collection with the `"storage"` name, and the API returns an `IResourceBuilder<AzureBicepResource>` instance that can be used to further customize the resource.

Reference Bicep inline

While having a Bicep file on disk is the most common scenario, you can also add Bicep templates inline. Inline templates can be useful when you want to define a template in code or when you want to generate the template dynamically. To add an inline Bicep template, call the AddBicepTemplateString method with the Bicep template as a `string`. Consider the following example:

C#

```
builder.AddBicepTemplateString(
        name: "ai",
        bicepContent: """
        @description('That name is the name of our application.')
        param cognitiveServiceName string =
'CognitiveService-${uniqueString(resourceGroup().id)}'

        @description('Location for all resources.')
        param location string = resourceGroup().location

        @allowed([
          'S0'
        ])
        param sku string = 'S0'

        resource cognitiveService 'Microsoft.CognitiveServices/accounts@2021-10-01' =
{
          name: cognitiveServiceName
          location: location
          sku: {
            name: sku
          }
          kind: 'CognitiveServices'
          properties: {
            apiProperties: {
              statisticsEnabled: false
            }
          }
        }
        """
    );
```

In this example, the Bicep template is defined as an inline `string` and added to the application's resources collection with the name `"ai"`. This example provisions an Azure AI resource.

Pass parameters to Bicep templates

Bicep supports accepting parameters, which can be used to customize the behavior of the template. To pass parameters to a Bicep template from .NET Aspire, chain calls to the WithParameter method as shown in the following example:

C#

The preceding code:

- Adds a parameter named `"region"` to the `builder` instance.
- Adds a reference to a Bicep file located at `../infra/storage.bicep`.
- Passes the `"region"` parameter to the Bicep template, which is resolved using the standard parameter resolution.
- Passes the `"storageName"` parameter to the Bicep template with a *hardcoded* value.
- Passes the `"tags"` parameter to the Bicep template with an array of strings.

For more information, see External parameters.

.NET Aspire provides a set of well-known parameters that can be passed to Bicep templates. These parameters are used to provide information about the application and the environment to the Bicep templates. The following well-known parameters are available:

Field	Description	Value
AzureBicepResource.KnownParameters.KeyVaultName	The name of the key vault resource used to store secret outputs.	`"keyVaultName"`
AzureBicepResource.KnownParameters.Location	The location of the resource. This is required for all resources.	`"location"`
AzureBicepResource.KnownParameters.LogAnalyticsWorkspaceId	The resource ID of the log analytics workspace.	`"logAnalyticsWorkspaceId"`
AzureBicepResource.KnownParameters.PrincipalId	The principal ID of the current user or managed identity.	`"principalId"`
AzureBicepResource.KnownParameters.PrincipalName	The principal name of the	

Field	Description	Value
	current user or managed identity.	
AzureBicepResource.KnownParameters.PrincipalType	The principal type of the current user or managed identity. Either `User` or `ServicePrincipal`.	`"principalType"`

To use a well-known parameter, pass the parameter name to the WithParameter method, such as `WithParameter(AzureBicepResource.KnownParameters.KeyVaultName)`. You don't pass values for well-known parameters, as .NET Aspire resolves them on your behalf.

Consider an example where you want to set up an Azure Event Grid webhook. You might define the Bicep template as follows:

Bicep

```
param topicName string
param webHookEndpoint string
param principalId string
param principalType string
param location string = resourceGroup().location

// The topic name must be unique because it's represented by a DNS entry.
// must be between 3-50 characters and contain only values a-z, A-Z, 0-9, and "-".

resource topic 'Microsoft.EventGrid/topics@2023-12-15-preview' = {
  name: toLower(take('${topicName}${uniqueString(resourceGroup().id)}', 50))
  location: location

  resource eventSubscription 'eventSubscriptions' = {
    name: 'customSub'
    properties: {
      destination: {
        endpointType: 'WebHook'
        properties: {
          endpointUrl: webHookEndpoint
        }
      }
    }
  }
}
```

```
resource EventGridRoleAssignment 'Microsoft.Authorization/roleAssignments@2022-04-01'
= {
  name: guid(topic.id, principalId,
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', 'd5a91429-5739-
47e2-a06b-3470a27159e7'))
  scope: topic
  properties: {
    principalId: principalId
    principalType: principalType
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', 'd5a91429-5739-
47e2-a06b-3470a27159e7')
  }
}

output endpoint string = topic.properties.endpoint
```

This Bicep template defines several parameters, including the `topicName`, `webHookEndpoint`, `principalId`, `principalType`, and the optional `location`. To pass these parameters to the Bicep template, you can use the following code snippet:

C#

```
var webHookApi = builder.AddProject<Projects.WebHook_Api>("webhook-api");

var webHookEndpointExpression = ReferenceExpression.Create(
        $"{webHookApi.GetEndpoint("https")}/hook");

builder.AddBicepTemplate("event-grid-webhook", "../infra/event-grid-webhook.bicep")
       .WithParameter("topicName", "events")
       .WithParameter(AzureBicepResource.KnownParameters.PrincipalId)
       .WithParameter(AzureBicepResource.KnownParameters.PrincipalType)
       .WithParameter("webHookEndpoint", () => webHookEndpointExpression);
```

- The `webHookApi` project is added as a reference to the `builder`.
- The `topicName` parameter is passed a hardcoded name value.
- The `webHookEndpoint` parameter is passed as an expression that resolves to the URL from the `api` project references' "https" endpoint with the `/hook` route.
- The `principalId` and `principalType` parameters are passed as well-known parameters.

The well-known parameters are convention-based and shouldn't be accompanied with a corresponding value when passed using the `WithParameter` API. Well-known parameters simplify some common functionality, such as *role assignments*, when added to the Bicep templates, as shown in the preceding example. Role assignments are required for the Event Grid webhook to send events to the specified endpoint. For more information, see Event Grid Data Sender role assignment.

Get outputs from Bicep references

In addition to passing parameters to Bicep templates, you can also get outputs from the Bicep templates. Consider the following Bicep template, as it defines an `output` named `endpoint`:

Bicep

```
param storageName string
param location string = resourceGroup().location

resource myStorageAccount 'Microsoft.Storage/storageAccounts@2019-06-01' = {
  name: storageName
  location: location
  kind: 'StorageV2'
  sku:{
    name:'Standard_LRS'
    tier: 'Standard'
  }
  properties: {
    accessTier: 'Hot'
  }
}

output endpoint string = myStorageAccount.properties.primaryEndpoints.blob
```

The Bicep defines an output named `endpoint`. To get the output from the Bicep template, call the GetOutput method on an `IResourceBuilder<AzureBicepResource>` instance as demonstrated in following C# code snippet:

C#

```
var storage = builder.AddBicepTemplate(
        name: "storage",
        bicepFile: "../infra/storage.bicep"
    );

var endpoint = storage.GetOutput("endpoint");
```

In this example, the output from the Bicep template is retrieved and stored in an `endpoint` variable. Typically, you would pass this output as an environment variable to another resource that relies on it. For instance, if you had an ASP.NET Core Minimal API project that depended on this endpoint, you could pass the output as an environment variable to the project using the following code snippet:

C#

```
var storage = builder.AddBicepTemplate(
                name: "storage",
                bicepFile: "../infra/storage.bicep"
            );

var endpoint = storage.GetOutput("endpoint");

var apiService = builder.AddProject<Projects.AspireSample_ApiService>(
        name: "apiservice"
    )
    .WithEnvironment("STORAGE_ENDPOINT", endpoint);
```

For more information, see Bicep outputs.

Get secret outputs from Bicep references

It's important to avoid outputs for secrets when working with Bicep. If an output is considered a *secret*, meaning it shouldn't be exposed in logs or other places, you can treat it as such. This can be achieved by storing the secret in Azure Key Vault and referencing it in the Bicep template. .NET Aspire's Azure integration provides a pattern for securely storing outputs from the Bicep template by allows resources to use the `keyVaultName` parameter to store secrets in Azure Key Vault.

Consider the following Bicep template as an example the helps to demonstrate this concept of securing secret outputs:

Bicep

```
param databaseAccountName string
param keyVaultName string

param databases array = []

@description('Tags that will be applied to all resources')
param tags object = {}

param location string = resourceGroup().location

var resourceToken = uniqueString(resourceGroup().id)

resource cosmosDb 'Microsoft.DocumentDB/databaseAccounts@2023-04-15' = {
    name: replace('${databaseAccountName}-${resourceToken}', '-', '')
    location: location
    kind: 'GlobalDocumentDB'
    tags: tags
    properties: {
        consistencyPolicy: { defaultConsistencyLevel: 'Session' }
        locations: [
            {
                locationName: location
                failoverPriority: 0
            }
        ]
        databaseAccountOfferType: 'Standard'
    }

    resource db 'sqlDatabases@2023-04-15' = [for name in databases: {
        name: '${name}'
        location: location
        tags: tags
        properties: {
            resource: {
                id: '${name}'
            }
        }
    }]
}

var primaryMasterKey = cosmosDb.listKeys(cosmosDb.apiVersion).primaryMasterKey
```

```
resource vault 'Microsoft.KeyVault/vaults@2023-07-01' existing = {
    name: keyVaultName

    resource secret 'secrets@2023-07-01' = {
        name: 'connectionString'
        properties: {
            value:
'AccountEndpoint=${cosmosDb.properties.documentEndpoint};AccountKey=${primaryMasterKe
y}'
        }
    }
}
```

The preceding Bicep template expects a `keyVaultName` parameter, among several other parameters. It then defines an Azure Cosmos DB resource and stashes a secret into Azure Key Vault, named `connectionString` which represents the fully qualified connection string to the Cosmos DB instance. To access this secret connection string value, you can use the following code snippet:

C#

```
var cosmos = builder.AddBicepTemplate("cosmos", "../infra/cosmosdb.bicep")
    .WithParameter("databaseAccountName", "fallout-db")
    .WithParameter(AzureBicepResource.KnownParameters.KeyVaultName)
    .WithParameter("databases", ["vault-33", "vault-111"]);

var connectionString =
    cosmos.GetSecretOutput("connectionString");

builder.AddProject<Projects.WebHook_Api>("api")
    .WithEnvironment(
        "ConnectionStrings__cosmos",
        connectionString);
```

In the preceding code snippet, the `cosmos` Bicep template is added as a reference to the `builder`. The `connectionString` secret output is retrieved from the Bicep template and stored in a variable. The secret output is then passed as an environment variable (`ConnectionStrings__cosmos`) to the `api` project. This environment variable is used to connect to the Cosmos DB instance.

When this resource is deployed, the underlying deployment mechanism will automatically Reference secrets from Azure Key Vault. To guarantee secret isolation, .NET Aspire creates a Key Vault per source.

> **Note**
>
> In *local provisioning* mode, the secret is extracted from Key Vault and set it in an environment variable. For more information, see Local Azure provisioning.

Publishing

When you publish your app, the Azure provisioning generated Bicep is used by the Azure Developer CLI to create the Azure resources in your Azure subscription. .NET Aspire outputs a publishing manifest, that's also a vital part of the publishing process. The Azure Developer CLI is a command-line tool that provides a set of commands to manage Azure resources.

For more information on publishing and deployment, see Deploy a .NET Aspire project to Azure Container Apps using the Azure Developer CLI (in-depth guide).

Local Azure provisioning

Article • 12/16/2024

.NET Aspire simplifies local cloud-native app development with its compelling app host model. This model allows you to run your app locally with the same configuration and services as in Azure. In this article you learn how to provision Azure resources from your local development environment through the .NET Aspire app host.

> **Note**
>
> To be clear, resources are provisioned in Azure, but the provisioning process is initiated from your local development environment. To optimize your local development experience, consider using emulator or containers when available. For more information, see **Typical developer experience**.

Requirements

This article assumes that you have an Azure account and subscription. If you don't have an Azure account, you can create a free one at Azure Free Account. For provisioning functionality to work correctly, you'll need to be authenticated with Azure. Ensure that you have the Azure Developer CLI installed. Additionally, you'll need to provide some configuration values so that the provisioning logic can create resources on your behalf.

App host provisioning APIs

The app host provides a set of APIs to express Azure resources. These APIs are available as extension methods in .NET Aspire Azure hosting libraries, extending the IDistributedApplicationBuilder interface. When you add Azure resources to your app host, they'll add the appropriate provisioning functionality implicitly. In other words, you don't need to call any provisioning APIs directly.

When the app host starts, the following provisioning logic is executed:

1. The `Azure` configuration section is validated.
2. When invalid the dashboard and app host output provides hints as to what's missing. For more information, see Missing configuration value hints.
3. When valid Azure resources are conditionally provisioned:
    a. If an Azure deployment for a given resource doesn't exist, it's created and configured as a deployment.

b. The configuration of said deployment is stamped with a checksum as a means to support only provisioning resources as necessary.

Use existing Azure resources

The app host automatically manages provisioning of Azure resources. The first time the app host runs, it provisions the resources specified in the app host. Subsequent runs don't provision the resources again unless the app host configuration changes.

If you've already provisioned Azure resources outside of the app host and want to use them, you can provide the connection string with the AddConnectionString API as shown in the following Azure Key Vault example:

C#

```
// Service registration
var secrets = builder.ExecutionContext.IsPublishMode
    ? builder.AddAzureKeyVault("secrets")
    : builder.AddConnectionString("secrets");

// Service consumption
builder.AddProject<Projects.ExampleProject>()
       .WithReference(secrets)
```

The preceding code snippet shows how to add an Azure Key Vault to the app host. The AddAzureKeyVault API is used to add the Azure Key Vault to the app host. The `AddConnectionString` API is used to provide the connection string to the app host.

Alternatively, for some Azure resources, you can opt-in to running them as an emulator with the `RunAsEmulator` API. This API is available for Azure Cosmos DB and Azure Storage integrations. For example, to run Azure Cosmos DB as an emulator, you can use the following code snippet:

C#

```
var cosmos = builder.AddAzureCosmosDB("cosmos")
                    .RunAsEmulator();
```

The RunAsEmulator API configures an Azure Cosmos DB resource to be emulated using the Azure Cosmos DB emulator with the NoSQL API.

.NET Aspire Azure hosting integrations

If you're using Azure resources in your app host, you're using one or more of the .NET Aspire Azure hosting integrations. These hosting libraries provide extension methods to the IDistributedApplicationBuilder interface to add Azure resources to your app host.

Configuration

When utilizing Azure resources in your local development environment, you need to provide the necessary configuration values. Configuration values are specified under the `Azure` section:

- `SubscriptionId`: The Azure subscription ID.
- `AllowResourceGroupCreation`: A boolean value that indicates whether to create a new resource group.
- `ResourceGroup`: The name of the resource group to use.
- `Location`: The Azure region to use.

Consider the following example *appsettings.json* configuration:

JSON

```
{
  "Azure": {
    "SubscriptionId": "<Your subscription id>",
    "AllowResourceGroupCreation": true,
    "ResourceGroup": "<Valid resource group name>",
    "Location": "<Valid Azure location>"
  }
}
```

> **ⓘ Important**
>
> It's recommended to store these values as app secrets. For more information, see Manage app secrets.

After you've configured the necessary values, you can start provisioning Azure resources in your local development environment.

Azure provisioning credential store

The .NET Aspire app host uses a credential store for Azure resource authentication and authorization. Depending on your subscription, the correct credential store may be needed for multi-tenant provisioning scenarios.

With the Aspire.Hosting.Azure NuGet package installed, and if your app host depends on Azure resources, the default Azure credential store relies on the DefaultAzureCredential. To change this behavior, you can set the credential store value in the *appsettings.json* file, as shown in the following example:

JSON

```
{
  "Azure": {
    "CredentialSource": "AzureCli"
  }
}
```

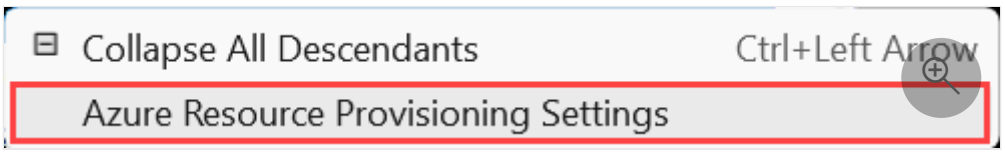
As with all configuration-based settings, you can configure these with alternative providers, such as user secrets or environment variables. The `Azure:CredentialSource` value can be set to one of the following values:

- `AzureCli`: Delegates to the AzureCliCredential.
- `AzurePowerShell`: Delegates to the AzurePowerShellCredential.
- `VisualStudio`: Delegates to the VisualStudioCredential.
- `VisualStudioCode`: Delegates to the VisualStudioCodeCredential.
- `AzureDeveloperCli`: Delegates to the AzureDeveloperCliCredential.
- `InteractiveBrowser`: Delegates to the InteractiveBrowserCredential.

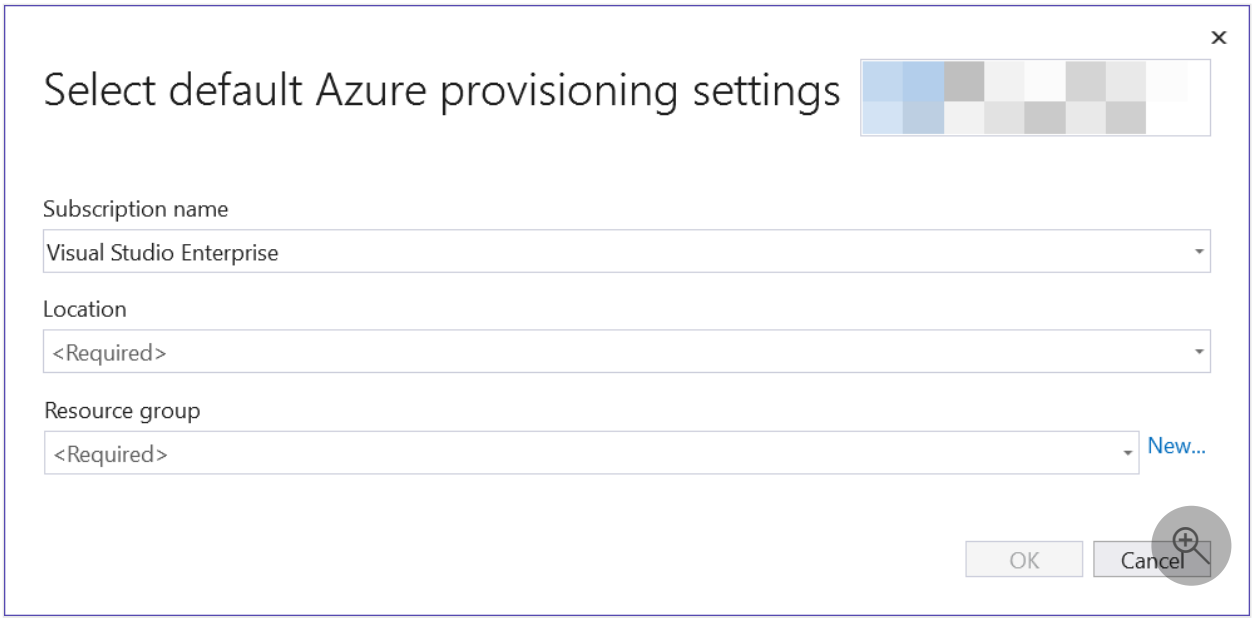
> **Tip**
>
> For more information about the Azure SDK authentication and authorization, see Credential chains in the Azure Identity library for .NET.

Tooling support

In Visual Studio, you can use Connected Services to configure the default Azure provisioning settings. Select the app host project, right-click on the **Connected Services** node, and select **Azure Resource Provisioning Settings**:

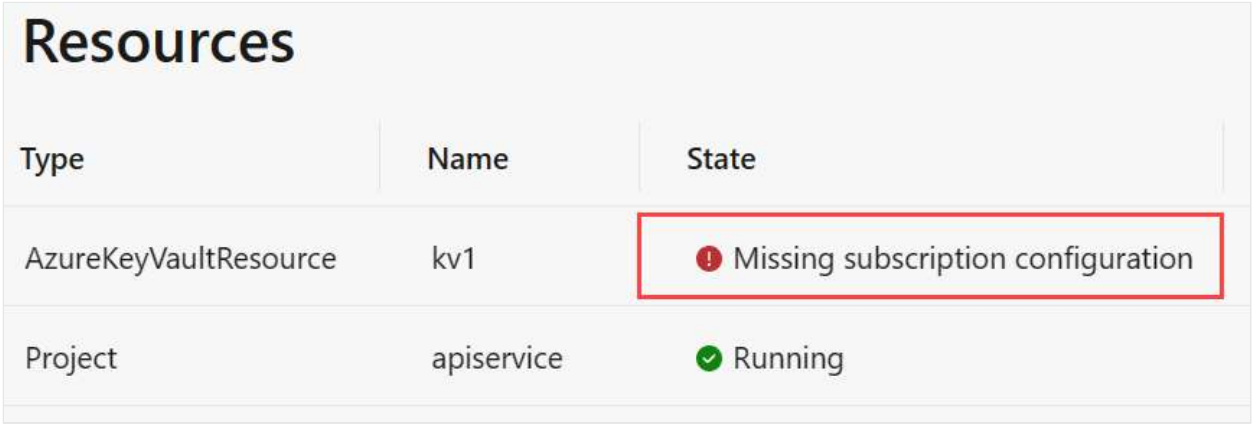


This will open a dialog where you can configure the Azure provisioning settings, as shown in the following screenshot:

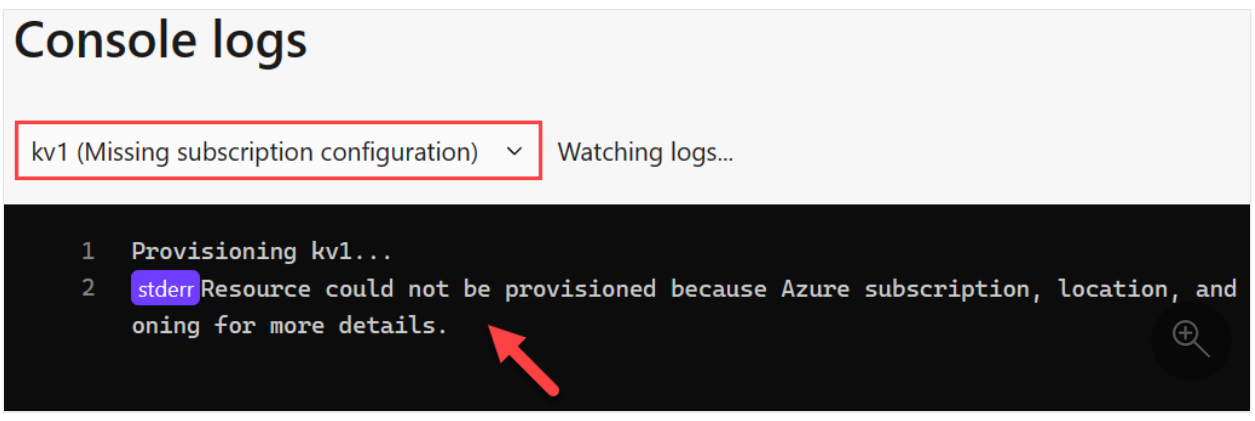


Missing configuration value hints

When the `Azure` configuration section is missing, has missing values, or is invalid, the .NET Aspire dashboard provides useful hints. For example, consider an app host that's missing the `SubscriptionId` configuration value that's attempting to use an Azure Key Vault resource. The **Resources** page indicates the **State** as **Missing subscription configuration**:



Additionally, the **Console logs** display this information as well, consider the following screenshot:



Known limitations

After provisioning Azure resources in this way, you must manually clean up the resources in the Azure portal as .NET Aspire doesn't provide a built-in mechanism to delete Azure resources. The easiest way to achieve this is by deleting the configured resource group. This can be done in the Azure portal or by using the Azure CLI:

Azure CLI

```
az group delete --name <ResourceGroupName>
```

Replace `<ResourceGroupName>` with the name of the resource group you want to delete. For more information, see az group delete.

.NET Aspire Azure AI Search integration

Article • 03/10/2025

Includes: ✅ Hosting integration and ✅ Client integration

The .NET Aspire Azure AI Search Documents integration enables you to connect to Azure AI Search (formerly Azure Cognitive Search) services from your .NET applications. Azure AI Search is an enterprise-ready information retrieval system for your heterogeneous content that you ingest into a search index, and surface to users through queries and apps. It comes with a comprehensive set of advanced search technologies, built for high-performance applications at any scale.

Hosting integration

The .NET Aspire Azure AI Search hosting integration models the Azure AI Search resource as the AzureSearchResource type. To access this type and APIs for expressing them within your app host project, install the Aspire.Hosting.Azure.Search NuGet package:

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Search
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add an Azure AI Search resource

To add an AzureSearchResource to your app host project, call the AddAzureSearch method providing a name:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var search = builder.AddAzureSearch("search");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(search);
```

```
// After adding all resources, run the app...
```

The preceding code adds an Azure AI Search resource named `search` to the app host project. The WithReference method passes the connection information to the `ExampleProject` project.

> **ⓘ Important**
>
> When you call AddAzureSearch, it implicitly calls AddAzureProvisioning(IDistributedApplicationBuilder)—which adds support for generating Azure resources dynamically during app startup. The app must configure the appropriate subscription and location. For more information, see Local provisioning: Configuration

Generated provisioning Bicep

If you're new to Bicep, it's a domain-specific language for defining Azure resources. With .NET Aspire, you don't need to write Bicep by hand; instead, the provisioning APIs generate Bicep for you. When you publish your app, the generated Bicep is output alongside the manifest file. When you add an Azure AI Search resource, Bicep is generated to provision the search service with appropriate defaults.

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location

param principalType string

param principalId string

resource search 'Microsoft.Search/searchServices@2023-11-01' = {
  name: take('search-${uniqueString(resourceGroup().id)}', 60)
  location: location
  properties: {
    hostingMode: 'default'
    disableLocalAuth: true
    partitionCount: 1
    replicaCount: 1
  }
  sku: {
    name: 'basic'
  }
  tags: {
    'aspire-resource-name': 'search'
```

```
  }
}

resource search_SearchIndexDataContributor
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(search.id, principalId,
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '8ebe5a00-
799e-43f5-93ac-243d3dce84a7'))
  properties: {
    principalId: principalId
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '8ebe5a00-
799e-43f5-93ac-243d3dce84a7')
    principalType: principalType
  }
  scope: search
}

resource search_SearchServiceContributor
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(search.id, principalId,
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '7ca78c08-
252a-4471-8644-bb5ff32d4ba0'))
  properties: {
    principalId: principalId
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '7ca78c08-
252a-4471-8644-bb5ff32d4ba0')
    principalType: principalType
  }
  scope: search
}

output connectionString string =
'Endpoint=https://${search.name}.search.windows.net'
```

The preceding Bicep is a module that provisions an Azure AI Search service resource with the following defaults:

- `location`: The location parameter of the resource group, defaults to `resourceGroup().location`.
- `principalType`: The principal type parameter of the Azure AI Search resource.
- `principalId`: The principal ID parameter of the Azure AI Search resource.
- `search`: The resource representing the Azure AI Search service.
  - `properties`: The properties of the Azure AI Search service:
    - `hostingMode`: Is set to `default`.
    - `disableLocalAuth`: Is set to `true`.
    - `partitionCount`: Is set to `1`.
    - `replicaCount`: Is set to `1`.

    - `sku`: Defaults to `basic`.
- `search_SearchIndexDataContributor`: The role assignment for the Azure AI Search index data contributor role. For more information, see Search Index Data Contributor.
- `search_SearchServiceContributor`: The role assignment for the Azure AI Search service contributor role. For more information, see Search Service Contributor.
- `connectionString`: The connection string for the Azure AI Search service, which is used to connect to the service. The connection string is generated using the `Endpoint` property of the Azure AI Search service.

The generated Bicep is a starting point and is influenced by changes to the provisioning infrastructure in C#. Customizations to the Bicep file directly will be overwritten, so make changes through the C# provisioning APIs to ensure they are reflected in the generated files.

Customize provisioning infrastructure

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This type enables the customization of the generated Bicep by providing a fluent API to configure the Azure resources—using the ConfigureInfrastructure<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API. For example, you can configure the search service partitions, replicas, and more:

C#

```
builder.AddAzureSearch("search")
    .ConfigureInfrastructure(infra =>
    {
        var searchService = infra.GetProvisionableResources()
                                  .OfType<SearchService>()
                                  .Single();

        searchService.PartitionCount = 6;
        searchService.ReplicaCount = 3;
        searchService.SearchSkuName = SearchServiceSkuName.Standard3;
        searchService.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
    - The `infra` parameter is an instance of the AzureResourceInfrastructure type.
    - The provisionable resources are retrieved by calling the GetProvisionableResources() method.

- The single SearchService resource is retrieved.
  - The SearchService.PartitionCount is set to `6`.
  - The SearchService.ReplicaCount is set to `3`.
  - The SearchService.SearchSkuName is set to SearchServiceSkuName.Standard3.
  - A tag is added to the Cognitive Services resource with a key of `ExampleKey` and a value of `Example value`.

There are many more configuration options available to customize the Azure AI Search resource. For more information, see Azure.Provisioning customization.

Connect to an existing Azure AI Search service

You might have an existing Azure AI Search service that you want to connect to. You can chain a call to annotate that your AzureSearchResource is an existing resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var existingSearchName = builder.AddParameter("existingSearchName");
var existingSearchResourceGroup =
builder.AddParameter("existingSearchResourceGroup");

var search = builder.AddAzureSearch("search")
                    .AsExisting(existingSearchName,
existingSearchResourceGroup);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(search);

// After adding all resources, run the app...
```

For more information on treating Azure AI Search resources as existing resources, see Use existing Azure resources.

Alternatively, instead of representing an Azure AI Search resource, you can add a connection string to the app host. Which is a weakly-typed approach that's based solely on a `string` value. To add a connection to an existing Azure AI Search service, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var search = builder.ExecutionContext.IsPublishMode
```

```
    ? builder.AddAzureSearch("search")
    : builder.AddConnectionString("search");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(search);

// After adding all resources, run the app...
```

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section:

JSON

```json
{
  "ConnectionStrings": {
    "search": "https://{account_name}.search.azure.com/"
  }
}
```

For more information, see Add existing Azure resources with connection strings.

Hosting integration health checks

The Azure AI Search hosting integration doesn't currently implement any health checks. This limitation is subject to change in future releases. As always, feel free to open an issue if you have any suggestions or feedback.

Client integration

To get started with the .NET Aspire Azure AI Search Documents client integration, install the Aspire.Azure.Search.Documents NuGet package in the client-consuming project, that is, the project for the application that uses the Azure AI Search Documents client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Azure.Search.Documents
```

Add an Azure AI Search index client

In the *Program.cs* file of your client-consuming project, call the AddAzureSearchClient extension method on any IHostApplicationBuilder to register a SearchIndexClient for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddAzureSearchClient(connectionName: "search");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Azure AI Search resource in the app host project. For more information, see **Add an Azure AI Search resource**.

After adding the `SearchIndexClient`, you can retrieve the client instance using dependency injection. For example, to retrieve the client instance from an example service:

C#

```
public class ExampleService(SearchIndexClient indexClient)
{
    // Use indexClient
}
```

You can also retrieve a `SearchClient` which can be used for querying, by calling the GetSearchClient(String) method:

C#

```
public class ExampleService(SearchIndexClient indexClient)
{
    public async Task<long> GetDocumentCountAsync(
        string indexName,
```

```
        CancellationToken cancellationToken)
    {
        var searchClient = indexClient.GetSearchClient(indexName);

        var documentCountResponse = await
searchClient.GetDocumentCountAsync(
            cancellationToken);

        return documentCountResponse.Value;
    }
}
```

For more information, see:

- Azure AI Search client library for .NET samples using the SearchIndexClient.
- Dependency injection in .NET for details on dependency injection.

Add keyed Azure AI Search index client

There might be situations where you want to register multiple `SearchIndexClient` instances with different connection names. To register keyed Azure AI Search clients, call the AddKeyedAzureSearchClient method:

C#

```
builder.AddKeyedAzureSearchClient(name: "images");
builder.AddKeyedAzureSearchClient(name: "documents");
```

> **ⓘ Important**
>
> When using keyed services, it's expected that your Azure AI Search resource configured two named connections, one for the `images` and one for the `documents`.

Then you can retrieve the client instances using dependency injection. For example, to retrieve the clients from a service:

C#

```
public class ExampleService(
    [KeyedService("images")] SearchIndexClient imagesClient,
    [KeyedService("documents")] SearchIndexClient documentsClient)
{
    // Use clients...
}
```

For more information, see Keyed services in .NET.

Configuration

The .NET Aspire Azure AI Search Documents library provides multiple options to configure the Azure AI Search connection based on the requirements and conventions of your project. Either an `Endpoint` or a `ConnectionString` is required to be supplied.

Use a connection string

A connection can be constructed from the **Keys and Endpoint** tab with the format `Endpoint={endpoint};Key={key};`. You can provide the name of the connection string when calling `builder.AddAzureSearchClient()`:

C#

```
builder.AddAzureSearchClient("searchConnectionName");
```

The connection string is retrieved from the `ConnectionStrings` configuration section. Two connection formats are supported:

Account endpoint

The recommended approach is to use an `Endpoint`, which works with the `AzureSearchSettings.Credential` property to establish a connection. If no credential is configured, the DefaultAzureCredential is used.

JSON

```
{
  "ConnectionStrings": {
    "search": "https://{search_service}.search.windows.net/"
  }
}
```

Connection string

Alternatively, a connection string with key can be used, however; it's not the recommended approach:

JSON

```
{
  "ConnectionStrings": {
    "search": "Endpoint=https://{search_service}.search.windows.net/;Key=
{account_key};"
  }
}
```

Use configuration providers

The .NET Aspire Azure AI Search library supports Microsoft.Extensions.Configuration. It loads the `AzureSearchSettings` and `SearchClientOptions` from configuration by using the `Aspire:Azure:Search:Documents` key. Example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "Azure": {
      "Search": {
        "Documents": {
          "DisableTracing": false
        }
      }
    }
  }
}
```

For the complete Azure AI Search Documents client integration JSON schema, see Aspire.Azure.Search.Documents/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<AzureSearchSettings> configureSettings` delegate to set up some or all the options inline, for example to disable tracing from code:

C#

```
builder.AddAzureSearchClient(
    "searchConnectionName",
    static settings => settings.DisableTracing = true);
```

You can also set up the SearchClientOptions using the optional `Action<IAzureClientBuilder<SearchIndexClient, SearchClientOptions>>`

`configureClientBuilder` parameter of the `AddAzureSearchClient` method. For example, to set the client ID for this client:

```
C#
```

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

Tracing

The .NET Aspire Azure AI Search Documents integration emits tracing activities using OpenTelemetry when interacting with the search service.

See also

- Azure AI Search
- .NET Aspire integrations overview
- .NET Aspire Azure integrations overview
- .NET Aspire GitHub repo

.NET Aspire Azure Cache for Redis® integration

Article • 02/11/2025

Includes: ✅ Hosting integration and ✅ Client integration

Azure Cache for Redis provides an in-memory data store based on the Redis software. Redis improves the performance and scalability of an application that uses backend data stores heavily. It's able to process large volumes of application requests by keeping frequently accessed data in the server memory, which can be written to and read from quickly. Redis brings a critical low-latency and high-throughput data storage solution to modern applications.

Azure Cache for Redis offers both the Redis open-source (OSS Redis) and a commercial product from Redis Inc. (Redis Enterprise) as a managed service. It provides secure and dedicated Redis server instances and full Redis API compatibility. Microsoft operates the service, hosted on Azure, and usable by any application within or outside of Azure.

The .NET Aspire Azure Cache for Redis integration enables you to connect to existing Azure Cache for Redis instances, or create new instances, or run as a container locally from .NET with the docker.io/library/redis container image.

Hosting integration

The .NET Aspire Azure Cache for Redis hosting integration models an Azure Redis resource as the AzureRedisCacheResource type. To access this type and APIs for expressing them as resources in your app host project, add the Aspire.Hosting.Azure.Redis NuGet package:

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Redis
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Azure Cache for Redis resource

In your app host project, call AddAzureRedis on the `builder` instance to add an Azure Cache for Redis resource, as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddAzureRedis("azcache");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The preceding call to `AddAzureRedis` configures the Redis server resource to be deployed as an Azure Cache for Redis.

> **Important**
>
> By default, `AddAzureRedis` configures Microsoft Entra ID authentication. This requires changes to applications that need to connect to these resources, for example, client integrations.

> **Tip**
>
> When you call AddAzureRedis, it implicitly calls AddAzureProvisioning—which adds support for generating Azure resources dynamically during app startup. The app must configure the appropriate subscription and location. For more information, see Local provisioning: Configuration.

Generated provisioning Bicep

If you're new to Bicep, it's a domain-specific language for defining Azure resources. With .NET Aspire, you don't need to write Bicep by-hand, instead the provisioning APIs generate Bicep for you. When you publish your app, the generated Bicep is output alongside the manifest file. When you add an Azure Cache for Redis resource, the following Bicep is generated:

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location
```

```
param principalId string

param principalName string

resource redis 'Microsoft.Cache/redis@2024-03-01' = {
  name: take('redis-${uniqueString(resourceGroup().id)}', 63)
  location: location
  properties: {
    sku: {
      name: 'Basic'
      family: 'C'
      capacity: 1
    }
    enableNonSslPort: false
    disableAccessKeyAuthentication: true
    minimumTlsVersion: '1.2'
    redisConfiguration: {
      'aad-enabled': 'true'
    }
  }
  tags: {
    'aspire-resource-name': 'redis'
  }
}

resource redis_contributor
'Microsoft.Cache/redis/accessPolicyAssignments@2024-03-01' = {
  name: take('rediscontributor${uniqueString(resourceGroup().id)}', 24)
  properties: {
    accessPolicyName: 'Data Contributor'
    objectId: principalId
    objectIdAlias: principalName
  }
  parent: redis
}

output connectionString string = '${redis.properties.hostName},ssl=true'
```

The preceding Bicep is a module that provisions an Azure Cache for Redis with the following defaults:

- `location`: The location of the Azure Cache for Redis resource. The default is the location of the resource group.
- `principalId`: The principal ID of the Azure Cache for Redis resource.
- `principalName`: The principal name of the Azure Cache for Redis resource.
- `sku`: The SKU of the Azure Cache for Redis resource. The default is `Basic` with a capacity of `1`.
- `enableNonSslPort`: The non-SSL port of the Azure Cache for Redis resource. The default is `false`.

- `disableAccessKeyAuthentication`: The access key authentication of the Azure Cache for Redis resource. The default is `true`.
- `minimumTlsVersion`: The minimum TLS version of the Azure Cache for Redis resource. The default is `1.2`.
- `redisConfiguration`: The Redis configuration of the Azure Cache for Redis resource. The default is `aad-enabled` set to `true`.
- `tags`: The tags of the Azure Cache for Redis resource. The default is `aspire-resource-name` set to the name of the Aspire resource, in this case `redis`.
- `redis_contributor`: The contributor of the Azure Cache for Redis resource, with an access policy name of `Data Contributor`.
- `connectionString`: The connection string of the Azure Cache for Redis resource.

In addition to the Azure Cache for Redis, it also provisions an access policy assignment to the application access to the cache. The generated Bicep is a starting point and is influenced by changes to the provisioning infrastructure in C#. Customizations to the Bicep file directly will be overwritten, so make changes through the C# provisioning APIs to ensure they are reflected in the generated files.

Customize provisioning infrastructure

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This type enables the customization of the generated Bicep by providing a fluent API to configure the Azure resources—using the ConfigureInfrastructure<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API. For example, you can configure the `kind`, `consistencyPolicy`, `locations`, and more. The following example demonstrates how to customize the Azure Cache for Redis resource:

C#

```
builder.AddAzureRedis("redis")
    .WithAccessKeyAuthentication()
    .ConfigureInfrastructure(infra =>
    {
        var redis = infra.GetProvisionableResources()
                         .OfType<RedisResource>()
                         .Single();

        redis.Sku = new()
        {
            Family = RedisSkuFamily.BasicOrStandard,
            Name = RedisSkuName.Standard,
            Capacity = 1,
        };
```

```
        redis.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
  - The `infra` parameter is an instance of the AzureResourceInfrastructure type.
  - The provisionable resources are retrieved by calling the GetProvisionableResources() method.
  - The single RedisResource is retrieved.
  - The `Sku` is set with a family of `BasicOrStandard`, a name of `Standard`, and a capacity of `1`.
  - A tag is added to the Redis resource with a key of `ExampleKey` and a value of `Example value`.

There are many more configuration options available to customize the Azure Cache for Redis resource. For more information, see Azure.Provisioning.Redis. For more information, see Azure.Provisioning customization.

Connect to an existing Azure Cache for Redis

You might have an existing Azure Cache for Redis that you want to connect to. Instead of representing a new Azure Cache for Redis resource, you can add a connection string to the app host. To add a connection to an existing Azure Cache for Redis, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddConnectionString("azure-redis");

builder.AddProject<Projects.WebApplication>("web")
       .WithReference(cache);

// After adding all resources, run the app...
```

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other

services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section. The app host injects this connection string as an environment variable into all dependent resources, for example:

JSON

```
{
    "ConnectionStrings": {
        "azure-redis": "<your-redis-
name>.redis.cache.windows.net:6380,ssl=true,abortConnect=False"
    }
}
```

The dependent resource can access the injected connection string by calling the GetConnectionString method, and passing the connection name as the parameter, in this case `"azure-redis"`. The `GetConnectionString` API is shorthand for `IConfiguration.GetSection("ConnectionStrings")[name]`.

Run Azure Cache for Redis resource as a container

The Azure Cache for Redis hosting integration supports running the Redis server as a local container. This is beneficial for situations where you want to run the Redis server locally for development and testing purposes, avoiding the need to provision an Azure resource or connect to an existing Azure Cache for Redis server.

To make use of the docker.io/library/redis container image, and run the Azure Cache for Redis instance as a container locally, chain a call to RunAsContainer, as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddAzureRedis("azcache")
                   .RunAsContainer();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The preceding code configures the Redis resource to run locally in a container.

> **Tip**
>
> The `RunAsContainer` method is useful for local development and testing. The API exposes an optional delegate that enables you to customize the underlying RedisResource configuration, such adding Redis Insights, Redis Commander, adding a data volume or data bind mount. For more information, see the .NET Aspire Redis hosting integration.

Configure the Azure Cache for Redis resource to use access key authentication

By default, the Azure Cache for Redis resource is configured to use Microsoft Entra ID authentication. If you want to use password authentication (not recommended), you can configure the server to use password authentication by calling the WithAccessKeyAuthentication method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddAzureRedis("azcache")
                   .WithAccessKeyAuthentication();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The preceding code configures the Azure Cache for Redis resource to use access key authentication. This alters the generated Bicep to use access key authentication instead of Microsoft Entra ID authentication. In other words, the connection string will contain a password, and will be added to an Azure Key Vault secret.

Client integration

To get started with the .NET Aspire Stack Exchange Redis client integration, install the Aspire.StackExchange.Redis NuGet package in the client-consuming project, that is, the project for the application that uses the Redis client. The Redis client integration registers an IConnectionMultiplexer instance that you can use to interact with Redis.

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis
```

Add Redis client

In the *Program.cs* file of your client-consuming project, call the AddRedisClient extension method on any IHostApplicationBuilder to register an `IConnectionMultiplexer` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddRedisClient(connectionName: "cache");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Azure Cache for Redis resource in the app host project. For more information, see **Add Azure Cache for Redis resource**.

You can then retrieve the `IConnectionMultiplexer` instance using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(IConnectionMultiplexer connectionMux)
{
    // Use connection multiplexer...
}
```

For more information on dependency injection, see .NET dependency injection.

Add Azure Cache for Redis authenticated client

By default, when you call AddAzureRedis in your Redis hosting integration, it configures Microsoft Entra ID. Install the Microsoft.Azure.StackExchangeRedis NuGet package to enable authentication:

.NET CLI

```
dotnet add package Microsoft.Azure.StackExchangeRedis
```

The Redis connection can be consumed using the client integration and `Microsoft.Azure.StackExchangeRedis`. Consider the following configuration code:

C#

```
var azureOptionsProvider = new AzureOptionsProvider();

var configurationOptions = ConfigurationOptions.Parse(
    builder.Configuration.GetConnectionString("cache") ??
    throw new InvalidOperationException("Could not find a 'cache' connection
string."));

if (configurationOptions.EndPoints.Any(azureOptionsProvider.IsMatch))
{
    await configurationOptions.ConfigureForAzureWithTokenCredentialAsync(
        new DefaultAzureCredential());
}

builder.AddRedisClient("cache", configureOptions: options =>
{
    options.Defaults = configurationOptions.Defaults;
});
```

For more information, see the Microsoft.Azure.StackExchangeRedis repo.

Add keyed Redis client

There might be situations where you want to register multiple `IConnectionMultiplexer` instances with different connection names. To register keyed Redis clients, call the AddKeyedRedisClient method:

C#

```
builder.AddKeyedRedisClient(name: "chat");
builder.AddKeyedRedisClient(name: "queue");
```

Then you can retrieve the `IConnectionMultiplexer` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("chat")] IConnectionMultiplexer chatConnectionMux,
    [FromKeyedServices("queue")] IConnectionMultiplexer queueConnectionMux)
{
    // Use connections...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire Stack Exchange Redis client integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling AddRedis:

C#

```
builder.AddRedis("cache");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the Stack Exchange Redis configuration docs.

Use configuration providers

The .NET Aspire Stack Exchange Redis integration supports Microsoft.Extensions.Configuration. It loads the StackExchangeRedisSettings from configuration by using the `Aspire:StackExchange:Redis` key. Example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        },
        "DisableHealthChecks": true,
        "DisableTracing": false
      }
    }
  }
}
```

For the complete Redis client integration JSON schema, see Aspire.StackExchange.Redis/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings>` delegate to set up some or all the options inline, for example to configure `DisableTracing`:

C#

```
builder.AddRedisClient(
    "cache",
    static settings => settings.DisableTracing = true);
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

The .NET Aspire Stack Exchange Redis integration handles the following:

- Adds the health check when StackExchangeRedisSettings.DisableHealthChecks is `false`, which attempts to connect to the container instance.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Stack Exchange Redis integration uses the following log categories:

- `Aspire.StackExchange.Redis`

Tracing

The .NET Aspire Stack Exchange Redis integration will emit the following tracing activities using OpenTelemetry:

- `OpenTelemetry.Instrumentation.StackExchangeRedis`

Metrics

The .NET Aspire Stack Exchange Redis integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- Azure Cache for Redis docs
- Stack Exchange Redis docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

> *: *Redis is a registered trademark of Redis Ltd. Any rights therein are reserved to Redis Ltd. Any use by Microsoft is for referential purposes only and does not indicate any sponsorship, endorsement or affiliation between Redis and Microsoft.* *Return to top**?*

.NET Aspire Azure Cache for Redis® distributed caching integration

Article • 02/11/2025

Includes:

In your app host project, call AddAzureRedis on the `builder` instance to add an Azure Cache for Redis resource, as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddAzureRedis("azcache");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The preceding call to `AddAzureRedis` configures the Redis server resource to be deployed as an Azure Cache for Redis.

> **Important**
>
> By default, `AddAzureRedis` configures Microsoft Entra ID authentication. This requires changes to applications that need to connect to these resources, for example, client integrations.

> **Tip**
>
> When you call AddAzureRedis, it implicitly calls AddAzureProvisioning—which adds support for generating Azure resources dynamically during app startup. The app must configure the appropriate subscription and location. For more information, see Local provisioning: Configuration.

Generated provisioning Bicep

If you're new to Bicep, it's a domain-specific language for defining Azure resources. With .NET Aspire, you don't need to write Bicep by-hand, instead the provisioning APIs generate Bicep for you. When you publish your app, the generated Bicep is output alongside the manifest file. When you add an Azure Cache for Redis resource, the following Bicep is generated:

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location
```

```
param principalId string

param principalName string

resource redis 'Microsoft.Cache/redis@2024-03-01' = {
  name: take('redis-${uniqueString(resourceGroup().id)}', 63)
  location: location
  properties: {
    sku: {
      name: 'Basic'
      family: 'C'
      capacity: 1
    }
    enableNonSslPort: false
    disableAccessKeyAuthentication: true
    minimumTlsVersion: '1.2'
    redisConfiguration: {
      'aad-enabled': 'true'
    }
  }
  tags: {
    'aspire-resource-name': 'redis'
  }
}

resource redis_contributor
'Microsoft.Cache/redis/accessPolicyAssignments@2024-03-01' = {
  name: take('rediscontributor${uniqueString(resourceGroup().id)}', 24)
  properties: {
    accessPolicyName: 'Data Contributor'
    objectId: principalId
    objectIdAlias: principalName
  }
  parent: redis
}

output connectionString string = '${redis.properties.hostName},ssl=true'
```

The preceding Bicep is a module that provisions an Azure Cache for Redis with the following defaults:

- `location`: The location of the Azure Cache for Redis resource. The default is the location of the resource group.
- `principalId`: The principal ID of the Azure Cache for Redis resource.
- `principalName`: The principal name of the Azure Cache for Redis resource.
- `sku`: The SKU of the Azure Cache for Redis resource. The default is `Basic` with a capacity of `1`.
- `enableNonSslPort`: The non-SSL port of the Azure Cache for Redis resource. The default is `false`.

- `disableAccessKeyAuthentication`: The access key authentication of the Azure Cache for Redis resource. The default is `true`.
- `minimumTlsVersion`: The minimum TLS version of the Azure Cache for Redis resource. The default is `1.2`.
- `redisConfiguration`: The Redis configuration of the Azure Cache for Redis resource. The default is `aad-enabled` set to `true`.
- `tags`: The tags of the Azure Cache for Redis resource. The default is `aspire-resource-name` set to the name of the Aspire resource, in this case `redis`.
- `redis_contributor`: The contributor of the Azure Cache for Redis resource, with an access policy name of `Data Contributor`.
- `connectionString`: The connection string of the Azure Cache for Redis resource.

In addition to the Azure Cache for Redis, it also provisions an access policy assignment to the application access to the cache. The generated Bicep is a starting point and is influenced by changes to the provisioning infrastructure in C#. Customizations to the Bicep file directly will be overwritten, so make changes through the C# provisioning APIs to ensure they are reflected in the generated files.

Customize provisioning infrastructure

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This type enables the customization of the generated Bicep by providing a fluent API to configure the Azure resources—using the ConfigureInfrastructure<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API. For example, you can configure the `kind`, `consistencyPolicy`, `locations`, and more. The following example demonstrates how to customize the Azure Cache for Redis resource:

C#

```
builder.AddAzureRedis("redis")
    .WithAccessKeyAuthentication()
    .ConfigureInfrastructure(infra =>
    {
        var redis = infra.GetProvisionableResources()
                         .OfType<RedisResource>()
                         .Single();

        redis.Sku = new()
        {
            Family = RedisSkuFamily.BasicOrStandard,
            Name = RedisSkuName.Standard,
            Capacity = 1,
        };
```

```
        redis.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
  - The `infra` parameter is an instance of the AzureResourceInfrastructure type.
  - The provisionable resources are retrieved by calling the GetProvisionableResources() method.
  - The single RedisResource is retrieved.
  - The `Sku` is set with a family of `BasicOrStandard`, a name of `Standard`, and a capacity of `1`.
  - A tag is added to the Redis resource with a key of `ExampleKey` and a value of `Example value`.

There are many more configuration options available to customize the Azure Cache for Redis resource. For more information, see Azure.Provisioning.Redis. For more information, see Azure.Provisioning customization.

Connect to an existing Azure Cache for Redis

You might have an existing Azure Cache for Redis that you want to connect to. Instead of representing a new Azure Cache for Redis resource, you can add a connection string to the app host. To add a connection to an existing Azure Cache for Redis, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddConnectionString("azure-redis");

builder.AddProject<Projects.WebApplication>("web")
       .WithReference(cache);

// After adding all resources, run the app...
```

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other

services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section. The app host injects this connection string as an environment variable into all dependent resources, for example:

JSON

```
{
    "ConnectionStrings": {
        "azure-redis": "<your-redis-
name>.redis.cache.windows.net:6380,ssl=true,abortConnect=False"
    }
}
```

The dependent resource can access the injected connection string by calling the GetConnectionString method, and passing the connection name as the parameter, in this case `"azure-redis"`. The `GetConnectionString` API is shorthand for `IConfiguration.GetSection("ConnectionStrings")[name]`.

Run Azure Cache for Redis resource as a container

The Azure Cache for Redis hosting integration supports running the Redis server as a local container. This is beneficial for situations where you want to run the Redis server locally for development and testing purposes, avoiding the need to provision an Azure resource or connect to an existing Azure Cache for Redis server.

To make use of the docker.io/library/redis container image, and run the Azure Cache for Redis instance as a container locally, chain a call to RunAsContainer, as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddAzureRedis("azcache")
                   .RunAsContainer();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The preceding code configures the Redis resource to run locally in a container.

> **Tip**
>
> The `RunAsContainer` method is useful for local development and testing. The API exposes an optional delegate that enables you to customize the underlying RedisResource configuration, such adding Redis Insights, Redis Commander, adding a data volume or data bind mount. For more information, see the .NET Aspire Redis hosting integration.

Configure the Azure Cache for Redis resource to use access key authentication

By default, the Azure Cache for Redis resource is configured to use Microsoft Entra ID authentication. If you want to use password authentication (not recommended), you can configure the server to use password authentication by calling the WithAccessKeyAuthentication method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddAzureRedis("azcache")
                   .WithAccessKeyAuthentication();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The preceding code configures the Azure Cache for Redis resource to use access key authentication. This alters the generated Bicep to use access key authentication instead of Microsoft Entra ID authentication. In other words, the connection string will contain a password, and will be added to an Azure Key Vault secret.

Client integration

To get started with the .NET Aspire Redis distributed caching integration, install the Aspire.StackExchange.Redis.DistributedCaching NuGet package in the client-consuming project, i.e., the project for the application that uses the Redis distributed caching client. The Redis client integration registers an IDistributedCache instance that you can use to interact with Redis.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.DistributedCaching
```

Add Redis distributed cache client

In the *Program.cs* file of your client-consuming project, call the AddRedisDistributedCache extension to register the required services for distributed caching and add a IDistributedCache for use via the dependency injection container.

C#

```
builder.AddRedisDistributedCache(connectionName: "cache");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Azure Cache for Redis resource in the app host project. For more information, see **Add Azure Cache for Redis resource**.

You can then retrieve the `IDistributedCache` instance using dependency injection. For example, to retrieve the cache from a service:

C#

```
public class ExampleService(IDistributedCache cache)
{
    // Use cache...
}
```

For more information on dependency injection, see .NET dependency injection.

Add Azure Cache for Redis authenticated distributed client

By default, when you call AddAzureRedis in your app host project, the Redis hosting integration configures Microsoft.Azure.StackExchangeRedis NuGet package to enable authentication:

.NET CLI

```
dotnet add package Microsoft.Azure.StackExchangeRedis
```

The Redis connection can be consumed using the client integration and `Microsoft.Azure.StackExchangeRedis`. Consider the following configuration code:

C#

```
var azureOptionsProvider = new AzureOptionsProvider();

var configurationOptions = ConfigurationOptions.Parse(
    builder.Configuration.GetConnectionString("cache") ??
    throw new InvalidOperationException("Could not find a 'cache' connection
string."));

if (configurationOptions.EndPoints.Any(azureOptionsProvider.IsMatch))
{
    await configurationOptions.ConfigureForAzureWithTokenCredentialAsync(
        new DefaultAzureCredential());
}

builder.AddRedisDistributedCache("cache", configureOptions: options =>
{
    options.Defaults = configurationOptions.Defaults;
});
```

For more information, see the Microsoft.Azure.StackExchangeRedis repo.

Add keyed Redis client

There might be situations where you want to register multiple `IDistributedCache` instances with different connection names. To register keyed Redis clients, call the AddKeyedRedisDistributedCache method:

C#

```
builder.AddKeyedRedisDistributedCache(name: "chat");
builder.AddKeyedRedisDistributedCache(name: "product");
```

Then you can retrieve the `IDistributedCache` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("chat")] IDistributedCache chatCache,
    [FromKeyedServices("product")] IDistributedCache productCache)
{
    // Use caches...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire Redis distributed caching integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddRedisDistributedCache`:

C#

```
builder.AddRedisDistributedCache("cache");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the Stack Exchange Redis configuration docs.

Use configuration providers

The .NET Aspire Stack Exchange Redis distributed caching integration supports Microsoft.Extensions.Configuration. It loads the StackExchangeRedisSettings from configuration by using the `Aspire:StackExchange:Redis` key. Example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        },
        "DisableHealthChecks": true,
        "DisableTracing": false
      }
    }
  }
}
```

For the complete Redis distributed caching client integration JSON schema, see Aspire.StackExchange.Redis.DistributedCaching/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings>` delegate to set up some or all the options inline, for example to configure `DisableTracing`:

C#

```
builder.AddRedisDistributedCache(
    "cache",
    settings => settings.DisableTracing = true);
```

You can also set up the ConfigurationOptions using the `Action<ConfigurationOptions> configureOptions` delegate parameter of the `AddRedisDistributedCache` method. For example to set the connection timeout:

C#

```
builder.AddRedisDistributedCache(
    "cache",
    static settings => settings.ConnectTimeout = 3_000);
```

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

The .NET Aspire Redis distributed caching integration handles the following:

- Adds the health check when StackExchangeRedisSettings.DisableHealthChecks is `false`, which attempts to connect to the container instance.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not

The .NET Aspire Redis Distributed caching integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- Azure Cache for Redis docs
- Stack Exchange Redis docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

> *: *Redis is a registered trademark of Redis Ltd. Any rights therein are reserved to Redis Ltd. Any use by Microsoft is for referential purposes only and does not indicate any sponsorship, endorsement or affiliation between Redis and Microsoft.* *Return to top*?

.NET Aspire Azure Cache for Redis® output caching integration

Article • 02/11/2025

Includes: ✅ Hosting integration and ✅ Client integration

Azure Cache for Redis provides an in-memory data store based on the Redis software. Redis improves the performance and scalability of an application that uses backend data stores heavily. It's able to process large volumes of application requests by keeping frequently accessed data in the server memory, which can be written to and read from quickly. Redis brings a critical low-latency and high-throughput data storage solution to modern applications.

Azure Cache for Redis offers both the Redis open-source (OSS Redis) and a commercial product from Redis Inc. (Redis Enterprise) as a managed service. It provides secure and dedicated Redis server instances and full Redis API compatibility. Microsoft operates the service, hosted on Azure, and usable by any application within or outside of Azure.

The .NET Aspire Azure Cache for Redis integration enables you to connect to existing Azure Cache for Redis instances, or create new instances from .NET with the docker.io/library/redis container image.

Hosting integration

The .NET Aspire Azure Cache for Redis hosting integration models an Azure Redis resource as the AzureRedisCacheResource type. To access this type and APIs for expressing them as resources in your app host project, add the Aspire.Hosting.Azure.Redis NuGet package:

.NET CLI

```
.NET CLI
dotnet add package Aspire.Hosting.Azure.Redis
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Azure Cache for Redis resource

In your app host project, call AddAzureRedis on the `builder` instance to add an Azure Cache for Redis resource, as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddAzureRedis("azcache");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The preceding call to `AddAzureRedis` configures the Redis server resource to be deployed as an Azure Cache for Redis.

> **Important**
>
> By default, `AddAzureRedis` configures Microsoft Entra ID authentication. This requires changes to applications that need to connect to these resources, for example, client integrations.

> **Tip**
>
> When you call AddAzureRedis, it implicitly calls AddAzureProvisioning—which adds support for generating Azure resources dynamically during app startup. The app must configure the appropriate subscription and location. For more information, see Local provisioning: Configuration.

Generated provisioning Bicep

If you're new to Bicep, it's a domain-specific language for defining Azure resources. With .NET Aspire, you don't need to write Bicep by-hand, instead the provisioning APIs generate Bicep for you. When you publish your app, the generated Bicep is output alongside the manifest file. When you add an Azure Cache for Redis resource, the following Bicep is generated:

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location
```

```
param principalId string

param principalName string

resource redis 'Microsoft.Cache/redis@2024-03-01' = {
  name: take('redis-${uniqueString(resourceGroup().id)}', 63)
  location: location
  properties: {
    sku: {
      name: 'Basic'
      family: 'C'
      capacity: 1
    }
    enableNonSslPort: false
    disableAccessKeyAuthentication: true
    minimumTlsVersion: '1.2'
    redisConfiguration: {
      'aad-enabled': 'true'
    }
  }
  tags: {
    'aspire-resource-name': 'redis'
  }
}

resource redis_contributor
'Microsoft.Cache/redis/accessPolicyAssignments@2024-03-01' = {
  name: take('rediscontributor${uniqueString(resourceGroup().id)}', 24)
  properties: {
    accessPolicyName: 'Data Contributor'
    objectId: principalId
    objectIdAlias: principalName
  }
  parent: redis
}

output connectionString string = '${redis.properties.hostName},ssl=true'
```

The preceding Bicep is a module that provisions an Azure Cache for Redis with the following defaults:

- `location`: The location of the Azure Cache for Redis resource. The default is the location of the resource group.
- `principalId`: The principal ID of the Azure Cache for Redis resource.
- `principalName`: The principal name of the Azure Cache for Redis resource.
- `sku`: The SKU of the Azure Cache for Redis resource. The default is `Basic` with a capacity of `1`.
- `enableNonSslPort`: The non-SSL port of the Azure Cache for Redis resource. The default is `false`.

- `disableAccessKeyAuthentication`: The access key authentication of the Azure Cache for Redis resource. The default is `true`.
- `minimumTlsVersion`: The minimum TLS version of the Azure Cache for Redis resource. The default is `1.2`.
- `redisConfiguration`: The Redis configuration of the Azure Cache for Redis resource. The default is `aad-enabled` set to `true`.
- `tags`: The tags of the Azure Cache for Redis resource. The default is `aspire-resource-name` set to the name of the Aspire resource, in this case `redis`.
- `redis_contributor`: The contributor of the Azure Cache for Redis resource, with an access policy name of `Data Contributor`.
- `connectionString`: The connection string of the Azure Cache for Redis resource.

In addition to the Azure Cache for Redis, it also provisions an access policy assignment to the application access to the cache. The generated Bicep is a starting point and is influenced by changes to the provisioning infrastructure in C#. Customizations to the Bicep file directly will be overwritten, so make changes through the C# provisioning APIs to ensure they are reflected in the generated files.

Customize provisioning infrastructure

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This type enables the customization of the generated Bicep by providing a fluent API to configure the Azure resources—using the ConfigureInfrastructure<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API. For example, you can configure the `kind`, `consistencyPolicy`, `locations`, and more. The following example demonstrates how to customize the Azure Cache for Redis resource:

C#

```
builder.AddAzureRedis("redis")
    .WithAccessKeyAuthentication()
    .ConfigureInfrastructure(infra =>
    {
        var redis = infra.GetProvisionableResources()
                         .OfType<RedisResource>()
                         .Single();

        redis.Sku = new()
        {
            Family = RedisSkuFamily.BasicOrStandard,
            Name = RedisSkuName.Standard,
            Capacity = 1,
        };
```

```
        redis.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
  - The `infra` parameter is an instance of the AzureResourceInfrastructure type.
  - The provisionable resources are retrieved by calling the GetProvisionableResources() method.
  - The single RedisResource is retrieved.
  - The `Sku` is set with a family of `BasicOrStandard`, a name of `Standard`, and a capacity of `1`.
  - A tag is added to the Redis resource with a key of `ExampleKey` and a value of `Example value`.

There are many more configuration options available to customize the Azure Cache for Redis resource. For more information, see Azure.Provisioning.Redis. For more information, see Azure.Provisioning customization.

Connect to an existing Azure Cache for Redis

You might have an existing Azure Cache for Redis that you want to connect to. Instead of representing a new Azure Cache for Redis resource, you can add a connection string to the app host. To add a connection to an existing Azure Cache for Redis, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddConnectionString("azure-redis");

builder.AddProject<Projects.WebApplication>("web")
       .WithReference(cache);

// After adding all resources, run the app...
```

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other

services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section. The app host injects this connection string as an environment variable into all dependent resources, for example:

JSON

```
{
    "ConnectionStrings": {
        "azure-redis": "<your-redis-
name>.redis.cache.windows.net:6380,ssl=true,abortConnect=False"
    }
}
```

The dependent resource can access the injected connection string by calling the GetConnectionString method, and passing the connection name as the parameter, in this case `"azure-redis"`. The `GetConnectionString` API is shorthand for `IConfiguration.GetSection("ConnectionStrings")[name]`.

Run Azure Cache for Redis resource as a container

The Azure Cache for Redis hosting integration supports running the Redis server as a local container. This is beneficial for situations where you want to run the Redis server locally for development and testing purposes, avoiding the need to provision an Azure resource or connect to an existing Azure Cache for Redis server.

To make use of the docker.io/library/redis container image, and run the Azure Cache for Redis instance as a container locally, chain a call to RunAsContainer, as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddAzureRedis("azcache")
                   .RunAsContainer();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The preceding code configures the Redis resource to run locally in a container.

> **Tip**
>
> The `RunAsContainer` method is useful for local development and testing. The API exposes an optional delegate that enables you to customize the underlying RedisResource configuration, such adding Redis Insights, Redis Commander, adding a data volume or data bind mount. For more information, see the .NET Aspire Redis hosting integration.

Configure the Azure Cache for Redis resource to use access key authentication

By default, the Azure Cache for Redis resource is configured to use Microsoft Entra ID authentication. If you want to use password authentication (not recommended), you can configure the server to use password authentication by calling the WithAccessKeyAuthentication method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddAzureRedis("azcache")
                   .WithAccessKeyAuthentication();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The preceding code configures the Azure Cache for Redis resource to use access key authentication. This alters the generated Bicep to use access key authentication instead of Microsoft Entra ID authentication. In other words, the connection string will contain a password, and will be added to an Azure Key Vault secret.

Client integration

To get started with the .NET Aspire Stack Exchange Redis output caching client integration, install the Aspire.StackExchange.Redis.OutputCaching NuGet package in the client-consuming project, that is, the project for the application that uses the output caching client. The Redis output caching client integration registers services

required for enabling CacheOutput method calls and [OutputCache] attribute usage to rely on Redis as its caching mechanism.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.OutputCaching
```

Add output caching

In the *Program.cs* file of your client-consuming project, call the AddRedisOutputCache extension method on any IHostApplicationBuilder to register the required services for output caching.

C#

```
builder.AddRedisOutputCache(connectionName: "cache");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Azure Cache for Redis resource in the app host project. For more information, see **Add Azure Cache for Redis resource**.

Add the middleware to the request processing pipeline by calling UseOutputCache(IApplicationBuilder):

C#

```
var app = builder.Build();

app.UseOutputCache();
```

For minimal API apps, configure an endpoint to do caching by calling CacheOutput, or by applying the OutputCacheAttribute, as shown in the following examples:

C#

```
app.MapGet("/cached", () => "Hello world!")
   .CacheOutput();
```

```
app.MapGet(
    "/attribute",
    [OutputCache] () => "Hello world!");
```

For apps with controllers, apply the `[OutputCache]` attribute to the action method. For Razor Pages apps, apply the attribute to the Razor page class.

Add Azure Cache for Redis authenticated output client

By default, when you call AddAzureRedis in your Redis hosting integration, it configures Microsoft.Azure.StackExchangeRedis NuGet package to enable authentication:

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Azure.StackExchangeRedis
```

The Redis connection can be consumed using the client integration and `Microsoft.Azure.StackExchangeRedis`. Consider the following configuration code:

C#

```
var azureOptionsProvider = new AzureOptionsProvider();

var configurationOptions = ConfigurationOptions.Parse(
    builder.Configuration.GetConnectionString("cache") ??
    throw new InvalidOperationException("Could not find a 'cache' connection
string."));

if (configurationOptions.EndPoints.Any(azureOptionsProvider.IsMatch))
{
    await configurationOptions.ConfigureForAzureWithTokenCredentialAsync(
        new DefaultAzureCredential());
}

builder.AddRedisOutputCache("cache", configureOptions: options =>
{
    options.Defaults = configurationOptions.Defaults;
});
```

For more information, see the Microsoft.Azure.StackExchangeRedis repo.

Configuration

The .NET Aspire Stack Exchange Redis output caching integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling AddRedisOutputCache:

C#

```
builder.AddRedisOutputCache(connectionName: "cache");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the Stack Exchange Redis configuration docs.

Use configuration providers

The .NET Aspire Stack Exchange Redis output caching integration supports Microsoft.Extensions.Configuration. It loads the StackExchangeRedisSettings from configuration by using the `Aspire:StackExchange:Redis` key. Example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        },
        "DisableHealthChecks": true,
```

```
        "DisableTracing": false
      }
    }
  }
}
```

For the complete Redis output caching client integration JSON schema, see Aspire.StackExchange.Redis.OutputCaching/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings> configurationSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddRedisOutputCache(
    "cache",
    static settings => settings.DisableHealthChecks  = true);
```

You can also set up the ConfigurationOptions using the `Action<ConfigurationOptions> configureOptions` delegate parameter of the AddRedisOutputCache method. For example to set the connection timeout:

C#

```
builder.AddRedisOutputCache(
    "cache",
    static settings => settings.ConnectTimeout = 3_000);
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

The .NET Aspire Stack Exchange Redis output caching integration handles the following:

- Adds the health check when StackExchangeRedisSettings.DisableHealthChecks is `false`, which attempts to connect to the container instance.

- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Stack Exchange Redis output caching integration uses the following Log categories:

- `Aspire.StackExchange.Redis`
- `Microsoft.AspNetCore.OutputCaching.StackExchangeRedis`

Tracing

The .NET Aspire Stack Exchange Redis output caching integration will emit the following Tracing activities using OpenTelemetry:

- `OpenTelemetry.Instrumentation.StackExchangeRedis`

Metrics

The .NET Aspire Stack Exchange Redis output caching integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- Azure Cache for Redis docs
- Stack Exchange Redis docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

> *: *Redis is a registered trademark of Redis Ltd. Any rights therein are reserved to Redis Ltd. Any use by Microsoft is for referential purposes only and does not indicate any sponsorship, endorsement or affiliation between Redis and Microsoft.* *Return to top**?*

.NET Aspire Azure Cosmos DB integration

Article • 02/26/2025

Includes: ✅ Hosting integration and ✅ Client integration

Azure Cosmos DB is a fully managed NoSQL database service for modern app development. The .NET Aspire Azure Cosmos DB integration enables you to connect to existing Cosmos DB instances or create new instances from .NET with the Azure Cosmos DB emulator.

Hosting integration

The .NET Aspire Azure Cosmos DB hosting integration models the various Cosmos DB resources as the following types:

- AzureCosmosDBResource: Represents an Azure Cosmos DB resource.
- AzureCosmosDBEmulatorResource: Represents an Azure Cosmos DB emulator resource.

To access these types and APIs for expressing them, add the 📦 Aspire.Hosting.Azure.CosmosDB NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.CosmosDB
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Azure Cosmos DB resource

In your app host project, call AddAzureCosmosDB to add and return an Azure Cosmos DB resource builder.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db");

// After adding all resources, run the app...
```

When you add an AzureCosmosDBResource to the app host, it exposes other useful APIs to add databases and containers. In other words, you must add an `AzureCosmosDBResource` before adding any of the other Cosmos DB resources.

> **ⓘ Important**
>
> When you call AddAzureCosmosDB, it implicitly calls AddAzureProvisioning—which adds support for generating Azure resources dynamically during app startup. The app must configure the appropriate subscription and location. For more information, see Local provisioning: Configuration.

Generated provisioning Bicep

If you're new to Bicep, it's a domain-specific language for defining Azure resources. With .NET Aspire, you don't need to write Bicep by-hand, instead the provisioning APIs generate Bicep for you. When you publish your app, the generated Bicep is output alongside the manifest file. When you add an Azure Cosmos DB resource, the following Bicep is generated:

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location

param principalType string

param principalId string

resource cosmos 'Microsoft.DocumentDB/databaseAccounts@2024-08-15' = {
  name: take('cosmos-${uniqueString(resourceGroup().id)}', 44)
  location: location
  properties: {
    locations: [
      {
        locationName: location
        failoverPriority: 0
      }
    ]
    consistencyPolicy: {
```

```
      defaultConsistencyLevel: 'Session'
    }
    databaseAccountOfferType: 'Standard'
    disableLocalAuth: true
  }
  kind: 'GlobalDocumentDB'
  tags: {
    'aspire-resource-name': 'cosmos'
  }
}

resource cosmos_roleDefinition
'Microsoft.DocumentDB/databaseAccounts/sqlRoleDefinitions@2024-08-15'
existing = {
  name: '00000000-0000-0000-0000-000000000002'
  parent: cosmos
}

resource cosmos_roleAssignment
'Microsoft.DocumentDB/databaseAccounts/sqlRoleAssignments@2024-08-15' = {
  name: guid(principalId, cosmos_roleDefinition.id, cosmos.id)
  properties: {
    principalId: principalId
    roleDefinitionId: cosmos_roleDefinition.id
    scope: cosmos.id
  }
  parent: cosmos
}

output connectionString string = cosmos.properties.documentEndpoint
```

The preceding Bicep is a module that provisions an Azure Cosmos DB account with the following defaults:

- `kind`: The kind of Cosmos DB account. The default is `GlobalDocumentDB`.
- `consistencyPolicy`: The consistency policy of the Cosmos DB account. The default is `Session`.
- `locations`: The locations for the Cosmos DB account. The default is the resource group's location.

In addition to the Cosmos DB account, it also adds the current application to the `Data Contributor` role for the Cosmos DB account. The generated Bicep is a starting point and is influenced by changes to the provisioning infrastructure in C#. Customizations to the Bicep file directly will be overwritten, so make changes through the C# provisioning APIs to ensure they are reflected in the generated files.

Customize provisioning infrastructure

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This type enables the customization of the generated Bicep by providing a fluent API to configure the Azure resources—using the ConfigureInfrastructure<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API. For example, you can configure the `kind`, `consistencyPolicy`, `locations`, and more. The following example demonstrates how to customize the Azure Cosmos DB resource:

C#

```
builder.AddAzureCosmosDB("cosmos-db")
    .ConfigureInfrastructure(infra =>
    {
        var cosmosDbAccount = infra.GetProvisionableResources()
                                   .OfType<CosmosDBAccount>()
                                   .Single();

        cosmosDbAccount.Kind = CosmosDBAccountKind.MongoDB;
        cosmosDbAccount.ConsistencyPolicy = new()
        {
            DefaultConsistencyLevel = DefaultConsistencyLevel.Strong,
        };
        cosmosDbAccount.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
  - The `infra` parameter is an instance of the AzureResourceInfrastructure type.
  - The provisionable resources are retrieved by calling the GetProvisionableResources() method.
  - The single CosmosDBAccount is retrieved.
  - The CosmosDBAccount.ConsistencyPolicy is assigned to a DefaultConsistencyLevel.Strong.
  - A tag is added to the Cosmos DB account with a key of `ExampleKey` and a value of `Example value`.

There are many more configuration options available to customize the Azure Cosmos DB resource. For more information, see Azure.Provisioning.CosmosDB. For more information, see Azure.Provisioning customization.

Connect to an existing Azure Cosmos DB account

You might have an existing Azure Cosmos DB account that you want to connect to. Instead of representing a new Azure Cosmos DB resource, you can add a connection

string to the app host. To add a connection to an existing Azure Cosmos DB account, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddConnectionString("cosmos-db");

builder.AddProject<Projects.WebApplication>("web")
       .WithReference(cosmos);

// After adding all resources, run the app...
```

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section. The app host injects this connection string as an environment variable into all dependent resources, for example:

JSON

```
{
    "ConnectionStrings": {
        "cosmos-db":
"AccountEndpoint=https://{account_name}.documents.azure.com:443/;AccountKey=
{account_key};"
    }
}
```

The dependent resource can access the injected connection string by calling the GetConnectionString method, and passing the connection name as the parameter, in this case `"cosmos-db"`. The `GetConnectionString` API is shorthand for `IConfiguration.GetSection("ConnectionStrings")[name]`.

Add Azure Cosmos DB database and container resources

To add an Azure Cosmos DB database resource, call the AddCosmosDatabase method on an `IResourceBuilder<AzureCosmosDBResource>` instance:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db");
cosmos.AddCosmosDatabase("db");

// After adding all resources, run the app...
```

When you call `AddCosmosDatabase`, it adds a database named `db` to your Cosmos DB resources and returns the newly created database resource. The database is created in the Cosmos DB account that's represented by the `AzureCosmosDBResource` that you added earlier. The database is a logical container for collections and users.

An Azure Cosmos DB container is where data is stored. When you create a container, you need to supply a partition key.

To add an Azure Cosmos DB container resource, call the AddContainer method on an `IResourceBuilder<AzureCosmosDBDatabaseResource>` instance:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db");
var db = cosmos.AddCosmosDatabase("db");
db.AddContainer("entries", "/id");

// After adding all resources, run the app...
```

The container is created in the database that's represented by the `AzureCosmosDBDatabaseResource` that you added earlier.

For more information, see Databases, containers, and items in Azure Cosmos DB.

Add Azure Cosmos DB emulator resource

To add an Azure Cosmos DB emulator resource, chain a call on an `IResourceBuilder<AzureCosmosDBResource>` to the RunAsEmulator API:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db")
                    .RunAsEmulator();
```

```
// After adding all resources, run the app...
```

When you call `RunAsEmulator`, it configures your Cosmos DB resources to run locally using an emulator. The emulator in this case is the Azure Cosmos DB Emulator. The Azure Cosmos DB Emulator provides a free local environment for testing your Azure Cosmos DB apps and it's a perfect companion to the .NET Aspire Azure hosting integration. The emulator isn't installed, instead, it's accessible to .NET Aspire as a container. When you add a container to the app host, as shown in the preceding example with the `mcr.microsoft.com/cosmosdb/emulator` image, it creates and starts the container when the app host starts. For more information, see Container resource lifecycle.

Configure Cosmos DB emulator container

There are various configurations available to container resources, for example, you can configure the container's ports, environment variables, it's lifetime, and more.

Configure Cosmos DB emulator container gateway port

By default, the Cosmos DB emulator container when configured by .NET Aspire, exposes the following endpoints:

Expand table

Endpoint	Container port	Host port
`https`	8081	dynamic

The port that it's listening on is dynamic by default. When the container starts, the port is mapped to a random port on the host machine. To configure the endpoint port, chain calls on the container resource builder provided by the `RunAsEmulator` method as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db").RunAsEmulator(
                     emulator =>
                     {
                         emulator.WithGatewayPort(7777);
                     });
```

```
// After adding all resources, run the app...
```

The preceding code configures the Cosmos DB emulator container's existing `https` endpoint to listen on port `8081`. The Cosmos DB emulator container's port is mapped to the host port as shown in the following table:

Expand table

Endpoint name	Port mapping (`container:host`)
`https`	`8081:7777`

Configure Cosmos DB emulator container with persistent lifetime

To configure the Cosmos DB emulator container with a persistent lifetime, call the WithLifetime method on the Cosmos DB emulator container resource and pass ContainerLifetime.Persistent:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db").RunAsEmulator(
                     emulator =>
                     {

emulator.WithLifetime(ContainerLifetime.Persistent);
                     });

// After adding all resources, run the app...
```

For more information, see Container resource lifetime.

Configure Cosmos DB emulator container with data volume

To add a data volume to the Azure Cosmos DB emulator resource, call the WithDataVolume method on the Azure Cosmos DB emulator resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db").RunAsEmulator(
                     emulator =>
                     {
```

```
                        emulator.WithDataVolume();
                    });

// After adding all resources, run the app...
```

The data volume is used to persist the Cosmos DB emulator data outside the lifecycle of its container. The data volume is mounted at the `/tmp/cosmos/appdata` path in the Cosmos DB emulator container and when a `name` parameter isn't provided, the name is generated. The emulator has its `AZURE_COSMOS_EMULATOR_ENABLE_DATA_PERSISTENCE` environment variable set to `true`. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Configure Cosmos DB emulator container partition count

To configure the partition count of the Cosmos DB emulator container, call the WithPartitionCount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db").RunAsEmulator(
                    emulator =>
                    {
                        emulator.WithPartitionCount(100); // Defaults to 25
                    });

// After adding all resources, run the app...
```

The preceding code configures the Cosmos DB emulator container to have a partition count of `100`. This is a shorthand for setting the `AZURE_COSMOS_EMULATOR_PARTITION_COUNT` environment variable.

Use Linux-based emulator (preview)

The next generation of the Azure Cosmos DB emulator is entirely Linux-based and is available as a Docker container. It supports running on a wide variety of processors and operating systems.

To use the preview version of the Cosmos DB emulator, call the RunAsPreviewEmulator method. Since this feature is in preview, you need to explicitly opt into the preview feature by suppressing the `ASPIRECOSMOSDB001` experimental diagnostic.

The preview emulator also supports exposing a "Data Explorer" endpoint which allows you to view the data stored in the Cosmos DB emulator via a web UI. To enable the Data Explorer, call the WithDataExplorer method.

C#

```
#pragma warning disable ASPIRECOSMOSDB001

var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db").RunAsPreviewEmulator(
                     emulator =>
                     {
                         emulator.WithDataExplorer();
                     });

// After adding all resources, run the app...
```

The preceding code configures the Linux-based preview Cosmos DB emulator container, with the Data Explorer endpoint, to use at run time.

Hosting integration health checks

The Azure Cosmos DB hosting integration automatically adds a health check for the Cosmos DB resource. The health check verifies that the Cosmos DB is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.CosmosDb NuGet package.

Client integration

To get started with the .NET Aspire Azure Cosmos DB client integration, install the Aspire.Microsoft.Azure.Cosmos NuGet package in the client-consuming project, that is, the project for the application that uses the Cosmos DB client. The Cosmos DB client integration registers a CosmosClient instance that you can use to interact with Cosmos DB.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Microsoft.Azure.Cosmos
```

Add Cosmos DB client

In the Program.cs file of your client-consuming project, call the AddAzureCosmosClient extension method on any IHostApplicationBuilder to register a CosmosClient for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddAzureCosmosClient(connectionName: "cosmos-db");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Cosmos DB resource in the app host project. In other words, when you call `AddAzureCosmosDB` and provide a name of `cosmos-db` that same name should be used when calling `AddAzureCosmosClient`. For more information, see **Add Azure Cosmos DB resource**.

You can then retrieve the CosmosClient instance using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(CosmosClient client)
{
    // Use client...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed Cosmos DB client

There might be situations where you want to register multiple `CosmosClient` instances with different connection names. To register keyed Cosmos DB clients, call the AddKeyedAzureCosmosClient method:

C#

```
builder.AddKeyedAzureCosmosClient(name: "mainDb");
builder.AddKeyedAzureCosmosClient(name: "loggingDb");
```

> **ⓘ Important**
>
> When using keyed services, it's expected that your Cosmos DB resource configured two named databases, one for the `mainDb` and one for the `loggingDb`.

Then you can retrieve the `CosmosClient` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("mainDb")] CosmosClient mainDbClient,
    [FromKeyedServices("loggingDb")] CosmosClient loggingDbClient)
{
    // Use clients...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire Azure Cosmos DB integration provides multiple options to configure the connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling the AddAzureCosmosClient method:

C#

```
builder.AddAzureCosmosClient("cosmos-db");
```

Then the connection string is retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "cosmos-db":
"AccountEndpoint=https://{account_name}.documents.azure.com:443/;AccountKey=
```

```
{account_key};"
  }
}
```

For more information on how to format this connection string, see the ConnectionString documentation.

Use configuration providers

The .NET Aspire Azure Cosmos DB integration supports Microsoft.Extensions.Configuration. It loads the MicrosoftAzureCosmosSettings from configuration by using the `Aspire:Microsoft:Azure:Cosmos` key. The following snippet is an example of a appsettings.json file that configures some of the options:

JSON

```
{
  "Aspire": {
    "Microsoft": {
      "Azure": {
        "Cosmos": {
          "DisableTracing": false,
        }
      }
    }
  }
}
```

For the complete Cosmos DB client integration JSON schema, see Aspire.Microsoft.Azure.Cosmos/ConfigurationSchema.json.

Use inline delegates

Also you can pass the `Action<MicrosoftAzureCosmosSettings> configureSettings` delegate to set up some or all the options inline, for example to disable tracing from code:

C#

```
builder.AddAzureCosmosClient(
    "cosmos-db",
    static settings => settings.DisableTracing = true);
```

You can also set up the Microsoft.Azure.Cosmos.CosmosClientOptions using the optional `Action<CosmosClientOptions> configureClientOptions` parameter of the `AddAzureCosmosClient` method. For example to set the CosmosClientOptions.ApplicationName user-agent header suffix for all requests issues by this client:

```
C#
```

The .NET Aspire Cosmos DB client integration currently doesn't implement health checks, though this may change in future releases.

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

The .NET Aspire Azure Cosmos DB integration uses the following log categories:

-

```
```

```
    configureClientOptions:
        clientOptions => {
            clientOptions.CosmosClientTelemetryOptions = new()
            {
                CosmosThresholdOptions = new()
                {
                    PointOperationLatencyThreshold =
TimeSpan.FromMilliseconds(50),
                    NonPointOperationLatencyThreshold =
TimeSpan.FromMilliseconds(300)
                }
            };
        });
```

Tracing

The .NET Aspire Azure Cosmos DB integration will emit the following tracing activities using OpenTelemetry:

- `Azure.Cosmos.Operation`

Azure Cosmos DB tracing is currently in preview, so you must set the experimental switch to ensure traces are emitted.

C#

```
AppContext.SetSwitch("Azure.Experimental.EnableActivitySource", true);
```

For more information, see Azure Cosmos DB SDK observability: Trace attributes.

Metrics

The .NET Aspire Azure Cosmos DB integration currently doesn't support metrics by default due to limitations with the Azure SDK.

See also

- Azure Cosmos DB
- .NET Aspire Cosmos DB Entity Framework Core integration
- .NET Aspire integrations overview
- .NET Aspire Azure integrations overview
- .NET Aspire GitHub repo

.NET Aspire Azure Event Hubs integration

Article • 03/18/2025

Includes: ✅ Hosting integration and ✅ Client integration

Azure Event Hubs is a native data-streaming service in the cloud that can stream millions of events per second, with low latency, from any source to any destination. The .NET Aspire Azure Event Hubs integration enables you to connect to Azure Event Hubs instances from your .NET applications.

Hosting integration

The .NET Aspire Azure Event Hubs hosting integration models the various Event Hub resources as the following types:

- AzureEventHubsResource: Represents a top-level Azure Event Hubs resource, used for representing collections of hubs and the connection information to the underlying Azure resource.
- AzureEventHubResource: Represents a single Event Hub resource.
- AzureEventHubsEmulatorResource: Represents an Azure Event Hubs emulator as a container resource.
- AzureEventHubConsumerGroupResource: Represents a consumer group within an Event Hub resource.

To access these types and APIs for expressing them within your app host project, install the Aspire.Hosting.Azure.EventHubs NuGet package:

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.EventHubs
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add an Azure Event Hubs resource

To add an AzureEventHubsResource to your app host project, call the AddAzureEventHubs method providing a name, and then call AddHub:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var eventHubs = builder.AddAzureEventHubs("event-hubs");
eventHubs.AddHub("messages");

builder.AddProject<Projects.ExampleService>()
       .WithReference(eventHubs);

// After adding all resources, run the app...
```

When you add an Azure Event Hubs resource to the app host, it exposes other useful APIs to add Event Hub resources, consumer groups, express explicit provisioning configuration, and enables the use of the Azure Event Hubs emulator. The preceding code adds an Azure Event Hubs resource named `event-hubs` and an Event

Hub named `messages` to the app host project. The WithReference method passes the connection information to the `ExampleService` project.

> **Important**
>
> When you call AddAzureEventHubs, it implicitly calls

If you're new to Bicep, it's a domain-specific language for defining Azure resources. With .NET Aspire, you don't need to write Bicep by-hand, instead the provisioning APIs generate Bicep for you. When you publish your app, the generated Bicep is output alongside the manifest file. When you add an Azure Event Hubs resource, the following Bicep is generated:

Bicep

The

messages : The Event

oning infrastructure

AzureProvisioningResource type. This type enables the roviding a fluent API to configure the Azure resources—using the

Con ureInfrastructure< ler<T>, Action<AzureResourceInfrastructure>) API. For example, you

ind locations , and more. The following example demonstrates how to

```
eEventHubs("event-hubs") {

ventHubs = infra.GetProvi ingle();
ventHubs.Sku =
    new EventHubsSku{ Name = Even bsSkuName.Premium,

    "ExampleKey", "Example value");
```

- Chains a call t onfigu tructure A.
  - The `infra` pa is an i of the Azu
  - The provisionab es are ed by calli
  - The single re is retri
  - The Event
  - Name and tier and a `Capac 7`.
  - The PublicNetworkAccess property is assigned t redByPer
  - A tag is added to the Event Hubs resource with a key `ExampleKey` and a value of `Example value`.

erty is assigned the

There are many more configuration options available to customize the Event Hubs resource resource. For more information, see Azure.Provisioning.PostgreSql. For more information, see Azure.Provisioning customization.

Connect to an existing Azure Event Hubs namespace

You might have an existing Azure Event Hubs namespace that you want to connect to. Instead of representing a new Azure Event Hubs resource, you can add a connection string to the app host. To add a connection to an existing Azure Event Hubs namespace, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var eventHubs = builder.AddConnectionString("event-hubs");

builder.AddProject<Projects.WebApplication>("web")
       .WithReference(eventHubs);

// After adding all resources, run the app...
```

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section. The app host injects this connection string as an environment variable into all dependent resources, for example:

JSON

```
{
  "ConnectionStrings": {
    "event-hubs": "{your_namespace}.servicebus.windows.net"
  }
}
```

The dependent resource can access the injected connection string by calling the GetConnectionString method, and passing the connection name as the parameter, in this case `"event-hubs"`. The `GetConnectionString` API is shorthand for `IConfiguration.GetSection("ConnectionStrings")[name]`.

Add Event Hub consumer group

To add a consumer group, chain a call on an `IResourceBuilder<AzureEventHubsResource>` to the AddConsumerGroup API:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var eventHubs = builder.AddAzureEventHubs("event-hubs");
var messages = eventHubs.AddHub("messages");
messages.AddConsumerGroup("messagesConsumer");
```

When you call `AddConsumerGroup`, it configures your `messages` Event Hub resource to have a consumer group named `messagesConsumer`. The consumer group is created in the Azure Event Hubs namespace that's represented by the `AzureEventHubsResource` that you added earlier. For more information, see Azure Event Hubs: Consumer groups.

The .NET Aspire Azure Event Hubs hosting integration supports running the Event Hubs resource as an emulator locally, based on the `mcr.microsoft.com/azure-messaging/eventhubs-emulator/latest` container image. This is beneficial for situations where you want to run the Event Hubs resource locally for development and testing purposes, avoiding the need to provision an Azure resource or connect to an existing Azure Event Hubs server.

To run the Event Hubs resource as an emulator, call the RunAsEmulator method:

C#

The preceding code configures an Azure Event Hubs resource to run locally in a container. For more information, see Azure Event Hubs Emulator.

There are various configurations available for container resources, for example, you can configure the container's ports, data bind mounts, data volumes, or providing a wholistic JSON configuration which overrides everything.

By default, the Event Hubs emulator container when configured by .NET Aspire, exposes the following endpoints:

Endpoint	Image	Container port	Host port

The port that it's listening on is dynamic by default. When the container starts, the port is mapped to a random port on the host machine. To configure the endpoint port, chain calls on the container resource builder provided by the `RunAsEmulator` method and then the WithHostPort(IResourceBuilder<AzureEventHubsEmulatorResource>, Nullable<Int32>) as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var eventHubs = builder.AddAzureEventHubs("event-hubs")
                       .RunAsEmulator(emulator =>
                       {
                           emulator.WithHostPort(7777);
                       });

eventHubs.AddHub("messages");

builder.AddProject<Projects.ExampleService>()
       .WithReference(eventHubs);

// After adding all resources, run the app...
```

The preceding code configures the Azure Event emulator container's existing `emulator` endpoint to listen on port `7777`. The Azure Event emulator container's port is mapped to the host port as shown in the following table:

Expand table

Endpoint name	Port mapping (`container:host`)
`emulator`	`5672:7777`

Add Event Hubs emulator with data volume

To add a data volume to the Event Hubs emulator resource, call the WithDataVolume method on the Event Hubs emulator resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var eventHubs = builder.AddAzureEventHubs("event-hubs")
                       .RunAsEmulator(emulator =>
                       {
                           emulator.WithDataVolume();
                       });

eventHubs.AddHub("messages");

builder.AddProject<Projects.ExampleService>()
       .WithReference(eventHubs);

// After adding all resources, run the app...
```

The data volume is used to persist the Event Hubs emulator data outside the lifecycle of its container. The data volume is mounted at the `/data` path in the container. A name is generated at random unless you provide a

set the `name` parameter. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add Event Hubs emulator with data bind mount

The add a bind mount to the Event Hubs emulator container, chain a call to the WithDataBindMount API, as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var eventHubs = builder.AddAzureEventHubs("event-hubs")
                       .RunAsEmulator(emulator =>
                       {
                           emulator.WithDataBindMount("/path/to/data");
                       });

eventHubs.AddHub("messages");

builder.AddProject<Projects.ExampleService>()
       .WithReference(eventHubs);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Azure Event Hubs emulator resource data across container restarts. The data bind mount is mounted at the `/path/to/data` path on the host machine in the container. For more information on data bind mounts, see Docker docs: Bind mounts.

Configure Event Hubs emulator container JSON configuration

The Event Hubs emulator container runs with a default *config.json* file. You can override this file entirely, or update the JSON configuration with a JsonNode representation of the configuration.

To provide a custom JSON configuration file, call the WithConfigurationFile(IResourceBuilder<AzureEventHubsEmulatorResource>, String) method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var eventHubs = builder.AddAzureEventHubs("event-hubs")
                       .RunAsEmulator(emulator =>
                       {
                           emulator.WithConfigurationFile("./messaging/custom-config.json");
                       });

eventHubs.AddHub("messages");
```

```
builder.AddProject<Projects.ExampleService>()
       .WithReference(eventHubs);

// After adding all resources, run the app...
```

The preceding code configures the Event Hubs emulator container to use a custom JSON configuration file located at `./messaging/custom-config.json`. This will be mounted at the `/Eventhubs_Emulator/ConfigFiles/Config.json` path on the container, as a read-only file. To instead override specific properties in the default configuration, call the WithConfiguration(IResourceBuilder<AzureEventHubsEmulatorResource>, Action<JsonNode>) method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var eventHubs = builder.AddAzureEventHubs("event-hubs")
                       .RunAsEmulator(emulator =>
                       {
                           emulator.WithConfiguration(
                               (JsonNode configuration) =>
                               {
                                   var userConfig = configuration["UserConfig"];
                                   var ns = userConfig["NamespaceConfig"][0];
                                   var firstEntity = ns["Entities"][0];

                                   firstEntity["PartitionCount"] = 5;
                               });
                       });

eventHubs.AddHub("messages");

builder.AddProject<Projects.ExampleService>()
       .WithReference(eventHubs);

// After adding all resources, run the app...
```

The preceding code retrieves the `UserConfig` node from the default configuration. It then updates the first entity's `PartitionCount` to a `5`.

Hosting integration health checks

The Azure Event Hubs hosting integration automatically adds a health check for the Event Hubs resource. The health check verifies that the Event Hubs is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Azure.Messaging.EventHubs NuGet package.

Client integration

To get started with the .NET Aspire Azure Event Hubs client integration, install the Aspire.Azure.Messaging.EventHubs NuGet package in the client-consuming project, that is, the project for the application that uses the Event Hubs client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Azure.Messaging.EventHubs
```

Supported Event Hubs client types

The following Event Hub clients are supported by the library, along with their corresponding options and settings classes:

Expand table

Azure client type	Azure options class	.NET Aspire settings class
EventHubProducerClient	EventHubProducerClientOptions	AzureMessagingEventHubsProducerSettings
EventHubBufferedProducerClient	EventHubBufferedProducerClientOptions	AzureMessagingEventHubsBufferedProducerSettings
EventHubConsumerClient	EventHubConsumerClientOptions	AzureMessagingEventHubsConsumerSettings
EventProcessorClient	EventProcessorClientOptions	AzureMessagingEventHubsProcessorSettings
PartitionReceiver	PartitionReceiverOptions	AzureMessagingEventHubsPartitionReceiverSettings

The client types are from the Azure SDK for .NET, as are the corresponding options classes. The settings classes are provided by the .NET Aspire. The settings classes are used to configure the client instances.

Add an Event Hubs producer client

In the *Program.cs* file of your client-consuming project, call the AddAzureEventHubProducerClient extension method on any IHostApplicationBuilder to register an EventHubProducerClient for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddAzureEventHubProducerClient(connectionName: "event-hubs");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Event Hubs resource in the app host project. For more information, see **Add an Azure Event Hubs resource**.

After adding the `EventHubProducerClient`, you can retrieve the client instance using dependency injection. For example, to retrieve your data source object from an example service define it as a constructor parameter and ensure the `ExampleService` class is registered with the dependency injection container:

C#

```
public class ExampleService(EventHubProducerClient client)
{
    // Use client...
}
```

For more information, see:

- Azure.Messaging.EventHubs documentation for examples on using the `EventHubProducerClient`.
- Dependency injection in .NET for details on dependency injection.

Additional APIs to consider

The client integration provides additional APIs to configure client instances. When you need to register an Event Hubs client, consider the following APIs:

Azure client type	Registration API
EventHubProducerClient	AddAzureEventHubProducerClient
EventHubBufferedProducerClient	AddAzureEventHubBufferedProducerClient
EventHubConsumerClient	AddAzureEventHubConsumerClient
EventProcessorClient	AddAzureEventProcessorClient
PartitionReceiver	AddAzurePartitionReceiverClient

All of the aforementioned APIs include optional parameters to configure the client instances.

There might be situations where you want to register multiple `EventHubProducerClient` instances with different connection names. To register keyed Event Hubs clients, call the AddKeyedAzureServiceBusClient method:

C#

Then you can retrieve the client instances using dependency injection. For example, to retrieve the clients from a service:

C#

For more information, see Keyed services in .NET.

Additional keyed APIs to consider

The client integration provides additional APIs to configure keyed client instances. When you need to register a keyed Event Hubs client, consider the following APIs:

Expand table

Azure client type	Registration API
EventHubProducerClient	AddKeyedAzureEventHubProducerClient
EventHubBufferedProducerClient	AddKeyedAzureEventHubBufferedProducerClient
EventHubConsumerClient	AddKeyedAzureEventHubConsumerClient
EventProcessorClient	AddKeyedAzureEventProcessorClient
PartitionReceiver	AddKeyedAzurePartitionReceiverClient

All of the aforementioned APIs include optional parameters to configure the client instances.

Configuration

The .NET Aspire Azure Event Hubs library provides multiple options to configure the Azure Event Hubs connection based on the requirements and conventions of your project. Either a `FullyQualifiedNamespace` or a `ConnectionString` is a required to be supplied.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, provide the name of the connection string when calling `builder.AddAzureEventHubProducerClient()` and other supported Event Hubs clients. In this example, the connection string does not include the `EntityPath` property, so the `EventHubName` property must be set in the settings callback:

C#

```
builder.AddAzureEventHubProducerClient(
    "event-hubs",
    static settings =>
    {
        settings.EventHubName = "MyHub";
    });
```

And then the connection information will be retrieved from the `ConnectionStrings` configuration section. Two connection formats are supported:

Fully Qualified Namespace (FQN)

The recommended approach is to use a fully qualified namespace, which works with the AzureMessagingEventHubsSettings.Credential property to establish a connection. If no credential is configured, the DefaultAzureCredential is used.

JSON

```
{
  "ConnectionStrings": {
    "event-hubs": "{your_namespace}.servicebus.windows.net"
  }
}
```

Connection string

Alternatively, use a connection string:

JSON

```
{
  "ConnectionStrings": {
    "event-hubs":
"Endpoint=sb://mynamespace.servicebus.windows.net/;SharedAccessKeyName=accesskeyname;SharedAccess
Key=accesskey;EntityPath=MyHub"
  }
}
```

Use configuration providers

The .NET Aspire Azure Event Hubs library supports Microsoft.Extensions.Configuration. It loads the `AzureMessagingEventHubsSettings` and the associated Options, e.g. `EventProcessorClientOptions`, from configuration by using the `Aspire:Azure:Messaging:EventHubs:` key prefix, followed by the name of the specific client in use. For example, consider the *appsettings.json* that configures some of the options for an `EventProcessorClient`:

JSON

```
{
  "Aspire": {
    "Azure": {
      "Messaging": {
        "EventHubs": {
          "EventProcessorClient": {
            "EventHubName": "MyHub",
            "ClientOptions": {
              "Identifier": "PROCESSOR_ID"
            }
          }
        }
      }
    }
  }
}
```

For the complete Azure Event Hubs client integration JSON schema, see Aspire.Azure.Messaging.EventHubs/ConfigurationSchema.json.

You can also setup the Options type using the optional `Action<IAzureClientBuilder<EventProcessorClient, EventProcessorClientOptions>> configureClientBuilder` parameter of the `AddAzureEventProcessorClient` method. For example, to set the processor's client ID for this client:

C#

```
builder.AddAzureEventProcessorClient(
    "event-hubs",
    configureClientBuilder: clientBuilder => clientBuilder.ConfigureOptions(
        options => options.Identifier = "PROCESSOR_ID"));
```

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Azure Event Hubs integration uses the following log categories:

- `Azure.Core`
- `Azure.Identity`

Tracing

The .NET Aspire Azure Event Hubs integration will emit the following tracing activities using OpenTelemetry:

- `Azure.Messaging.EventHubs.*`

Metrics

The .NET Aspire Azure Event Hubs integration currently doesn't support metrics by default due to limitations with the Azure SDK for .NET. If that changes in the future, this section will be updated to reflect those changes.

See also

- Azure Event Hubs
- .NET Aspire Azure integrations overview
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire Azure Functions integration (Preview)

Article • 11/14/2024

Includes: Hosting integration not Client integration

> **Important**
>
> The .NET Aspire Azure Functions integration is currently in preview and is subject to change.

Azure Functions is a serverless solution that allows you to write less code, maintain less infrastructure, and save on costs. The .NET Aspire Azure Functions integration enables you to develop, debug, and orchestrate an Azure Functions .NET project as part of the app host.

It's expected that you've installed the required Azure tooling:

- Configure Visual Studio for Azure development with .NET

Supported scenarios

The .NET Aspire Azure Functions integration has several key supported scenarios. This section outlines the scenarios and provides details related to the implementation of each approach.

Supported triggers

The following table lists the supported triggers for Azure Functions in the .NET Aspire integration:

Expand table

Trigger	Attribute	Details
Azure Event Hubs trigger	`EventHubTrigger`	Aspire.Hosting.Azure.EventHubs
Azure Service Bus trigger	`ServiceBusTrigger`	Aspire.Hosting.Azure.ServiceBus

Trigger	Attribute	Details
Azure Storage Blobs trigger	`BlobTrigger`	Aspire.Hosting.Azure.Storage
Azure Storage Queues trigger	`QueueTrigger`	Aspire.Hosting.Azure.Storage
Azure CosmosDB trigger	`CosmosDbTrigger`	Aspire.Hosting.Azure.CosmosDB
HTTP trigger	`HttpTrigger`	Supported without any additional resource dependencies.
Timer trigger	`TimerTrigger`	Supported without any additional resource dependencies—relies on implicit host storage.

> **Important**
>
> Other Azure Functions triggers and bindings aren't currently supported in the .NET Aspire Azure Functions integration.

Deployment

Currently, deployment is supported only to containers on Azure Container Apps (ACA) using the SDK container publish function in `Microsoft.Azure.Functions.Worker.Sdk`. This deployment methodology doesn't currently support KEDA-based autoscaling.

Configure external HTTP endpoints

To make HTTP triggers publicly accessible, call the WithExternalHttpEndpoints API on the AzureFunctionsProjectResource. For more information, see Add Azure Functions resource.

Azure Function project constraints

The .NET Aspire Azure Functions integration has the following project constraints:

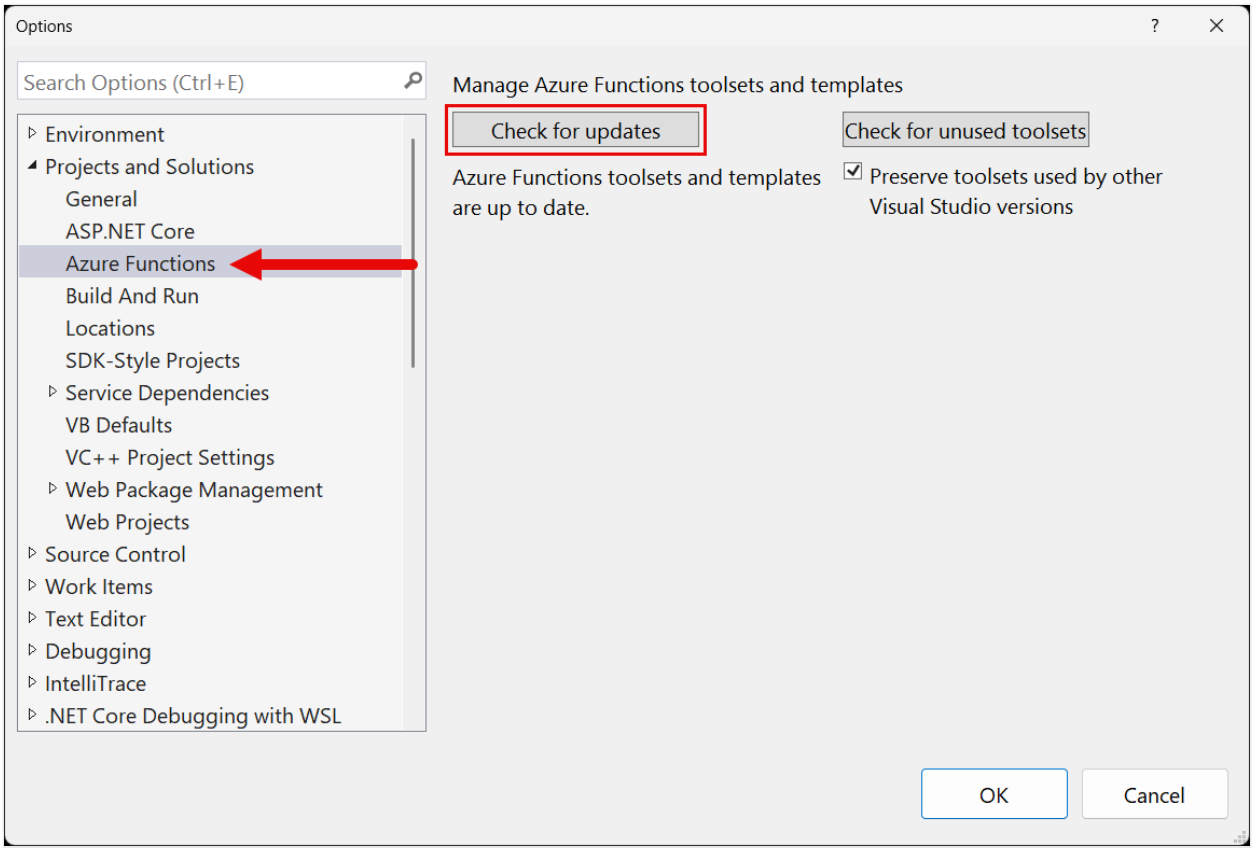
- You must target .NET 8.0 or later.
- You must use a .NET 9 SDK.
- It currently only supports .NET workers with the isolated worker model.
- Requires the following NuGet packages:
  - Microsoft.Azure.Functions.Worker: Use the `FunctionsApplicationBuilder`.

- Microsoft.Azure.Functions.Worker.Sdk: Adds support for `dotnet run` and `azd publish`.
- Microsoft.Azure.Functions.Http.AspNetCore: Adds HTTP trigger-supporting APIs.

If you encounter issues with the Azure Functions project, such as:

> There is no Functions runtime available that matches the version specified in the project

In Visual Studio, try checking for an update on the Azure Functions tooling. Open the **Options** dialog, navigate to **Projects and Solutions**, and then select **Azure Functions**. Select the **Check for updates** button to ensure you have the latest version of the Azure Functions tooling:



Hosting integration

The Azure Functions hosting integration models an Azure Functions resource as the AzureFunctionsProjectResource (subtype of ProjectResource) type. To access this type and APIs that allow you to add it to your app host project install the Aspire.Hosting.Azure.Functions NuGet package.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Functions --prerelease
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Azure Functions resource

In your app host project, call AddAzureFunctionsProject on the `builder` instance to add an Azure Functions resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var functions = builder.AddAzureFunctionsProject<Projects.ExampleFunctions>
("functions")
                       .WithExternalHttpEndpoints();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(functions)
       .WaitFor(functions);

// After adding all resources, run the app...
```

When .NET Aspire adds an Azure Functions project resource the app host, as shown in the preceding example, the `functions` resource can be referenced by other project resources. The WithReference method configures a connection in the `ExampleProject` named `"functions"`. If the Azure Resource was deployed and it exposed an HTTP trigger, its endpoint would be external due to the call to WithExternalHttpEndpoints. For more information, see Reference resources.

Add Azure Functions resource with host storage

If you want to modify the default host storage account that the Azure Functions host uses, call the WithHostStorage method on the Azure Functions project resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage")
```

```
                    .RunAsEmulator();

var functions = builder.AddAzureFunctionsProject<Projects.ExampleFunctions>
("functions")
                      .WithHostStorage(storage);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(functions)
       .WaitFor(functions);

// After adding all resources, run the app...
```

The preceding code relies on the Aspire.Hosting.Azure.Storage NuGet package to add an Azure Storage resource that runs as an emulator. The `storage` resource is then passed to the `WithHostStorage` API, explicitly setting the host storage to the emulated resource.

> **Note**
>
> If you're not using the implicit host storage, you must manually assign the `StorageAccountContributor` role to your resource for deployed instances. This role is automatically assigned for the implicitly generated host storage.

Reference resources in Azure Functions

To reference other Azure resources in an Azure Functions project, chain a call to `WithReference` on the Azure Functions project resource and provide the resource to reference:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage").RunAsEmulator();
var blobs = storage.AddBlobs("blobs");

builder.AddAzureFunctionsProject<Projects.ExampleFunctions>("functions")
       .WithHostStorage(storage)
       .WithReference(blobs);

builder.Build().Run();
```

The preceding code adds an Azure Storage resource to the app host and references it in the Azure Functions project. The `blobs` resource is added to the `storage` resource and then referenced by the `functions` resource. The connection information required to

connect to the `blobs` resource is automatically injected into the Azure Functions project and enables the project to define a `BlobTrigger` that relies on `blobs` resource.

See also

- .NET Aspire integrations
- .NET Aspire GitHub repo
- Azure Functions documentation
- .NET Aspire and Functions image gallery sample

.NET Aspire Azure Key Vault integration

Article • 02/15/2025

Includes: ✅ Hosting integration and ✅ Client integration

Azure Key Vault is a cloud service for securely storing and accessing secrets. The .NET Aspire Azure Key Vault integration enables you to connect to Azure Key Vault instances from your .NET applications.

Hosting integration

The Azure Key Vault hosting integration models a Key Vault resource as the AzureKeyVaultResource type. To access this type and APIs for expressing them within your app host project, install the Aspire.Hosting.Azure.KeyVault NuGet package:

.NET CLI

```
.NET CLI
dotnet add package Aspire.Hosting.Azure.KeyVault
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Azure Key Vault resource

In your app host project, call AddAzureKeyVault on the builder instance to add an Azure Key Vault resource:

```
C#
var builder = DistributedApplication.CreateBuilder(args);

var keyVault = builder.AddAzureKeyVault("key-vault");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(keyVault);

// After adding all resources, run the app...
```

The WithReference method configures a connection in the `ExampleProject` named `"key-vault"`.

> **Important**
>
> By default, `AddAzureKeyVault` configures a Key Vault Administrator built-in role.

> **Tip**
>
> When you call AddAzureKeyVault, it implicitly calls AddAzureProvisioning, which adds support for generating Azure resources dynamically during app startup. The app must configure the appropriate subscription and location. For more information, see Local provisioning: Configuration.

Generated provisioning Bicep

If you're new to Bicep, it's a domain-specific language for defining Azure resources. With .NET Aspire, you don't need to write Bicep by-hand, instead the provisioning APIs generate Bicep for you. When you publish your app, the generated Bicep is output alongside the manifest file. When you add an Azure Key Vault resource, the following Bicep is generated:

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location

param principalType string

param principalId string

resource key_vault 'Microsoft.KeyVault/vaults@2023-07-01' = {
  name: take('keyvault-${uniqueString(resourceGroup().id)}', 24)
  location: location
  properties: {
    tenantId: tenant().tenantId
    sku: {
      family: 'A'
      name: 'standard'
    }
    enableRbacAuthorization: true
  }
  tags: {
    'aspire-resource-name': 'key-vault'
  }
```

```
}

resource key_vault_KeyVaultAdministrator
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(key_vault.id, principalId,
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '00482a5a-
887f-4fb3-b363-3b7fe8e74483'))
  properties: {
    principalId: principalId
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '00482a5a-
887f-4fb3-b363-3b7fe8e74483')
    principalType: principalType
  }
  scope: key_vault
}

output vaultUri string = key_vault.properties.vaultUri
```

The preceding Bicep is a module that provisions an Azure Key Vault resource with the following defaults:

- `location`: The location of the resource group.
- `principalId`: The principal ID of the user or service principal.
- `principalType`: The principal type of the user or service principal.
- `key_vault`: The Azure Key Vault resource:
  - `name`: A unique name for the Azure Key Vault.
  - `properties`: The Azure Key Vault properties:
    - `tenantId`: The tenant ID of the Azure Key Vault.
    - `sku`: The Azure Key Vault SKU:
      - `family`: The SKU family.
      - `name`: The SKU name.
    - `enableRbacAuthorization`: A boolean value that indicates whether the Azure Key Vault has role-based access control (RBAC) authorization enabled.
  - `tags`: The Azure Key Vault tags.
- `key_vault_KeyVaultAdministrator`: The Azure Key Vault administrator role assignment:
  - `name`: A unique name for the role assignment.
  - `properties`: The role assignment properties:
    - `principalId`: The principal ID of the user or service principal.
    - `roleDefinitionId`: The role definition ID of the Azure Key Vault administrator role.
    - `principalType`: The principal type of the user or service principal.
  - `scope`: The scope of the role assignment.

- `output`: The Azure Key Vault URI.

The generated Bicep is a starting point and is influenced by changes to the provisioning infrastructure in C#. Customizations to the Bicep file directly will be overwritten, so make changes through the C# provisioning APIs to ensure they are reflected in the generated files.

Customize provisioning infrastructure

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This type enables the customization of the generated Bicep by providing a fluent API to configure the Azure resources by using the ConfigureInfrastructure<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API. For example, you can configure the `sku`, `RBAC`, `tags`, and more. The following example demonstrates how to customize the Azure Key Vault resource:

C#

```
builder.AddAzureKeyVault("key-vault")
    .ConfigureInfrastructure(infra =>
    {
        var keyVault = infra.GetProvisionableResources()
                            .OfType<KeyVaultService>()
                            .Single();

        keyVault.Properties.Sku = new()
        {
            Family = KeyVaultSkuFamily.A,
            Name = KeyVaultSkuName.Premium,
        };
        keyVault.Properties.EnableRbacAuthorization = true;
        keyVault.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
  - The `infra` parameter is an instance of the AzureResourceInfrastructure type.
  - The provisionable resources are retrieved by calling the GetProvisionableResources() method.
  - The single KeyVaultService resource is retrieved.
  - The `Sku` property is set to a new KeyVaultSku instance.
  - The KeyVaultProperties.EnableRbacAuthorization property is set to `true`.
  - A tag is added to the resource with a key of `ExampleKey` and a value of `Example value`.

There are many more configuration options available to customize the Key Vault resource. For more information, see Azure.Provisioning.KeyVault and Azure.Provisioning customization.

Connect to an existing Azure Key Vault instance

You might have an existing Azure Key Vault instance that you want to connect to. Instead of representing a new Azure Key Vault resource, you can add a connection string to the app host. To add a connection to an existing Azure Key Vault resource, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var keyVault = builder.AddConnectionString("key-vault");

builder.AddProject<Projects.WebApplication>("web")
       .WithReference(keyVault);

// After adding all resources, run the app...
```

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section. The app host injects this connection string as an environment variable into all dependent resources, for example:

JSON

```
{
  "ConnectionStrings": {
    "key-vault": "https://{account_name}.vault.azure.net/"
  }
}
```

The dependent resource can access the injected connection string by calling the GetConnectionString method, and passing the connection name as the parameter, in

this case `"key-vault"`. The `GetConnectionString` API is shorthand for `IConfiguration.GetSection("ConnectionStrings")[name]`.

Client integration

To get started with the .NET Aspire Azure Key Vault client integration, install the 📦 Aspire.Azure.Security.KeyVault NuGet package in the client-consuming project, that is, the project for the application that uses the Azure Key Vault client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Azure.Security.KeyVault
```

The client integration provides two ways to access secrets from Azure Key Vault:

- Add secrets to app configuration, using either the `IConfiguration` or the `IOptions<T>` pattern.
- Use a `SecretClient` to retrieve secrets on demand.

Add secrets to configuration

In the Program.cs file of your client-consuming project, call the AddAzureKeyVaultSecrets extension method on the IConfiguration to add the secrets as part of your app's configuration. The method takes a connection name parameter.

C#

```
builder.Configuration.AddAzureKeyVaultSecrets(connectionName: "key-vault");
```

> **Note**
>
> The `AddAzureKeyVaultSecrets` API name has caused a bit of confusion. The method is used to configure the `SecretClient` based on the given connection name, and *it's not used* to add secrets to the configuration.

> **Tip**

The `connectionName` parameter must match the name used when adding the Azure Key Vault resource in the app host project. For more information, see Add Azure Key Vault resource.

You can then retrieve a secret-based configuration value through the normal IConfiguration APIs, or even by binding to strongly-typed classes with the options pattern. To retrieve a secret from an example service class that's been registered with the dependency injection container, consider the following snippets:

Retrieve `IConfiguration` instance

C#

```
public class ExampleService(IConfiguration configuration)
{
    // Use configuration...
    private string _secretValue = configuration["SecretKey"];
}
```

The preceding example assumes that you've also registered the `IConfiguration` instance for dependency injection. For more information, see Dependency injection in .NET.

Retrieve `IOptions<T>` instance

C#

```
public class ExampleService(IOptions<SecretOptions> options)
{
    // Use options...
    private string _secretValue = options.Value.SecretKey;
}
```

The preceding example assumes that you've configured a `SecretOptions` class for use with the options pattern. For more information, see Options pattern in .NET.

Additional `AddAzureKeyVaultSecrets` API parameters are available optionally for the following scenarios:

- `Action<AzureSecurityKeyVaultSettings>? configureSettings`: To set up some or all the options inline.
- `Action<SecretClientOptions>? configureClientOptions`: To set up the SecretClientOptions inline.

- `AzureKeyVaultConfigurationOptions? options`: To configure the AzureKeyVaultConfigurationOptions inline.

Add an Azure Secret client

Alternatively, you can use the SecretClient directly to retrieve the secrets on demand. This requires a slightly different registration API.

In the Program.cs file of your client-consuming project, call the AddAzureKeyVaultClient extension on the IHostApplicationBuilder instance to register a `SecretClient` for use via the dependency injection container.

C#

```
builder.AddAzureKeyVaultClient(connectionName: "key-vault");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Azure Key Vault resource in the app host project. For more information, see **Add Azure Key Vault resource**.

After adding the `SecretClient` to the builder, you can get the SecretClient instance using dependency injection. For example, to retrieve the client from an example service define it as a constructor parameter and ensure the `ExampleService` class is registered with the dependency injection container:

C#

```
public class ExampleService(SecretClient client)
{
    // Use client...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed Azure Key Vault client

There might be situations where you want to register multiple `SecretClient` instances with different connection names. To register keyed Azure Key Vault clients, call the AddKeyedAzureKeyVaultClient method:

C#

```
builder.AddKeyedAzureKeyVaultClient(name: "feature-toggles");
builder.AddKeyedAzureKeyVaultClient(name: "admin-portal");
```

Then you can retrieve the `SecretClient` instances using dependency injection. For example, to retrieve the client from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("feature-toggles")] SecretClient
featureTogglesClient,
    [FromKeyedServices("admin-portal")] SecretClient adminPortalClient)
{
    // Use clients...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire Azure Key Vault integration provides multiple options to configure the `SecretClient` based on the requirements and conventions of your project.

Use configuration providers

The .NET Aspire Azure Key Vault integration supports Microsoft.Extensions.Configuration. It loads the AzureSecurityKeyVaultSettings from *appsettings.json* or other configuration files using `Aspire:Azure:Security:KeyVault` key.

JSON

```
{
  "Aspire": {
    "Azure": {
      "Security": {
        "KeyVault": {
          "DisableHealthChecks": true,
          "DisableTracing": false,
          "ClientOptions": {
            "Diagnostics": {
              "ApplicationId": "myapp"
            }
          }
        }
      }
```

```
    }
  }
}
```

For the complete Azure Key Vault client integration JSON schema, see Aspire.Azure.Security.KeyVault/ConfigurationSchema.json.

If you have set up your configurations in the `Aspire:Azure:Security:KeyVault` section of your *appsettings.json* file you can just call the method `AddAzureKeyVaultSecrets` without passing any parameters.

Use inline delegates

You can also pass the `Action<AzureSecurityKeyVaultSettings>` delegate to set up some or all the options inline, for example to set the AzureSecurityKeyVaultSettings.VaultUri:

C#

```
builder.AddAzureKeyVaultSecrets(
    connectionName: "key-vault",
    configureSettings: settings => settings.VaultUri = new
Uri("KEY_VAULT_URI"));
```

You can also set up the SecretClientOptions using `Action<SecretClientOptions>` delegate, which is an optional parameter of the `AddAzureKeyVaultSecrets` method. For example to set the KeyClientOptions.DisableChallengeResourceVerification ID to identify the client:

C#

```
builder.AddAzureKeyVaultSecrets(
    connectionName: "key-vault",
    configureClientOptions: options =>
options.DisableChallengeResourceVerification = true))
```

Configuration options

The following configurable options are exposed through the AzureSecurityKeyVaultSettings class:

Expand table

Name	Description
AzureSecurityKeyVaultSettings.Credential	The credential used to authenticate to the Azure Key Vault.
AzureSecurityKeyVaultSettings.DisableHealthChecks	A boolean value that indicates whether the Key Vault health check is disabled or not.
AzureSecurityKeyVaultSettings.DisableTracing	A boolean value that indicates whether the OpenTelemetry tracing is disabled or not.
AzureSecurityKeyVaultSettings.VaultUri	A URI to the vault on which the client operates. Appears as "DNS Name" in the Azure portal.

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

The .NET Aspire Azure Key Vault integration includes the following health checks:

- Adds the `AzureKeyVaultSecretsHealthCheck` health check, which attempts to connect to and query the Key Vault
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Azure Key Vault integration uses the following log categories:

- `Azure.Core`
- `Azure.Identity`

Tracing

The .NET Aspire Azure Key Vault integration will emit the following tracing activities using OpenTelemetry:

- `Azure.Security.KeyVault.Secrets.SecretClient`

Metrics

The .NET Aspire Azure Key Vault integration currently does not support metrics by default due to limitations with the Azure SDK.

See also

- Azure Key Vault docs
- Video: Introduction to Azure Key Vault and .NET Aspire
- .NET Aspire Azure integrations overview
- .NET Aspire integrations overview
- .NET Aspire GitHub repo

.NET Aspire Azure PostgreSQL integration

Article • 01/31/2025

Includes: ✅ Hosting integration and ✅ Client integration

Azure Database for PostgreSQL—Flexible Server is a relational database service based on the open-source Postgres database engine. It's a fully managed database-as-a-service that can handle mission-critical workloads with predictable performance, security, high availability, and dynamic scalability. The .NET Aspire Azure PostgreSQL integration provides a way to connect to existing Azure PostgreSQL databases, or create new instances from .NET with the docker.io/library/postgres container image.

Hosting integration

The .NET Aspire Azure PostgreSQL hosting integration models a PostgreSQL flexible server and database as the AzurePostgresFlexibleServerResource and AzurePostgresFlexibleServerDatabaseResource types. Other types that are inherently available in the hosting integration are represented in the following resources:

- PostgresServerResource
- PostgresDatabaseResource
- PgAdminContainerResource
- PgWebContainerResource

To access these types and APIs for expressing them as resources in your app host project, install the Aspire.Hosting.Azure.PostgreSQL NuGet package:

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.PostgreSQL
```

For more information, see dotnet add package.

The Azure PostgreSQL hosting integration takes a dependency on the Aspire.Hosting.PostgreSQL NuGet package, extending it to support Azure. Everything that you can do with the .NET Aspire PostgreSQL integration and .NET Aspire PostgreSQL Entity Framework Core integration you can also do with this integration.

Add Azure PostgreSQL server resource

After you've installed the .NET Aspire Azure PostgreSQL hosting integration, call the AddAzurePostgresFlexibleServer extension method in your app host project:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddAzurePostgresFlexibleServer("postgres");
var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);
```

The preceding call to `AddAzurePostgresFlexibleServer` configures the PostgresSQL server resource to be deployed as an Azure Postgres Flexible Server.

> **Important**
>
> By default, `AddAzurePostgresFlexibleServer` configures Microsoft Entra ID authentication. This requires changes to applications that need to connect to these resources. For more information, see Client integration.

> **Tip**
>
> When you call AddAzurePostgresFlexibleServer, it implicitly calls AddAzureProvisioning—which adds support for generating Azure resources dynamically during app startup. The app must configure the appropriate subscription and location. For more information, see Local provisioning: Configuration.

Generated provisioning Bicep

If you're new to Bicep, it's a domain-specific language for defining Azure resources. With .NET Aspire, you don't need to write Bicep by hand, because the provisioning APIs generate Bicep for you. When you publish your app, the generated Bicep is output alongside the manifest file. When you add an Azure PostgreSQL resource, the following Bicep is generated:

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location

param principalId string

param principalType string

param principalName string

resource postgres_flexible 'Microsoft.DBforPostgreSQL/flexibleServers@2024-
08-01' = {
  name: take('postgresflexible-${uniqueString(resourceGroup().id)}', 63)
  location: location
  properties: {
    authConfig: {
      activeDirectoryAuth: 'Enabled'
      passwordAuth: 'Disabled'
    }
    availabilityZone: '1'
    backup: {
      backupRetentionDays: 7
      geoRedundantBackup: 'Disabled'
    }
    highAvailability: {
      mode: 'Disabled'
    }
    storage: {
      storageSizeGB: 32
    }
    version: '16'
  }
  sku: {
    name: 'Standard_B1ms'
    tier: 'Burstable'
  }
  tags: {
    'aspire-resource-name': 'postgres-flexible'
  }
}

resource postgreSqlFirewallRule_AllowAllAzureIps
'Microsoft.DBforPostgreSQL/flexibleServers/firewallRules@2024-08-01' = {
  name: 'AllowAllAzureIps'
  properties: {
    endIpAddress: '0.0.0.0'
    startIpAddress: '0.0.0.0'
  }
  parent: postgres_flexible
}

resource postgres_flexible_admin
'Microsoft.DBforPostgreSQL/flexibleServers/administrators@2024-08-01' = {
  name: principalId
  properties: {
```

```
    principalName: principalName
    principalType: principalType
  }
  parent: postgres_flexible
  dependsOn: [
    postgres_flexible
    postgreSqlFirewallRule_AllowAllAzureIps
  ]
}

output connectionString string =
'Host=${postgres_flexible.properties.fullyQualifiedDomainName};Username=${pr
incipalName}'
```

The preceding Bicep is a module that provisions an Azure PostgreSQL flexible server with the following defaults:

- `authConfig`: The authentication configuration of the PostgreSQL server. The default is `ActiveDirectoryAuth` enabled and `PasswordAuth` disabled.
- `availabilityZone`: The availability zone of the PostgreSQL server. The default is `1`.
- `backup`: The backup configuration of the PostgreSQL server. The default is `BackupRetentionDays` set to `7` and `GeoRedundantBackup` set to `Disabled`.
- `highAvailability`: The high availability configuration of the PostgreSQL server. The default is `Disabled`.
- `storage`: The storage configuration of the PostgreSQL server. The default is `StorageSizeGB` set to `32`.
- `version`: The version of the PostgreSQL server. The default is `16`.
- `sku`: The SKU of the PostgreSQL server. The default is `Standard_B1ms`.
- `tags`: The tags of the PostgreSQL server. The default is `aspire-resource-name` set to the name of the Aspire resource, in this case `postgres-flexible`.

In addition to the PostgreSQL flexible server, it also provisions an Azure Firewall rule to allow all Azure IP addresses. Finally, an administrator is created for the PostgreSQL server, and the connection string is outputted as an output variable. The generated Bicep is a starting point and is influenced by changes to the provisioning infrastructure in C#. Customizations to the Bicep file directly will be overwritten, so make changes through the C# provisioning APIs to ensure they are reflected in the generated files.

Customize provisioning infrastructure

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This type enables the customization of the generated Bicep by providing a fluent API to configure the Azure resources by using the ConfigureInfrastructure<T>

(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API. For example, you can configure the `kind`, `consistencyPolicy`, `locations`, and more. The following example demonstrates how to customize the PostgreSQL server resource:

C#

```
builder.AddAzurePostgresFlexibleServer("postgres")
    .ConfigureInfrastructure(infra =>
    {
        var flexibleServer = infra.GetProvisionableResources()
                                  .OfType<PostgreSqlFlexibleServer>()
                                  .Single();

        flexibleServer.Sku = new PostgreSqlFlexibleServerSku
        {
            Tier = PostgreSqlFlexibleServerSkuTier.Burstable,
        };
        flexibleServer.HighAvailability = new
PostgreSqlFlexibleServerHighAvailability
        {
            Mode =
PostgreSqlFlexibleServerHighAvailabilityMode.ZoneRedundant,
            StandbyAvailabilityZone = "2",
        };
        flexibleServer.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
  - The `infra` parameter is an instance of the AzureResourceInfrastructure type.
  - The provisionable resources are retrieved by calling the GetProvisionableResources() method.
  - The single PostgreSqlFlexibleServer is retrieved.
  - The `sku` is set with PostgreSqlFlexibleServerSkuTier.Burstable.
  - The high availability properties are set with PostgreSqlFlexibleServerHighAvailabilityMode.ZoneRedundant in standby availability zone `"2"`.
  - A tag is added to the flexible server with a key of `ExampleKey` and a value of `Example value`.

There are many more configuration options available to customize the PostgreSQL flexible server resource. For more information, see Azure.Provisioning.PostgreSql and Azure.Provisioning customization.

Connect to an existing Azure PostgreSQL flexible server

You might have an existing Azure PostgreSQL flexible server that you want to connect to. Instead of representing a new Azure PostgreSQL flexible server resource, you can add a connection string to the app host. To add a connection to an existing Azure PostgreSQL flexible server, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddConnectionString("postgres");

builder.AddProject<Projects.WebApplication>("web")
       .WithReference(postgres);

// After adding all resources, run the app...
```

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section. The app host injects this connection string as an environment variable into all dependent resources, for example:

JSON

```
{
    "ConnectionStrings": {
        "postgres": "Server=<PostgreSQL-server-
name>.postgres.database.azure.com;Database=<database-name>;Port=5432;Ssl
Mode=Require;User Id=<username>;"
    }
}
```

The dependent resource can access the injected connection string by calling the GetConnectionString method, and passing the connection name as the parameter, in this case `"postgres"`. The `GetConnectionString` API is shorthand for `IConfiguration.GetSection("ConnectionStrings")[name]`.

Run Azure PostgreSQL resource as a container

The Azure PostgreSQL hosting integration supports running the PostgreSQL server as a local container. This is beneficial for situations where you want to run the PostgreSQL server locally for development and testing purposes, avoiding the need to provision an Azure resource or connect to an existing Azure PostgreSQL server.

To run the PostgreSQL server as a container, call the RunAsContainer method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddAzurePostgresFlexibleServer("postgres")
                      .RunAsContainer();

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);
```

The preceding code configures an Azure PostgreSQL Flexible Server resource to run locally in a container.

> **Tip**
>
> The `RunAsContainer` method is useful for local development and testing. The API exposes an optional delegate that enables you to customize the underlying PostgresServerResource configuration. For example, you can add pgAdmin and pgWeb, add a data volume or data bind mount, and add an init bind mount. For more information, see the .NET Aspire PostgreSQL hosting integration section.

Configure the Azure PostgreSQL server to use password authentication

By default, the Azure PostgreSQL server is configured to use Microsoft Entra ID authentication. If you want to use password authentication, you can configure the server to use password authentication by calling the WithPasswordAuthentication method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var username = builder.AddParameter("username", secret: true);
var password = builder.AddParameter("password", secret: true);
```

```
var postgres = builder.AddAzurePostgresFlexibleServer("postgres")
                      .WithPasswordAuthentication(username, password);

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);
```

The preceding code configures the Azure PostgreSQL server to use password authentication. The `username` and `password` parameters are added to the app host as parameters, and the `WithPasswordAuthentication` method is called to configure the Azure PostgreSQL server to use password authentication. For more information, see External parameters.

Client integration

To get started with the .NET Aspire PostgreSQL client integration, install the Aspire.Npgsql NuGet package in the client-consuming project, that is, the project for the application that uses the PostgreSQL client. The PostgreSQL client integration registers an NpgsqlDataSource instance that you can use to interact with PostgreSQL.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Npgsql
```

Add Npgsql client

In the *Program.cs* file of your client-consuming project, call the AddNpgsqlDataSource extension method on any IHostApplicationBuilder to register an `NpgsqlDataSource` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddNpgsqlDataSource(connectionName: "postgresdb");
```

Tip

The `connectionName` parameter must match the name used when adding the PostgreSQL server resource in the app host project. For more information, see **Add PostgreSQL server resource**.

After adding `NpgsqlDataSource` to the builder, you can get the `NpgsqlDataSource` instance using dependency injection. For example, to retrieve your data source object from an example service define it as a constructor parameter and ensure the `ExampleService` class is registered with the dependency injection container:

C#

```
public class ExampleService(NpgsqlDataSource dataSource)
{
    // Use dataSource...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed Npgsql client

There might be situations where you want to register multiple `NpgsqlDataSource` instances with different connection names. To register keyed Npgsql clients, call the AddKeyedNpgsqlDataSource method:

C#

```
builder.AddKeyedNpgsqlDataSource(name: "chat");
builder.AddKeyedNpgsqlDataSource(name: "queue");
```

Then you can retrieve the `NpgsqlDataSource` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("chat")] NpgsqlDataSource chatDataSource,
    [FromKeyedServices("queue")] NpgsqlDataSource queueDataSource)
{
    // Use data sources...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire PostgreSQL integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling the AddNpgsqlDataSource method:

C#

```
builder.AddNpgsqlDataSource("postgresdb");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "postgresdb": "Host=myserver;Database=postgresdb"
  }
}
```

For more information, see the ConnectionString.

Use configuration providers

The .NET Aspire PostgreSQL integration supports Microsoft.Extensions.Configuration. It loads the NpgsqlSettings from *appsettings.json* or other configuration files by using the `Aspire:Npgsql` key. Example *appsettings.json* that configures some of the options:

The following example shows an *appsettings.json* file that configures some of the available options:

JSON

```
{
  "Aspire": {
    "Npgsql": {
      "ConnectionString": "Host=myserver;Database=postgresdb",
      "DisableHealthChecks": false,
```

```
      "DisableTracing": true,
      "DisableMetrics": false
    }
  }
}
```

For the complete PostgreSQL client integration JSON schema, see Aspire.Npgsql/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<NpgsqlSettings> configureSettings` delegate to set up some or all the options inline, for example to disable health checks:

C#

```
builder.AddNpgsqlDataSource(
    "postgresdb",
     static settings => settings.DisableHealthChecks = true);
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

- Adds the NpgSqlHealthCheck, which verifies that commands can be successfully executed against the underlying Postgres database.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not

metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire PostgreSQL integration uses the following log categories:

- `Npgsql.Connection`
- `Npgsql.Command`
- `Npgsql.Transaction`
- `Npgsql.Copy`
- `Npgsql.Replication`
- `Npgsql.Exception`

Tracing

The .NET Aspire PostgreSQL integration will emit the following tracing activities using OpenTelemetry:

- `Npgsql`

Metrics

The .NET Aspire PostgreSQL integration will emit the following metrics using OpenTelemetry:

- Npgsql:
  - `ec_Npgsql_bytes_written_per_second`
  - `ec_Npgsql_bytes_read_per_second`
  - `ec_Npgsql_commands_per_second`
  - `ec_Npgsql_total_commands`
  - `ec_Npgsql_current_commands`
  - `ec_Npgsql_failed_commands`
  - `ec_Npgsql_prepared_commands_ratio`
  - `ec_Npgsql_connection_pools`
  - `ec_Npgsql_multiplexing_average_commands_per_batch`
  - `ec_Npgsql_multiplexing_average_write_time_per_batch`

Add Azure authenticated Npgsql client

By default, when you call `AddAzurePostgresFlexibleServer` in your PostgreSQL hosting integration, it configures Azure.Identity NuGet package to enable authentication:

.NET CLI

.NET CLI

```
dotnet add package Azure.Identity
```

The PostgreSQL connection can be consumed using the client integration and Azure.Identity:

C#

```
builder.AddNpgsqlDataSource(
    "postgresdb",
    configureDataSourceBuilder: (dataSourceBuilder) =>
{
    if
(string.IsNullOrEmpty(dataSourceBuilder.ConnectionStringBuilder.Password))
    {
        var credentials = new DefaultAzureCredential();
        var tokenRequest = new TokenRequestContext(["https://ossrdbms-
aad.database.windows.net/.default"]);

        dataSourceBuilder.UsePasswordProvider(
            passwordProvider: _ => credentials.GetToken(tokenRequest).Token,
            passwordProviderAsync: async (_, ct) => (await
credentials.GetTokenAsync(tokenRequest, ct)).Token);
    }
});
```

The preceding code snippet demonstrates how to use the DefaultAzureCredential class from the Azure.Identity package to authenticate with Microsoft Entra ID and retrieve a token to connect to the PostgreSQL database. The UsePasswordProvider method is used to provide the token to the data source builder.

See also

- PostgreSQL docs
- Azure Database for PostgreSQL
- .NET Aspire Azure PostgreSQL Entity Framework Core integration
- .NET Aspire PostgreSQL integration
- .NET Aspire integrations

- .NET Aspire GitHub repo

.NET Aspire Azure OpenAI integration (Preview)

Article • 03/07/2025

Includes: ✅ Hosting integration and ✅ Client integration

Azure OpenAI Service provides access to OpenAI's powerful language and embedding models with the security and enterprise promise of Azure. The .NET Aspire Azure OpenAI integration enables you to connect to Azure OpenAI Service or OpenAI's API from your .NET applications.

Hosting integration

The .NET Aspire Azure OpenAI hosting integration models Azure OpenAI resources as AzureOpenAIResource. To access these types and APIs for expressing them within your app host project, install the Aspire.Hosting.Azure.CognitiveServices NuGet package:

.NET CLI

```
.NET CLI
dotnet add package Aspire.Hosting.Azure.CognitiveServices
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add an Azure OpenAI resource

To add an AzureOpenAIResource to your app host project, call the AddAzureOpenAI method:

```
C#
var builder = DistributedApplication.CreateBuilder(args);

var openai = builder.AddAzureOpenAI("openai");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(openai);
```

```
// After adding all resources, run the app...
```

The preceding code adds an Azure OpenAI resource named `openai` to the app host project. The WithReference method passes the connection information to the `ExampleProject` project.

> **ⓘ Important**
>
> When you call AddAzureOpenAI, it implicitly calls AddAzureProvisioning(IDistributedApplicationBuilder)—which adds support for generating Azure resources dynamically during app startup. The app must configure the appropriate subscription and location. For more information, see Local provisioning: Configuration.

Add an Azure OpenAI deployment resource

To add an Azure OpenAI deployment resource, call the AddDeployment(IResourceBuilder<AzureOpenAIResource>, AzureOpenAIDeployment) method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var openai = builder.AddAzureOpenAI("openai");
openai.AddDeployment(
    new AzureOpenAIDeployment(
        name: "preview",
        modelName: "gpt-4.5-preview",
        modelVersion: "2025-02-27"));

builder.AddProject<Projects.ExampleProject>()
       .WithReference(openai)
       .WaitFor(openai);

// After adding all resources, run the app...
```

The preceding code:

- Adds an Azure OpenAI resource named `openai`.
- Adds an Azure OpenAI deployment resource named `preview` with a model name of `gpt-4.5-preview`. The model name must correspond to an available model in the Azure OpenAI service.

Generated provisioning Bicep

If you're new to Bicep, it's a domain-specific language for defining Azure resources. With .NET Aspire, you don't need to write Bicep by-hand, instead the provisioning APIs generate Bicep for you. When you publish your app, the generated Bicep provisions an Azure OpenAI resource with standard defaults.

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location

param principalType string

param principalId string

resource openai 'Microsoft.CognitiveServices/accounts@2024-10-01' = {
  name: take('openai-${uniqueString(resourceGroup().id)}', 64)
  location: location
  kind: 'OpenAI'
  properties: {
    customSubDomainName: toLower(take(concat('openai',
uniqueString(resourceGroup().id)), 24))
    publicNetworkAccess: 'Enabled'
    disableLocalAuth: true
  }
  sku: {
    name: 'S0'
  }
  tags: {
    'aspire-resource-name': 'openai'
  }
}

resource openai_CognitiveServicesOpenAIContributor
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(openai.id, principalId,
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', 'a001fd3d-
188f-4b5d-821b-7da978bf7442'))
  properties: {
    principalId: principalId
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', 'a001fd3d-
188f-4b5d-821b-7da978bf7442')
    principalType: principalType
  }
  scope: openai
}

resource preview 'Microsoft.CognitiveServices/accounts/deployments@2024-10-
01' = {
  name: 'preview'
```

```
  properties: {
    model: {
      format: 'OpenAI'
      name: 'gpt-4.5-preview'
      version: '2025-02-27'
    }
  }
  sku: {
    name: 'Standard'
    capacity: 8
  }
  parent: openai
}

output connectionString string = 'Endpoint=${openai.properties.endpoint}'
```

The preceding Bicep is a module that provisions an Azure Cognitive Services resource with the following defaults:

- `location`: The location of the resource group.
- `principalType`: The principal type of the Cognitive Services resource.
- `principalId`: The principal ID of the Cognitive Services resource.
- `openai`: The Cognitive Services account resource.
  - `kind`: The kind of the resource, set to `OpenAI`.
  - `properties`: The properties of the resource.
    - `customSubDomainName`: The custom subdomain name for the resource, based on the unique string of the resource group ID.
    - `publicNetworkAccess`: Set to `Enabled`.
    - `disableLocalAuth`: Set to `true`.
  - `sku`: The SKU of the resource, set to `S0`.
- `openai_CognitiveServicesOpenAIContributor`: The Cognitive Services resource owner, based on the build-in `Azure Cognitive Services OpenAI Contributor` role. For more information, see Azure Cognitive Services OpenAI Contributor.
- `preview`: The deployment resource, based on the `preview` name.
  - `properties`: The properties of the deployment resource.
    - `format`: The format of the deployment resource, set to `OpenAI`.
    - `modelName`: The model name of the deployment resource, set to `gpt-4.5-preview`.
    - `modelVersion`: The model version of the deployment resource, set to `2025-02-27`.
- `connectionString`: The connection string, containing the endpoint of the Cognitive Services resource.

The generated Bicep is a starting point and is influenced by changes to the provisioning infrastructure in C#. Customizations to the Bicep file directly will be overwritten, so make changes through the C# provisioning APIs to ensure they are reflected in the generated files.

Customize provisioning infrastructure

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This enables customization of the generated Bicep by providing a fluent API to configure the Azure resources—using the ConfigureInfrastructure<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API:

C#

```
builder.AddAzureOpenAI("openai")
    .ConfigureInfrastructure(infra =>
    {
        var resources = infra.GetProvisionableResources();
        var account = resources.OfType<CognitiveServicesAccount>().Single();

        account.Sku = new CognitiveServicesSku
        {
            Tier = CognitiveServicesSkuTier.Enterprise,
            Name = "E0"
        };
        account.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
  - The `infra` parameter is an instance of the AzureResourceInfrastructure type.
  - The provisionable resources are retrieved by calling the GetProvisionableResources() method.
  - The single CognitiveServicesAccount resource is retrieved.
  - The CognitiveServicesAccount.Sku property is assigned to a new instance of CognitiveServicesSku with an `E0` name and CognitiveServicesSkuTier.Enterprise tier.
  - A tag is added to the Cognitive Services resource with a key of `ExampleKey` and a value of `Example value`.

Connect to an existing Azure OpenAI service

You might have an existing Azure OpenAI service that you want to connect to. You can chain a call to annotate that your AzureOpenAIResource is an existing resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var existingOpenAIName = builder.AddParameter("existingOpenAIName");
var existingOpenAIResourceGroup =
builder.AddParameter("existingOpenAIResourceGroup");

var openai = builder.AddAzureOpenAI("openai")
                    .AsExisting(existingOpenAIName,
existingOpenAIResourceGroup);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(openai);

// After adding all resources, run the app...
```

For more information on treating Azure OpenAI resources as existing resources, see Use existing Azure resources.

Alternatively, instead of representing an Azure OpenAI resource, you can add a connection string to the app host. Which is a weakly-typed approach that's based solely on a `string` value. To add a connection to an existing Azure OpenAI service, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var openai = builder.ExecutionContext.IsPublishMode
    ? builder.AddAzureOpenAI("openai")
    : builder.AddConnectionString("openai");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(openai);

// After adding all resources, run the app...
```

> ⓘ **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other

services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section:

JSON

```
{
  "ConnectionStrings": {
    "openai": "https://{account_name}.openai.azure.com/"
  }
}
```

For more information, see Add existing Azure resources with connection strings.

Client integration

To get started with the .NET Aspire Azure OpenAI client integration, install the Aspire.Azure.AI.OpenAI NuGet package in the client-consuming project, that is, the project for the application that uses the Azure OpenAI client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Azure.AI.OpenAI
```

Add an Azure OpenAI client

In the *Program.cs* file of your client-consuming project, use the AddAzureOpenAIClient(IHostApplicationBuilder, String, Action<AzureOpenAISettings>, Action<IAzureClientBuilder<AzureOpenAIClient,AzureOpenAIClientOptions>>) method on any IHostApplicationBuilder to register an `OpenAIClient` for dependency injection (DI). The `AzureOpenAIClient` is a subclass of `OpenAIClient`, allowing you to request either type from DI. This ensures code not dependent on Azure-specific features remains generic. The `AddAzureOpenAIClient` method requires a connection name parameter.

C#

```
builder.AddAzureOpenAIClient(connectionName: "openai");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Azure OpenAI resource in the app host project. For more information, see **Add an Azure OpenAI resource**.

After adding the `OpenAIClient`, you can retrieve the client instance using dependency injection:

C#

```
public class ExampleService(OpenAIClient client)
{
    // Use client...
}
```

For more information, see:

- Azure.AI.OpenAI documentation for examples on using the `OpenAIClient`.
- Dependency injection in .NET for details on dependency injection.
- Quickstart: Get started using GPT-35-Turbo and GPT-4 with Azure OpenAI Service.

Add Azure OpenAI client with registered `IChatClient`

If you're interested in using the IChatClient interface, with the OpenAI client, simply chain either of the following APIs to the `AddAzureOpenAIClient` method:

- AddChatClient(AspireOpenAIClientBuilder, String): Registers a singleton `IChatClient` in the services provided by the AspireOpenAIClientBuilder.
- AddKeyedChatClient(AspireOpenAIClientBuilder, String, String): Registers a keyed singleton `IChatClient` in the services provided by the AspireOpenAIClientBuilder.

For example, consider the following C# code that adds an `IChatClient` to the DI container:

C#

```
builder.AddAzureOpenAIClient(connectionName: "openai")
       .AddChatClient("deploymentName");
```

Similarly, you can add a keyed `IChatClient` with the following C# code:

```
C#

builder.AddAzureOpenAIClient(connectionName: "openai")
       .AddKeyedChatClient("serviceKey", "deploymentName");
```

For more information on the `IChatClient` and its corresponding library, see Artificial intelligence in .NET (Preview).

Configure Azure OpenAI client settings

The .NET Aspire Azure OpenAI library provides a set of settings to configure the Azure OpenAI client. The `AddAzureOpenAIClient` method exposes an optional `configureSettings` parameter of type `Action<AzureOpenAISettings>?`. To configure settings inline, consider the following example:

```
C#

builder.AddAzureOpenAIClient(
    connectionName: "openai",
    configureSettings: settings =>
    {
        settings.DisableTracing = true;

        var uriString = builder.Configuration["AZURE_OPENAI_ENDPOINT"]
            ?? throw new InvalidOperationException("AZURE_OPENAI_ENDPOINT is
not set.");

        settings.Endpoint = new Uri(uriString);
    });
```

The preceding code sets the AzureOpenAISettings.DisableTracing property to `true`, and sets the AzureOpenAISettings.Endpoint property to the Azure OpenAI endpoint.

Configure Azure OpenAI client builder options

To configure the AzureOpenAIClientOptions for the client, you can use the AddAzureOpenAIClient method. This method takes an optional `configureClientBuilder` parameter of type `Action<IAzureClientBuilder<OpenAIClient, AzureOpenAIClientOptions>>?`. Consider the following example:

```
C#
```

```
builder.AddAzureOpenAIClient(
    connectionName: "openai",
    configureClientBuilder: clientBuilder =>
    {
        clientBuilder.ConfigureOptions(options =>
        {
            options.UserAgentApplicationId = "CLIENT_ID";
        });
    });
```

The client builder is an instance of the IAzureClientBuilder<TClient,TOptions> type, which provides a fluent API to configure the client options. The preceding code sets the AzureOpenAIClientOptions.UserAgentApplicationId property to `CLIENT_ID`. For more information, see ConfigureOptions(ChatClientBuilder, Action<ChatOptions>).

Add Azure OpenAI client from configuration

Additionally, the package provides the AddOpenAIClientFromConfiguration(IHostApplicationBuilder, String) extension method to register an `OpenAIClient` or `AzureOpenAIClient` instance based on the provided connection string. This method follows these rules:

- If the `Endpoint` attribute is empty or missing, an `OpenAIClient` instance is registered using the provided key, for example, `Key={key};`.
- If the `IsAzure` attribute is `true`, an `AzureOpenAIClient` is registered; otherwise, an `OpenAIClient` is registered, for example, `Endpoint={azure_endpoint};Key={key};IsAzure=true` registers an `AzureOpenAIClient`, while `Endpoint=https://localhost:18889;Key={key}` registers an `OpenAIClient`.
- If the `Endpoint` attribute contains `".azure."`, an `AzureOpenAIClient` is registered; otherwise, an `OpenAIClient` is registered, for example, `Endpoint=https://{account}.azure.com;Key={key};`.

Consider the following example:

C#

```
builder.AddOpenAIClientFromConfiguration("openai");
```

> **Tip**
>
> A valid connection string must contain at least an `Endpoint` or a `Key`.

Consider the following example connection strings and whether they register an `OpenAIClient` or `AzureOpenAIClient`:

Expand table

Example connection string	Registered client type
`Endpoint=https://{account_name}.openai.azure.com/;Key={account_key}`	`AzureOpenAIClient`
`Endpoint=https://{account_name}.openai.azure.com/;Key={account_key};IsAzure=false`	`OpenAIClient`
`Endpoint=https://{account_name}.openai.azure.com/;Key={account_key};IsAzure=true`	`AzureOpenAIClient`
`Endpoint=https://localhost:18889;Key={account_key}`	`OpenAIClient`

Add keyed Azure OpenAI clients

There might be situations where you want to register multiple `OpenAIClient` instances with different connection names. To register keyed Azure OpenAI clients, call the AddKeyedAzureOpenAIClient method:

C#

```
builder.AddKeyedAzureOpenAIClient(name: "chat");
builder.AddKeyedAzureOpenAIClient(name: "code");
```

> **Important**
>
> When using keyed services, ensure that your Azure OpenAI resource configures two named connections, one for `chat` and one for `code`.

Then you can retrieve the client instances using dependency injection. For example, to retrieve the clients from a service:

C#

```
public class ExampleService(
    [KeyedService("chat")] OpenAIClient chatClient,
    [KeyedService("code")] OpenAIClient codeClient)
{
```

```
    // Use clients...
}
```

For more information, see Keyed services in .NET.

Add keyed Azure OpenAI clients from configuration

The same functionality and rules exist for keyed Azure OpenAI clients as for the nonkeyed clients. You can use the AddKeyedOpenAIClientFromConfiguration(IHostApplicationBuilder, String) extension method to register an `OpenAIClient` or `AzureOpenAIClient` instance based on the provided connection string.

Consider the following example:

C#

```
builder.AddKeyedOpenAIClientFromConfiguration("openai");
```

This method follows the same rules as detailed in the Add Azure OpenAI client from configuration.

Configuration

The .NET Aspire Azure OpenAI library provides multiple options to configure the Azure OpenAI connection based on the requirements and conventions of your project. Either a `Endpoint` or a `ConnectionString` is required to be supplied.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddAzureOpenAIClient`:

C#

```
builder.AddAzureOpenAIClient("openai");
```

The connection string is retrieved from the `ConnectionStrings` configuration section, and there are two supported formats:

Account endpoint

The recommended approach is to use an **Endpoint**, which works with the `AzureOpenAISettings.Credential` property to establish a connection. If no credential is configured, the DefaultAzureCredential is used.

JSON

```
{
  "ConnectionStrings": {
    "openai": "https://{account_name}.openai.azure.com/"
  }
}
```

For more information, see Use Azure OpenAI without keys.

Connection string

Alternatively, a custom connection string can be used:

JSON

```
{
  "ConnectionStrings": {
    "openai": "Endpoint=https://{account_name}.openai.azure.com/;Key=
{account_key};"
  }
}
```

In order to connect to the non-Azure OpenAI service, drop the `Endpoint` property and only set the Key property to set the API key.

Use configuration providers

The .NET Aspire Azure OpenAI integration supports Microsoft.Extensions.Configuration. It loads the `AzureOpenAISettings` from configuration by using the `Aspire:Azure:AI:OpenAI` key. Example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "Azure": {
      "AI": {
```

```
        "OpenAI": {
          "DisableTracing": false
        }
      }
    }
  }
}
```

For the complete Azure OpenAI client integration JSON schema, see Aspire.Azure.AI.OpenAI/ConfigurationSchema.json.

Use inline delegates

You can pass the `Action<AzureOpenAISettings> configureSettings` delegate to set up some or all the options inline, for example to disable tracing from code:

C#

```
builder.AddAzureOpenAIClient(
    "openai",
    static settings => settings.DisableTracing = true);
```

You can also set up the OpenAIClientOptions using the optional `Action<IAzureClientBuilder<OpenAIClient, OpenAIClientOptions>> configureClientBuilder` parameter of the `AddAzureOpenAIClient` method. For example, to set the client ID for this client:

C#

```
builder.AddAzureOpenAIClient(
    "openai",
    configureClientBuilder: builder => builder.ConfigureOptions(
        options => options.Diagnostics.ApplicationId = "CLIENT_ID"));
```

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Azure OpenAI integration uses the following log categories:

- `Azure`
- `Azure.Core`
- `Azure.Identity`

Tracing

The .NET Aspire Azure OpenAI integration emits tracing activities using OpenTelemetry for operations performed with the `OpenAIClient`.

> **ⓘ Important**
>
> Tracing is currently experimental with this integration. To opt-in to it, set either the `OPENAI_EXPERIMENTAL_ENABLE_OPEN_TELEMETRY` environment variable to `true` or `1`, or call `AppContext.SetSwitch("OpenAI.Experimental.EnableOpenTelemetry", true))` during app startup.

See also

- Azure OpenAI
- .NET Aspire integrations overview
- .NET Aspire Azure integrations overview
- .NET Aspire GitHub repo

.NET Aspire Azure SignalR Service integration

Article • 03/27/2025

Includes: ✅ Hosting integration not ⊗ Client integration

Azure SignalR Service is a fully managed real-time messaging service that simplifies adding real-time web functionality to your applications. The .NET Aspire Azure SignalR Service integration enables you to easily provision, configure, and connect your .NET applications to Azure SignalR Service instances.

This article describes how to integrate Azure SignalR Service into your .NET Aspire applications, covering both hosting and client integration.

Hosting integration

The .NET Aspire Azure SignalR Service hosting integration models Azure SignalR resources as the following types:

- AzureSignalRResource: Represents an Azure SignalR Service resource, including connection information to the underlying Azure resource.
- AzureSignalREmulatorResource: Represents an emulator for Azure SignalR Service, allowing local development and testing without requiring an Azure subscription.

To access the hosting types and APIs for expressing these resources in the distributed application builder, install the 📦 Aspire.Hosting.Azure.SignalR NuGet package in your app host project:

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.SignalR
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add an Azure SignalR Service resource

To add an Azure SignalR Service resource to your app host project, call the AddAzureSignalR method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var signalR = builder.AddAzureSignalR("signalr");

var api = builder.AddProject<Projects.ApiService>("api")
                 .WithReference(signalR)
                 .WaitFor(signalR);

builder.AddProject<Projects.WebApp>("webapp")
       .WithReference(api)
       .WaitFor(api);

// Continue configuring and run the app...
```

In the preceding example:

- An Azure SignalR Service resource named `signalr` is added.
- The `signalr` resource is referenced by the `api` project.
- The `api` project is referenced by the `webapp` project.

This architecture allows the `webapp` project to communicate with the `api` project, which in turn communicates with the Azure SignalR Service resource.

> **ⓘ Important**
>
> Calling `AddAzureSignalR` implicitly enables Azure provisioning support. Ensure your app host is configured with the appropriate Azure subscription and location. For more information, see Local provisioning: Configuration.

Generated provisioning Bicep

When you add an Azure SignalR Service resource, .NET Aspire generates provisioning infrastructure using Bicep. The generated Bicep includes defaults for location, SKU, and role assignments:

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location
```

```
param principalType string

param principalId string

resource signalr 'Microsoft.SignalRService/signalR@2024-03-01' = {
  name: take('signalr-${uniqueString(resourceGroup().id)}', 63)
  location: location
  properties: {
    cors: {
      allowedOrigins: [
        '*'
      ]
    }
    features: [
      {
        flag: 'ServiceMode'
        value: 'Default'
      }
    ]
  }
  kind: 'SignalR'
  sku: {
    name: 'Free_F1'
    capacity: 1
  }
  tags: {
    'aspire-resource-name': 'signalr'
  }
}

resource signalr_SignalRAppServer
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(signalr.id, principalId,
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '420fcaa2-
552c-430f-98ca-3264be4806c7'))
  properties: {
    principalId: principalId
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '420fcaa2-
552c-430f-98ca-3264be4806c7')
    principalType: principalType
  }
  scope: signalr
}

output hostName string = signalr.properties.hostName
```

The generated Bicep provides a starting point and can be customized further.

Customize provisioning infrastructure

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This enables customization of the generated Bicep by providing a fluent API to configure the Azure resources using the ConfigureInfrastructure<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API:

C#

```
builder.AddAzureSignalR("signalr")
    .ConfigureInfrastructure(infra =>
    {
        var signalRService = infra.GetProvisionableResources()
                                  .OfType<SignalRService>()
                                  .Single();

        signalRService.Sku.Name = "Premium_P1";
        signalRService.Sku.Capacity = 10;
        signalRService.PublicNetworkAccess = "Enabled";
        signalRService.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
  - The `infra` parameter is an instance of the AzureResourceInfrastructure type.
  - The provisionable resources are retrieved by calling the GetProvisionableResources() method.
  - The single SignalRService resource is retrieved.
  - The SignalRService.Sku property is assigned a name of `Premium_P1` and a capacity of `10`.
  - The SignalRService.PublicNetworkAccess property is set to `Enabled`.
  - A tag is added to the SignalR service resource with a key of `ExampleKey` and a value of `Example value`.

Connect to an existing Azure SignalR Service

You might have an existing Azure SignalR Service that you want to connect to. You can chain a call to annotate that your AzureSignalRResource is an existing resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var existingSignalRName = builder.AddParameter("existingSignalRName");
var existingSignalRResourceGroup =
builder.AddParameter("existingSignalRResourceGroup");
```

```
var signalr = builder.AddAzureSignalR("signalr")
                     .AsExisting(existingSignalRName,
existingSignalRResourceGroup);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(signalr);

// After adding all resources, run the app...
```

For more information on treating Azure SignalR resources as existing resources, see Use existing Azure resources.

Alternatively, instead of representing an Azure SignalR resource, you can add a connection string to the app host. Which is a weakly-typed approach that's based solely on a `string` value. To add a connection to an existing Azure SignalR Service, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var signalr = builder.ExecutionContext.IsPublishMode
    ? builder.AddAzureSignalR("signalr")
    : builder.AddConnectionString("signalr");

builder.AddProject<Projects.ApiService>("apiService")
       .WithReference(signalr);
```

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section:

JSON

```
{
  "ConnectionStrings": {
    "signalr": "Endpoint=https://your-signalr-
instance.service.signalr.net;AccessKey=your-access-key;Version=1.0;"
```

```
  }
}
```

For more information, see Add existing Azure resources with connection strings.

Add an Azure SignalR Service emulator resource

The Azure SignalR Service emulator is a local development and testing tool that emulates the behavior of Azure SignalR Service. This emulator only supports *Serverless* mode, which requires a specific configuration when using the emulator.

To use the emulator, chain a call to the RunAsEmulator(IResourceBuilder<AzureSignalRResource>, Action<IResourceBuilder<AzureSignalREmulatorResource>>) method:

C#

```
using Aspire.Hosting.Azure;

var builder = DistributedApplication.CreateBuilder(args);

var signalR = builder.AddAzureSignalR("signalr",
AzureSignalRServiceMode.Serverless)
                     .RunAsEmulator();

builder.AddProject<Projects.ApiService>("apiService")
       .WithReference(signalR)
       .WaitFor(signalR);

// After adding all resources, run the app...
```

In the preceding example, the `RunAsEmulator` method configures the Azure SignalR Service resource to run as an emulator. The emulator is based on the `mcr.microsoft.com/signalr/signalr-emulator:latest` container image. The emulator is started when the app host is run, and is stopped when the app host is stopped.

Azure SignalR Service modes

While the Azure SignalR Service emulator only supports the *Serverless* mode, the Azure SignalR Service resource can be configured to use either of the following modes:

- AzureSignalRServiceMode.Default
- AzureSignalRServiceMode.Serverless

The *Default* mode is the "default" configuration for Azure SignalR Service. Each mode has its own set of features and limitations. For more information, see Azure SignalR Service modes.

> **Important**
>
> The Azure SignalR Service emulator only works in *Serverless* mode and the `AddNamedAzureSignalR` method doesn't support *Serverless* mode.

Hub host integration

There isn't an official .NET Aspire Azure SignalR *client integration*. However, there is limited support for similar experiences. In these scenarios, the Azure SignalR Service acts as a proxy between the server (where the Hub or Hub<T> are hosted) and the client (where the SignalR client is hosted). The Azure SignalR Service routes traffic between the server and client, allowing for real-time communication.

> **Important**
>
> It's important to disambiguate between .NET Aspire client integrations and the .NET SignalR client. SignalR exposes hubs—which act as a server-side concept—and SignalR clients connect to those hubs. The .NET projects that host SignalR hubs are where you integrate with .NET Aspire. The SignalR client is a separate library that connects to those hubs, in a different project.

There are two packages available for, each with addressing specific scenarios such as managing the client connection to Azure SignalR Service, and hooking up to the Azure SignalR Service resource. To get started, install the Microsoft.Azure.SignalR NuGet package in the project hosting your SignalR hub.

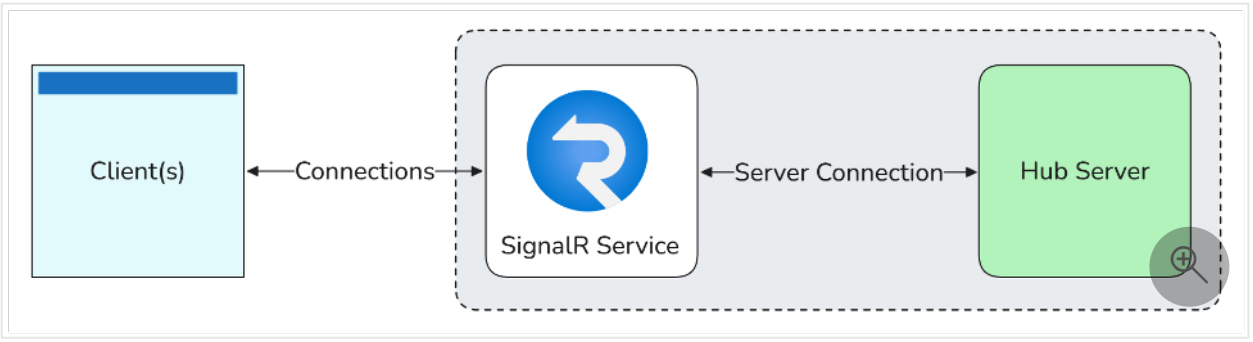
.NET CLI

.NET CLI

```
dotnet add package Microsoft.Azure.SignalR
```

Configure named Azure SignalR Service in Default mode

In *Default* mode, your consuming project needs to rely on a named Azure SignalR Service resource. Consider the following diagram that illustrates the architecture of Azure SignalR Service in *Default* mode:



For more information on *Default* mode, see Azure SignalR Service: Default mode.

In your SignalR hub host project, configure Azure SignalR Service by chaining calls to `.AddSignalR().AddNamedAzureSignalR("name")`:

```C#
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSignalR()
                .AddNamedAzureSignalR("signalr");

var app = builder.Build();

app.MapHub<ChatHub>("/chat");

app.Run();
```

The `AddNamedAzureSignalR` method configures the project to use the Azure SignalR Service resource named `signalr`. The connection string is read from the configuration key `ConnectionStrings:signalr`, and additional settings are loaded from the `Azure:SignalR:signalr` configuration section.

> **Note**
>
> If you're using the Azure SignalR emulator, you cannot use the `AddNamedAzureSignalR` method.

Configure Azure SignalR Service in Serverless mode

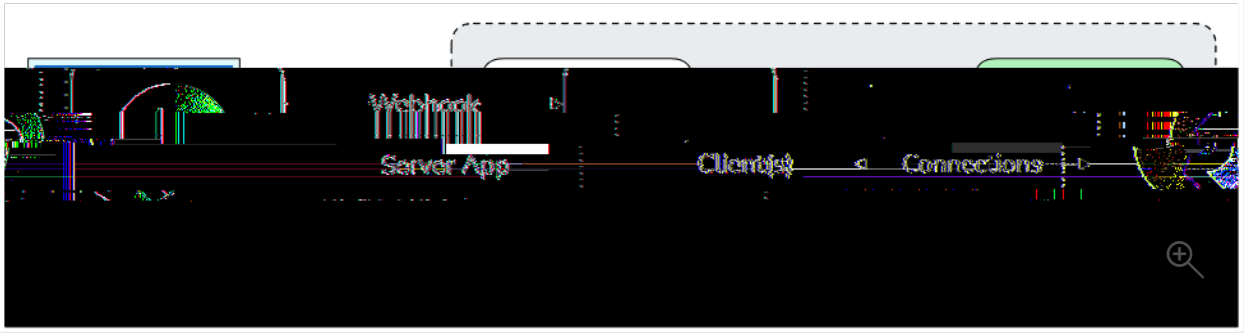
If you're app host is using the Azure SignalR emulator, you'll also need to install the Microsoft.Azure.SignalR.Management NuGet package.

.NET CLI

.NET CLI

```
dotnet add package Microsoft.Azure.SignalR.Management
```

Azure SignalR *Serverless* mode doesn't require a hub server to be running. The Azure SignalR Service is responsible for maintaining client connections. Additionally, in this mode, you cannot use traditional SignalR Hubs, such as Hub, Hub<T>, or IHubContext<THub>. Instead, configure an upstream endpoint which is usually an Azure Function SignalR trigger. Consider the following diagram that illustrates the architecture of Azure SignalR Service in *Serverless* mode:



For more information on *Serverless* mode, see Azure SignalR Service: Serverless mode.

In a project that's intended to communicate with the Azure SignalR Service, register the appropriate services by calling AddSignalR and then registering the `ServiceManager` using the `signalr` connection string and add a `/negotiate` endpoint:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton(sp =>
{
   return new ServiceManagerBuilder()
        .WithOptions(options =>
        {
            options.ConnectionString =
builder.Configuration.GetConnectionString("signalr");
        })
        .BuildServiceManager();
});
```

```
var app = builder.Build();

app.MapPost("/negotiate", async (string? userId, ServiceManager sm,
CancellationToken token) =>
{
    // The creation of the ServiceHubContext is expensive, so it's
recommended to
    // only create it once per named context / per app run if possible.
    var context = await sm.CreateHubContextAsync("messages", token);

    var negotiateResponse = await context.NegotiateAsync(new
NegotiationOptions
    {
        UserId = userId
    }, token);

    // The JSON serializer options need to be set to ignore null values,
otherwise the
    // response will contain null values for the properties that are not
set.
    // The .NET SignalR client will not be able to parse the response if the
null values are present.
    // For more information, see
https://github.com/dotnet/aspnetcore/issues/60935.
    return Results.Json(negotiateResponse, new
JsonSerializerOptions(JsonSerializerDefaults.Web)
    {
        DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull
    });
});

app.Run();
```

The preceding code configures the Azure SignalR Service using the `ServiceManagerBuilder` class, but doesn't call `AddSignalR` or `MapHub`. These two extensions aren't required with *Serverless* mode. The connection string is read from the configuration key `ConnectionStrings:signalr`. When using the emulator, only the HTTP endpoint is available. Within the app, you can use the `ServiceManager` instance to create a `ServiceHubContext`. The `ServiceHubContext` is used to broadcast messages and manage connections to clients.

The `/negotiate` endpoint is required to establish a connection between the connecting client and the Azure SignalR Service. The `ServiceHubContext` is created using the `ServiceManager.CreateHubContextAsync` method, which takes the hub name as a parameter. The `NegotiateAsync` method is called to negotiate the connection with the Azure SignalR Service, which returns an access token and the URL for the client to connect to.

For more information, see Use Azure SignalR Management SDK.

See also

- Azure SignalR Service overview
- Scale ASP.NET Core SignalR applications with Azure SignalR Service
- .NET Aspire Azure integrations overview
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire Azure Service Bus integration

Article • 02/25/2025

Includes: ✅ Hosting integration and ✅ Client integration

Azure Service Bus is a fully managed enterprise message broker with message queues and publish-subscribe topics. The .NET Aspire Azure Service Bus integration enables you to connect to Azure Service Bus instances from .NET applications.

Hosting integration

The .NET Aspire Azure Service Bus hosting integration models the various Service Bus resources as the following types:

- AzureServiceBusResource: Represents an Azure Service Bus resource.
- AzureServiceBusEmulatorResource: Represents an Azure Service Bus emulator resource.

To access these types and APIs for expressing them, add the 📦 Aspire.Hosting.Azure.ServiceBus NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.ServiceBus
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Azure Service Bus resource

In your app host project, call AddAzureServiceBus to add and return an Azure Service Bus resource builder.

C#

```
var builder = DistributedApplication.CreateBuilder(args);
```

```
var serviceBus = builder.AddAzureServiceBus("messaging");

// After adding all resources, run the app...
```

When you add an AzureServiceBusResource to the app host, it exposes other useful APIs to add queues and topics. In other words, you must add an `AzureServiceBusResource` before adding any of the other Service Bus resources.

> **ⓘ Important**
>
> When you call AddAzureServiceBus, it implicitly calls AddAzureProvisioning—which adds support for generating Azure resources dynamically during app startup. The app must configure the appropriate subscription and location. For more information, see Configuration.

Generated provisioning Bicep

If you're new to Bicep, it's a domain-specific language for defining Azure resources. With .NET Aspire, you don't need to write Bicep by-hand, instead the provisioning APIs generate Bicep for you. When you publish your app, the generated Bicep is output alongside the manifest file. When you add an Azure Service Bus resource, the following Bicep is generated:

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location

param sku string = 'Standard'

param principalType string

param principalId string

resource
```

```
}

resource service_bus_AzureServiceBusDataOwner
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(service_bus.          us         u    r  (
```

The preceding Bicep is a module that provisions an Azure Service Bus namespace with the following defaults:

- `sku`: The SKU of the Service Bus namespace. The default is Standard.
- `location`: The location for the Service Bus namespace. The default is the resource group's location.

In addition to the Service Bus namespace, it also provisions an Azure role-based access control (Azure RBAC) built-in role of Azure Service Bus Data Owner. The role is assigned to the Service Bus namespace's resource group. For more information, see Azure Service Bus Data Owner.

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This type enables the customization of the generated Bicep by providing a fluent API to configure the Azure resources—using the ConfigureInfrastructure<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API. For example, you can configure the sku, location, and more. The following example demonstrates how to customize the Azure Service Bus resource:

C#

```
                                        .Single();

        serviceBusNamespace.Sku = new ServiceBusSku
        {
            Tier = ServiceBusSkuTier.Premium
        };
        serviceBusNamespace.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
  - The infra parameter is an instance of the AzureResourceInfrastructure type.
  - The provisionable resources are retrieved by calling the GetProvisionableResources() method.
  - The single ServiceBusNamespace is retrieved.
  - The ServiceBusNamespace.Sku created with a ServiceBusSkuTier.Premium
  - A tag is added to the Service Bus namespace with a key of `ExampleKey` and a value of `Example value`.

There are many more configuration options available to customize the Azure Service Bus resource. For more information, see Azure.Provisioning.ServiceBus. For more information, see Azure.Provisioning customization.

Connect to an existing Azure Service Bus namespace

You might have an existing Azure Service Bus namespace that you want to connect to. Instead of representing a new Azure Service Bus resource, you can add a connection string to the app host. To add a connection to an existing Azure Service Bus namespace, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var serviceBus = builder.AddConnectionString("messaging");

builder.AddProject<Projects.WebApplication>("web")
       .WithReference(serviceBus);

// After adding all resources, run the app...
```

> ⓘ Note

Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section. The app host injects this connection string as an environment variable into all dependent resources, for example:

JSON

```
{
    "ConnectionStrings": {
        "messaging":
"Endpoint=sb://{namespace}.servicebus.windows.net/;SharedAccessKeyName=
{key_name};SharedAccessKey={key_value};"
    }
}
```

The dependent resource can access the injected connection string by calling the GetConnectionString method, and passing the connection name as the parameter, in this case `"messaging"`. The `GetConnectionString` API is shorthand for `IConfiguration.GetSection("ConnectionStrings")[name]`.

Add Azure Service Bus queue

To add an Azure Service Bus queue, call the AddServiceBusQueue method on the `IResourceBuilder<AzureServiceBusResource>`:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var serviceBus = builder.AddAzureServiceBus("messaging");
serviceBus.AddServiceBusQueue("queue");

// After adding all resources, run the app...
```

When you call AddServiceBusQueue(IResourceBuilder<AzureServiceBusResource>, String, String), it configures your Service Bus resources to have a queue named `queue`. The queue is created in the Service Bus namespace that's represented by the `AzureServiceBusResource` that you added earlier. For more information, see Queues, topics, and subscriptions in Azure Service Bus.

Add Azure Service Bus topic and subscription

To add an Azure Service Bus topic, call the AddServiceBusTopic method on the `IResourceBuilder<AzureServiceBusResource>`:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var serviceBus = builder.AddAzureServiceBus("messaging");
serviceBus.AddServiceBusTopic("topic");

// After adding all resources, run the app...
```

When you call AddServiceBusTopic(IResourceBuilder<AzureServiceBusResource>, String, String), it configures your Service Bus resources to have a topic named `topic`. The topic is created in the Service Bus namespace that's represented by the `AzureServiceBusResource` that you added earlier.

To add a subscription for the topic, call the AddServiceBusSubscription method on the `IResourceBuilder<AzureServiceBusTopicResource>` and configure it using the WithProperties method:

C#

```
using Aspire.Hosting.Azure;

var builder = DistributedApplication.CreateBuilder(args);

var serviceBus = builder.AddAzureServiceBus("messaging");
var topic = serviceBus.AddServiceBusTopic("topic");
topic.AddServiceBusSubscription("sub1")
     .WithProperties(subscription =>
     {
         subscription.MaxDeliveryCount = 10;
         subscription.Rules.Add(
             new AzureServiceBusRule("app-prop-filter-1")
             {
                 CorrelationFilter = new()
                 {
                     ContentType = "application/text",
                     CorrelationId = "id1",
                     Subject = "subject1",
                     MessageId = "msgid1",
                     ReplyTo = "someQueue",
                     ReplyToSessionId = "sessionId",
                     SessionId = "session1",
                     SendTo = "xyz"
                 }
```

```
            });
    });

// After adding all resources, run the app...
```

The preceding code not only adds a topic and creates and configures a subscription named `sub1` for the topic. The subscription has a maximum delivery count of `10` and a rule named `app-prop-filter-1`. The rule is a correlation filter that filters messages based on the `ContentType`, `CorrelationId`, `Subject`, `MessageId`, `ReplyTo`, `ReplyToSessionId`, `SessionId`, and `SendTo` properties.

For more information, see Queues, topics, and subscriptions in Azure Service Bus.

Add Azure Service Bus emulator resource

To add an Azure Service Bus emulator resource, chain a call on an `<IResourceBuilder<AzureServiceBusResource>>` to the RunAsEmulator API:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var serviceBus = builder.AddAzureServiceBus("messaging")
                        .RunAsEmulator();

// After adding all resources, run the app...
```

When you call `RunAsEmulator`, it configures your Service Bus resources to run locally using an emulator. The emulator in this case is the Azure Service Bus Emulator. The Azure Service Bus Emulator provides a free local environment for testing your Azure Service Bus apps and it's a perfect companion to the .NET Aspire Azure hosting integration. The emulator isn't installed, instead, it's accessible to .NET Aspire as a container. When you add a container to the app host, as shown in the preceding example with the `mcr.microsoft.com/azure-messaging/servicebus-emulator` image (and the companion `mcr.microsoft.com/azure-sql-edge` image), it creates and starts the container when the app host starts. For more information, see Container resource lifecycle.

Configure Service Bus emulator container

There are various configurations available for container resources, for example, you can configure the container's ports or providing a wholistic JSON configuration which overrides everything.

Configure Service Bus emulator container host port

By default, the Service Bus emulator container when configured by .NET Aspire, exposes the following endpoints:

Expand table

Endpoint	Image	Container port	Host port
`emulator`	`mcr.microsoft.com/azure-messaging/servicebus-emulator`	5672	dynamic
`tcp`	`mcr.microsoft.com/azure-sql-edge`	1433	dynamic

The port that it's listening on is dynamic by default. When the container starts, the port is mapped to a random port on the host machine. To configure the endpoint port, chain calls on the container resource builder provided by the `RunAsEmulator` method and then the WithHostPort(IResourceBuilder<AzureServiceBusEmulatorResource>, Nullable<Int32>) as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var serviceBus = builder.AddAzureServiceBus("messaging").RunAsEmulator(
                         emulator =>
                         {
                             emulator.WithHostPort(7777);
                         });

// After adding all resources, run the app...
```

The preceding code configures the Service Bus emulator container's existing `emulator` endpoint to listen on port `7777`. The Service Bus emulator container's port is mapped to the host port as shown in the following table:

Expand table

Endpoint name	Port mapping (`container:host`)
`emulator`	`5672:7777`

Configure Service Bus emulator container JSON configuration

The Service Bus emulator automatically generates a configration similar to this *config.json* file from the configured resources. You can override this generated file entirely, or update the JSON configuration with a JsonNode representation of the configuration.

To provide a custom JSON configuration file, call the WithConfigurationFile(IResourceBuilder<AzureServiceBusEmulatorResource>, String) method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var serviceBus = builder.AddAzureServiceBus("messaging").RunAsEmulator(
                         emulator =>
                         {
                             emulator.WithConfigurationFile(
                                 path: "./messaging/custom-config.json");
                         });
```

The preceding code configures the Service Bus emulator container to use a custom JSON configuration file located at `./messaging/custom-config.json`. To instead override specific properties in the default configuration, call the WithConfiguration(IResourceBuilder<AzureServiceBusEmulatorResource>, Action<JsonNode>) method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var serviceBus = builder.AddAzureServiceBus("messaging").RunAsEmulator(
                         emulator =>
                         {
                             emulator.WithConfiguration(
                                 (JsonNode configuration) =>
                                 {
                                     var userConfig =
configuration["UserConfig"];
                                     var ns = userConfig["Namespaces"][0];
                                     var firstQueue = ns["Queues"][0];
                                     var properties =
firstQueue["Properties"];

                                     properties["MaxDeliveryCount"] = 5;

properties["RequiresDuplicateDetection"] = true;
                                     properties["DefaultMessageTimeToLive"]
= "PT2H";
                                 });
```

```
                        });

// After adding all resources, run the app...
```

The preceding code retrieves the `UserConfig` node from the default configuration. It then updates the first queue's properties to set the `MaxDeliveryCount` to `5`, `RequiresDuplicateDetection` to `true`, and `DefaultMessageTimeToLive` to `2 hours`.

Hosting integration health checks

The Azure Service Bus hosting integration automatically adds a health check for the Service Bus resource. The health check verifies that the Service Bus is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.AzureServiceBus NuGet package.

Client integration

To get started with the .NET Aspire Azure Service Bus client integration, install the Aspire.Azure.Messaging.ServiceBus NuGet package in the client-consuming project, that is, the project for the application that uses the Service Bus client. The Service Bus client integration registers a ServiceBusClient instance that you can use to interact with Service Bus.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Azure.Messaging.ServiceBus
```

Add Service Bus client

In the Program.cs file of your client-consuming project, call the AddAzureServiceBusClient extension method on any IHostApplicationBuilder to register a ServiceBusClient for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddAzureServiceBusClient(connectionName: "messaging");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Service Bus resource in the app host project. In other words, when you call `AddAzureServiceBus` and provide a name of `messaging` that same name should be used when calling `AddAzureServiceBusClient`. For more information, see **Add Azure Service Bus resource**.

You can then retrieve the ServiceBusClient instance using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(ServiceBusClient client)
{
    // Use client...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed Service Bus client

There might be situations where you want to register multiple `ServiceBusClient` instances with different connection names. To register keyed Service Bus clients, call the AddKeyedAzureServiceBusClient method:

C#

```
builder.AddKeyedAzureServiceBusClient(name: "mainBus");
builder.AddKeyedAzureServiceBusClient(name: "loggingBus");
```

> **Important**
>
> When using keyed services, it's expected that your Service Bus resource configured two named buses, one for the `mainBus` and one for the `loggingBus`.

Then you can retrieve the `ServiceBusClient` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("mainBus")] ServiceBusClient mainBusClient,
    [FromKeyedServices("loggingBus")] ServiceBusClient loggingBusClient)
{
    // Use clients...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire Azure Service Bus integration provides multiple options to configure the connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling the AddAzureServiceBusClient method:

C#

```
builder.AddAzureServiceBusClient("messaging");
```

Then the connection string is retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "messaging":
"Endpoint=sb://{namespace}.servicebus.windows.net/;SharedAccessKeyName=
{keyName};SharedAccessKey={key};"
  }
}
```

For more information on how to format this connection string, see the ConnectionString documentation.

Use configuration providers

The .NET Aspire Azure Service Bus integration supports Microsoft.Extensions.Configuration. It loads the AzureMessagingServiceBusSettings from configuration by using the `Aspire:Azure:Messaging:ServiceBus` key. The following snippet is an example of a appsettings.json file that configures some of the options:

JSON

```
{
  "Aspire": {
    "Azure": {
      "Messaging": {
        "ServiceBus": {
          "ConnectionString":
"Endpoint=sb://{namespace}.servicebus.windows.net/;SharedAccessKeyName=
{keyName};SharedAccessKey={key};",
          "DisableTracing": false
        }
      }
    }
  }
}
```

For the complete Service Bus client integration JSON schema, see Aspire.Azure.Messaging.ServiceBus/ConfigurationSchema.json.

Use inline delegates

Also you can pass the `Action<AzureMessagingServiceBusSettings> configureSettings` delegate to set up some or all the options inline, for example to disable tracing from code:

C#

```
builder.AddAzureServiceBusClient(
    "messaging",
    static settings => settings.DisableTracing = true);
```

You can also set up the Azure.Messaging.ServiceBus.ServiceBusClientOptions using the optional `Action<ServiceBusClientOptions> configureClientOptions` parameter of the `AddAzureServiceBusClient` method. For example to set the ServiceBusClientOptions.Identifier user-agent header suffix for all requests issues by this client:

C#

```
builder.AddAzureServiceBusClient(
    "messaging",
    configureClientOptions:
        clientOptions => clientOptions.Identifier = "myapp");
```

Client integration health checks

By default, .NET Aspire integrations enable health checks for all services. For more information, see .NET Aspire integrations overview.

The .NET Aspire Azure Service Bus integration:

- Adds the health check when AzureMessagingServiceBusSettings.DisableTracing is `false`, which attempts to connect to the Service Bus.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Azure Service Bus integration uses the following log categories:

- `Azure.Core`
- `Azure.Identity`
- `Azure-Messaging-ServiceBus`

In addition to getting Azure Service Bus request diagnostics for failed requests, you can configure latency thresholds to determine which successful Azure Service Bus request diagnostics will be logged. The default values are 100 ms for point operations and 500 ms for non point operations.

C#

```
builder.AddAzureServiceBusClient(
    "messaging",
    configureClientOptions:
        clientOptions => {
            clientOptions.ServiceBusClientTelemetryOptions = new()
            {
                ServiceBusThresholdOptions = new()
                {
                    PointOperationLatencyThreshold =
TimeSpan.FromMilliseconds(50),
                    NonPointOperationLatencyThreshold =
TimeSpan.FromMilliseconds(300)
                }
            };
        });
```

Tracing

The .NET Aspire Azure Service Bus integration will emit the following tracing activities using OpenTelemetry:

- `Message`
- `ServiceBusSender.Send`
- `ServiceBusSender.Schedule`
- `ServiceBusSender.Cancel`
- `ServiceBusReceiver.Receive`
- `ServiceBusReceiver.ReceiveDeferred`
- `ServiceBusReceiver.Peek`
- `ServiceBusReceiver.Abandon`
- `ServiceBusReceiver.Complete`
- `ServiceBusReceiver.DeadLetter`
- `ServiceBusReceiver.Defer`
- `ServiceBusReceiver.RenewMessageLock`
- `ServiceBusSessionReceiver.RenewSessionLock`
- `ServiceBusSessionReceiver.GetSessionState`
- `ServiceBusSessionReceiver.SetSessionState`
- `ServiceBusProcessor.ProcessMessage`
- `ServiceBusSessionProcessor.ProcessSessionMessage`
- `ServiceBusRuleManager.CreateRule`
- `ServiceBusRuleManager.DeleteRule`
- `ServiceBusRuleManager.GetRules`

Azure Service Bus tracing is currently in preview, so you must set the experimental switch to ensure traces are emitted.

```C#
AppContext.SetSwitch("Azure.Experimental.EnableActivitySource", true);
```

For more information, see Azure Service Bus: Distributed tracing and correlation through Service Bus messaging.

Metrics

The .NET Aspire Azure Service Bus integration currently doesn't support metrics by default due to limitations with the Azure SDK.

See also

- Azure Service Bus
- .NET Aspire integrations overview
- .NET Aspire Azure integrations overview
- .NET Aspire GitHub repo

.NET Aspire Azure Blob Storage integration

Article • 12/11/2024

Includes: ✅ Hosting integration and ✅ Client integration

Azure Blob Storage is a service for storing large amounts of unstructured data. The .NET Aspire Azure Blob Storage integration enables you to connect to existing Azure Blob Storage instances or create new instances from .NET applications.

Hosting integration

The .NET Aspire Azure Storage hosting integration models the various storage resources as the following types:

- AzureStorageResource: Represents an Azure Storage resource.
- AzureStorageEmulatorResource: Represents an Azure Storage emulator resource (Azurite).
- AzureBlobStorageResource: Represents an Azure Blob storage resource.
- AzureQueueStorageResource: Represents an Azure Queue storage resource.
- AzureTableStorageResource: Represents an Azure Table storage resource.

To access these types and APIs for expressing them, add the 📦 Aspire.Hosting.Azure.Storage NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Storage
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Azure Storage resource

In your app host project, call AddAzureStorage to add and return an Azure Storage resource builder.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage");

// An Azure Storage resource is required to add any of the following:
//
// - Azure Blob storage resource.
// - Azure Queue storage resource.
// - Azure Table storage resource.

// After adding all resources, run the app...
```

When you add an `AzureStorageResource` to the app host, it exposes other useful APIs to add Azure Blob, Queue, and Table storage resources. In other words, you must add an `AzureStorageResource` before adding any of the other storage resources.

> **ⓘ Important**
>
> When you call AddAzureStorage, it implicitly calls AddAzureProvisioning—which adds support for generating Azure resources dynamically during app startup. The app must configure the appropriate subscription and location. For more information, see Local provisioning: Configuration.

Generated provisioning Bicep

If you're new to Bicep, it's a domain-specific language for defining Azure resources. With .NET Aspire, you don't need to write Bicep by-hand, instead the provisioning APIs generate Bicep for you. When you publish your app, the generated Bicep is output alongside the manifest file. When you add an Azure Storage resource, the following Bicep is generated:

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location

param principalType string

param principalId string

resource storage 'Microsoft.Storage/storageAccounts@2024-01-01' = {
  name: take('storage${uniqueString(resourceGroup().id)}', 24)
  kind: 'StorageV2'
  location: location
```

```
  sku: {
    name: 'Standard_GRS'
  }
  properties: {
    accessTier: 'Hot'
    allowSharedKeyAccess: false
    minimumTlsVersion: 'TLS1_2'
    networkAcls: {
      defaultAction: 'Allow'
    }
  }
  tags: {
    'aspire-resource-name': 'storage'
  }
}

resource blobs 'Microsoft.Storage/storageAccounts/blobServices@2024-01-01' =
{
  name: 'default'
  parent: storage
}

resource storage_StorageBlobDataContributor
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(storage.id, principalId,
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', 'ba92f5b4-
2d11-453d-a403-e96b0029c9fe'))
  properties: {
    principalId: principalId
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', 'ba92f5b4-
2d11-453d-a403-e96b0029c9fe')
    principalType: principalType
  }
  scope: storage
}

resource storage_StorageTableDataContributor
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(storage.id, principalId,
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '0a9a7e1f-
b9d0-4cc4-a60d-0319b160aaa3'))
  properties: {
    principalId: principalId
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '0a9a7e1f-
b9d0-4cc4-a60d-0319b160aaa3')
    principalType: principalType
  }
  scope: storage
}

resource storage_StorageQueueDataContributor
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(storage.id, principalId,
```

```
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '974c5e8b-
45b9-4653-ba55-5f855dd0fb88'))
  properties: {
    principalId: principalId
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '974c5e8b-
45b9-4653-ba55-5f855dd0fb88')
    principalType: principalType
  }
  scope: storage
}

output blobEndpoint string = storage.properties.primaryEndpoints.blob

output queueEndpoint string = storage.properties.primaryEndpoints.queue

output tableEndpoint string = storage.properties.primaryEndpoints.table
```

The preceding Bicep is a module that provisions an Azure Storage account with the following defaults:

- `kind`: The kind of storage account. The default is `StorageV2`.
- `sku`: The SKU of the storage account. The default is `Standard_GRS`.
- `properties`: The properties of the storage account:
  - `accessTier`: The access tier of the storage account. The default is `Hot`.
  - `allowSharedKeyAccess`: A boolean value that indicates whether the storage account permits requests to be authorized with the account access key. The default is `false`.
  - `minimumTlsVersion`: The minimum supported TLS version for the storage account. The default is `TLS1_2`.
  - `networkAcls`: The network ACLs for the storage account. The default is `{ defaultAction: 'Allow' }`.

In addition to the storage account, it also provisions a blob container.

The following role assignments are added to the storage account to grant your application access. See the built-in Azure role-based access control (Azure RBAC) roles for more information:

Expand table

Role / ID	Description
Storage Blob Data Contributor `ba92f5b4-2d11-453d-a403-e96b0029c9fe`	Read, write, and delete Azure Storage containers and blobs.

Role / ID	Description
Storage Table Data Contributor `0a9a7e1f-b9d0-4cc4-a60d-0319b160aaa3`	Read, write, and delete Azure Storage tables and entities.
Storage Queue Data Contributor `974c5e8b-45b9-4653-ba55-5f855dd0fb88`	Read, write, and delete Azure Storage queues and queue messages.

The generated Bicep is a starting point and is influenced by changes to the provisioning infrastructure in C#. Customizations to the Bicep file directly will be overwritten, so make changes through the C# provisioning APIs to ensure they are reflected in the generated files.

Customize provisioning infrastructure

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This type enables the customization of the generated Bicep by providing a fluent API to configure the Azure resources—using the ConfigureInfrastructure<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API. For example, you can configure the `kind`, `sku`, `properties`, and more. The following example demonstrates how to customize the Azure Storage resource:

C#

```
builder.AddAzureStorage("storage")
    .ConfigureInfrastructure(infra =>
    {
        var storageAccount = infra.GetProvisionableResources()
                                  .OfType<StorageAccount>()
                                  .Single();

        storageAccount.AccessTier = StorageAccountAccessTier.Cool;
        storageAccount.Sku = new StorageSku { Name =
StorageSkuName.PremiumZrs };
        storageAccount.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
  - The `infra` parameter is an instance of the AzureResourceInfrastructure type.
  - The provisionable resources are retrieved by calling the GetProvisionableResources() method.

- The single StorageAccount is retrieved.
- The StorageAccount.AccessTier is assigned to StorageAccountAccessTier.Cool.
- The StorageAccount.Sku is assigned to a new StorageSku with a `Name` of PremiumZrs.
- A tag is added to the storage account with a key of `ExampleKey` and a value of `Example value`.

There are many more configuration options available to customize the Azure Storage resource. For more information, see Azure.Provisioning.Storage.

Connect to an existing Azure Storage account

You might have an existing Azure Storage account that you want to connect to. Instead of representing a new Azure Storage resource, you can add a connection string to the app host. To add a connection to an existing Azure Storage account, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var blobs = builder.AddConnectionString("blobs");

builder.AddProject<Projects.WebApplication>("web")
       .WithReference(blobs);

// After adding all resources, run the app...
```

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section. The app host injects this connection string as an environment variable into all dependent resources, for example:

JSON

```
{
    "ConnectionStrings": {
```

```
        "blobs": "https://{account_name}.blob.core.windows.net/"
    }
}
```

The dependent resource can access the injected connection string by calling the GetConnectionString method, and passing the connection name as the parameter, in this case `"blobs"`. The `GetConnectionString` API is shorthand for `IConfiguration.GetSection("ConnectionStrings")[name]`.

Add Azure Storage emulator resource

To add an Azure Storage emulator resource, chain a call on an `IResourceBuilder<AzureStorageResource>` to the RunAsEmulator API:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage")
                     .RunAsEmulator();

// After adding all resources, run the app...
```

When you call `RunAsEmulator`, it configures your storage resources to run locally using an emulator. The emulator in this case is Azurite. The Azurite open-source emulator provides a free local environment for testing your Azure Blob, Queue Storage, and Table Storage apps and it's a perfect companion to the .NET Aspire Azure hosting integration. Azurite isn't installed, instead, it's accessible to .NET Aspire as a container. When you add a container to the app host, as shown in the preceding example with the `mcr.microsoft.com/azure-storage/azurite` image, it creates and starts the container when the app host starts. For more information, see Container resource lifecycle.

Configure Azurite container

There are various configurations available to container resources, for example, you can configure the container's ports, environment variables, it's lifetime, and more.

Configure Azurite container ports

By default, the Azurite container when configured by .NET Aspire, exposes the following endpoints:

Expand table

Endpoint	Container port	Host port
`blob`	10000	dynamic
`queue`	10001	dynamic
`table`	10002	dynamic

The port that they're listening on is dynamic by default. When the container starts, the ports are mapped to a random port on the host machine. To configure the endpoint ports, chain calls on the container resource builder provided by the `RunAsEmulator` method as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage").RunAsEmulator(
                      azurite =>
                      {
                          azurite.WithBlobPort(27000)
                                 .WithQueuePort(27001)
                                 .WithTablePort(27002);
                      });

// After adding all resources, run the app...
```

The preceding code configures the Azurite container's existing `blob`, `queue`, and `table` endpoints to listen on ports `27000`, `27001`, and `27002`, respectively. The Azurite container's ports are mapped to the host ports as shown in the following table:

Expand table

Endpoint name	Port mapping (`container:host`)
`blob`	`10000:27000`
`queue`	`10001:27001`
`table`	`10002:27002`

Configure Azurite container with persistent lifetime

To configure the Azurite container with a persistent lifetime, call the WithLifetime method on the Azurite container resource and pass ContainerLifetime.Persistent:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage").RunAsEmulator(
                      azurite =>
                      {
                          azurite.WithLifetime(ContainerLifetime.Persistent);
                      });

// After adding all resources, run the app...
```

For more information, see Container resource lifetime.

Configure Azurite container with data volume

To add a data volume to the Azure Storage emulator resource, call the WithDataVolume method on the Azure Storage emulator resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage").RunAsEmulator(
                      azurite =>
                      {
                          azurite.WithDataVolume();
                      });

// After adding all resources, run the app...
```

The data volume is used to persist the Azurite data outside the lifecycle of its container. The data volume is mounted at the `/data` path in the Azurite container and when a `name` parameter isn't provided, the name is formatted as `.azurite/{resource name}`. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Configure Azurite container with data bind mount

To add a data bind mount to the Azure Storage emulator resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage").RunAsEmulator(
                      azurite =>
                      {
                          azurite.WithDataBindMount("../Azurite/Data");
                      });

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Azurite data across container restarts. The data bind mount is mounted at the `../Azurite/Data` path on the host machine relative to the app host directory (IDistributedApplicationBuilder.AppHostDirectory) in the Azurite container. For more information on data bind mounts, see Docker docs: Bind mounts.

Connect to storage resources

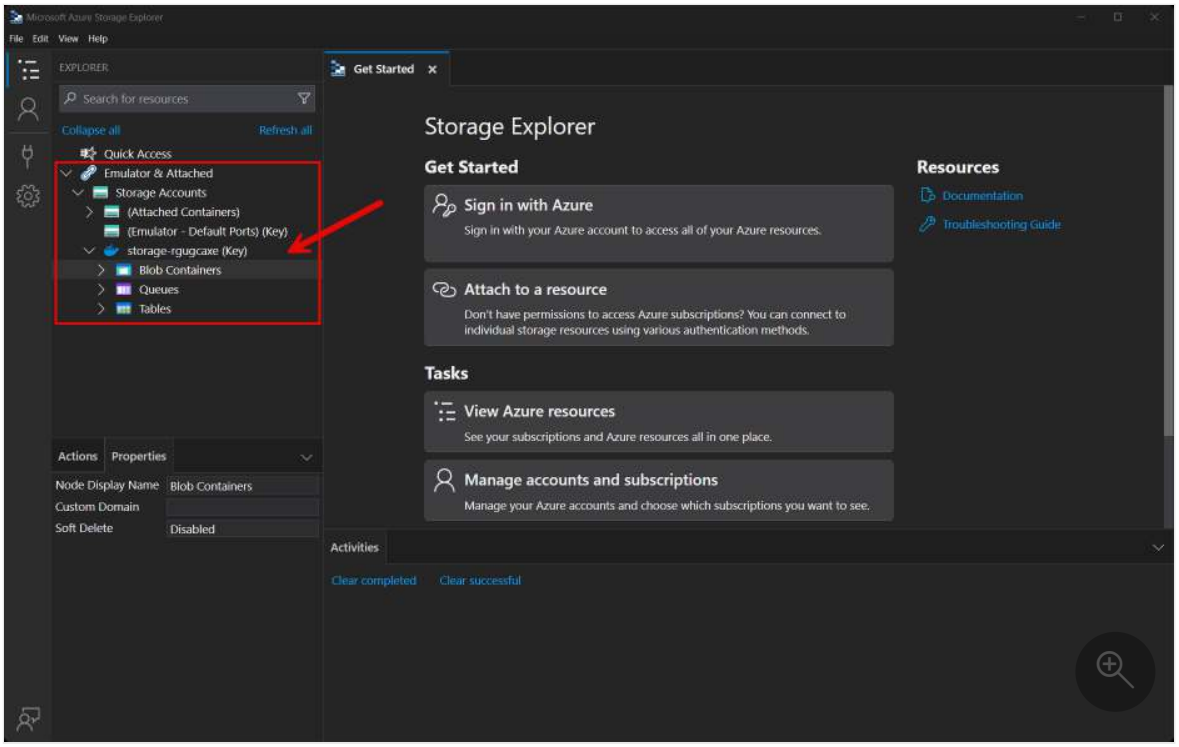
When the .NET Aspire app host runs, the storage resources can be accessed by external tools, such as the Azure Storage Explorer. If your storage resource is running locally using Azurite, it will automatically be picked up by the Azure Storage Explorer.

> **Note**
>
> The Azure Storage Explorer discovers Azurite storage resources assuming the default ports are used. If you've configured the Azurite container to use different ports, you'll need to configure the Azure Storage Explorer to connect to the correct ports.

To connect to the storage resource from Azure Storage Explorer, follow these steps:

1. Run the .NET Aspire app host.
2. Open the Azure Storage Explorer.

3. View the **Explorer** pane.
4. Select the **Refresh all** link to refresh the list of storage accounts.
5. Expand the **Emulator & Attached** node.
6. Expand the **Storage Accounts** node.
7. You should see a storage account with your resource's name as a prefix:



You're free to explore the storage account and its contents using the Azure Storage Explorer. For more information on using the Azure Storage Explorer, see Get started with Storage Explorer.

Add Azure Blob Storage resource

In your app host project, register the Azure Blob Storage integration by chaining a call to AddBlobs on the `IResourceBuilder<IAzureStorageResource>` instance returned by AddAzureStorage. The following example demonstrates how to add an Azure Blob Storage resource named `storage` and a blob container named `blobs`:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var blobs = builder.AddAzureStorage("storage")
                   .RunAsEmulator()
                   .AddBlobs("blobs");
```

```
builder.AddProject<Projects.ExampleProject>()
       .WithReference(blobs)
       .WaitFor(blobs);

// After adding all resources, run the app...
```

The preceding code:

- Adds an Azure Storage resource named `storage`.
- Chains a call to RunAsEmulator to configure the storage resource to run locally using an emulator. The emulator in this case is Azurite.
- Adds a blob container named `blobs` to the storage resource.
- Adds the `blobs` resource to the `ExampleProject` and waits for it to be ready before starting the project.

Hosting integration health checks

The Azure Storage hosting integration automatically adds a health check for the storage resource. It's added only when running as an emulator, and verifies the Azurite container is running and that a connection can be established to it. The hosting integration relies on the AspNetCore.HealthChecks.Azure.Storage.Blobs NuGet package.

Client integration

To get started with the .NET Aspire Azure Blob Storage client integration, install the Aspire.Azure.Storage.Blobs NuGet package in the client-consuming project, that is, the project for the application that uses the Azure Blob Storage client. The Azure Blob Storage client integration registers a BlobServiceClient instance that you can use to interact with Azure Blob Storage.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Azure.Storage.Blobs
```

Add Azure Blob Storage client

In the *Program.cs* file of your client-consuming project, call the AddAzureBlobClient extension method on any IHostApplicationBuilder to register a `BlobServiceClient` for

use via the dependency injection container. The method takes a connection name parameter.

```C#
builder.AddAzureBlobClient("blobs");
```

You can then retrieve the `BlobServiceClient` instance using dependency injection. For example, to retrieve the client from a service:

```C#
public class ExampleService(BlobServiceClient client)
{
    // Use client...
}
```

Configuration

The .NET Aspire Azure Blob Storage integration provides multiple options to configure the `BlobServiceClient` based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling AddAzureBlobClient:

```C#
builder.AddAzureBlobClient("blobs");
```

Then the connection string is retrieved from the `ConnectionStrings` configuration section, and two connection formats are supported:

Service URI

The recommended approach is to use a `ServiceUri`, which works with the AzureStorageBlobsSettings.Credential property to establish a connection. If no credential is configured, the Azure.Identity.DefaultAzureCredential is used.

```JSON
```

```
{
  "ConnectionStrings": {
    "blobs": "https://{account_name}.blob.core.windows.net/"
  }
}
```

Connection string

Alternatively, an Azure Storage connection string can be used.

JSON

```
{
  "ConnectionStrings": {
    "blobs": "AccountName=myaccount;AccountKey=myaccountkey"
  }
}
```

For more information, see Configure Azure Storage connection strings.

Use configuration providers

The .NET Aspire Azure Blob Storage integration supports Microsoft.Extensions.Configuration. It loads the AzureStorageBlobsSettings and BlobClientOptions from configuration by using the `Aspire:Azure:Storage:Blobs` key. The following snippet is an example of a *appsettings.json* file that configures some of the options:

JSON

```
{
  "Aspire": {
    "Azure": {
      "Storage": {
        "Blobs": {
          "DisableHealthChecks": true,
          "DisableTracing": false,
          "ClientOptions": {
            "Diagnostics": {
              "ApplicationId": "myapp"
            }
          }
        }
      }
    }
```

```
  }
}
```

For the complete Azure Blob Storage client integration JSON schema, see Aspire.Azure.Storage.Blobs/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<AzureStorageBlobsSettings> configureSettings` delegate to set up some or all the options inline, for example to configure health checks:

C#

```
builder.AddAzureBlobClient(
    "blobs",
    settings => settings.DisableHealthChecks = true);
```

You can also set up the BlobClientOptions using the `Action<IAzureClientBuilder<BlobServiceClient, BlobClientOptions>> configureClientBuilder` delegate, the second parameter of the `AddAzureBlobClient` method. For example, to set the first part of user-agent headers for all requests issues by this client:

C#

```
builder.AddAzureBlobClient(
    "blobs",
    configureClientBuilder: clientBuilder =>
        clientBuilder.ConfigureOptions(
            options => options.Diagnostics.ApplicationId = "myapp"));
```

Client integration health checks

By default, .NET Aspire integrations enable health checks for all services. For more information, see .NET Aspire integrations overview.

The .NET Aspire Azure Blob Storage integration:

- Adds the health check when AzureStorageBlobsSettings.DisableHealthChecks is `false`, which attempts to connect to the Azure Blob Storage.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Azure Blob Storage integration uses the following log categories:

- `Azure.Core`
- `Azure.Identity`

Tracing

The .NET Aspire Azure Blob Storage integration emits the following tracing activities using OpenTelemetry:

- `Azure.Storage.Blobs.BlobContainerClient`

Metrics

The .NET Aspire Azure Blob Storage integration currently doesn't support metrics by default due to limitations with the Azure SDK.

See also

- Azure Blob Storage docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire Azure Queue Storage integration

Article • 12/11/2024

Includes: ✅ Hosting integration and ✅ Client integration

Azure Queue Storage is a service for storing large numbers of messages that can be accessed from anywhere in the world via authenticated calls. The .NET Aspire Azure Queue Storage integration enables you to connect to existing Azure Queue Storage instances or create new instances from .NET applications.

Hosting integration

The .NET Aspire Azure Storage hosting integration models the various storage resources as the following types:

- AzureStorageResource: Represents an Azure Storage resource.
- AzureStorageEmulatorResource: Represents an Azure Storage emulator resource (Azurite).
- AzureBlobStorageResource: Represents an Azure Blob storage resource.
- AzureQueueStorageResource: Represents an Azure Queue storage resource.
- AzureTableStorageResource: Represents an Azure Table storage resource.

To access these types and APIs for expressing them, add the 📦 Aspire.Hosting.Azure.Storage NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Storage
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Azure Storage resource

In your app host project, call AddAzureStorage to add and return an Azure Storage resource builder.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage");

// An Azure Storage resource is required to add any of the following:
//
// - Azure Blob storage resource.
// - Azure Queue storage resource.
// - Azure Table storage resource.

// After adding all resources, run the app...
```

When you add an `AzureStorageResource` to the app host, it exposes other useful APIs to add Azure Blob, Queue, and Table storage resources. In other words, you must add an `AzureStorageResource` before adding any of the other storage resources.

> **ⓘ Important**
>
> When you call AddAzureStorage, it implicitly calls AddAzureProvisioning—which adds support for generating Azure resources dynamically during app startup. The app must configure the appropriate subscription and location. For more information, see Local provisioning: Configuration.

Generated provisioning Bicep

If you're new to Bicep, it's a domain-specific language for defining Azure resources. With .NET Aspire, you don't need to write Bicep by-hand, instead the provisioning APIs generate Bicep for you. When you publish your app, the generated Bicep is output alongside the manifest file. When you add an Azure Storage resource, the following Bicep is generated:

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location

param principalType string

param principalId string

resource storage 'Microsoft.Storage/storageAccounts@2024-01-01' = {
  name: take('storage${uniqueString(resourceGroup().id)}', 24)
  kind: 'StorageV2'
  location: location
```

```
  sku: {
    name: 'Standard_GRS'
  }
  properties: {
    accessTier: 'Hot'
    allowSharedKeyAccess: false
    minimumTlsVersion: 'TLS1_2'
    networkAcls: {
      defaultAction: 'Allow'
    }
  }
  tags: {
    'aspire-resource-name': 'storage'
  }
}

resource blobs 'Microsoft.Storage/storageAccounts/blobServices@2024-01-01' =
{
  name: 'default'
  parent: storage
}

resource storage_StorageBlobDataContributor
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(storage.id, principalId,
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', 'ba92f5b4-
2d11-453d-a403-e96b0029c9fe'))
  properties: {
    principalId: principalId
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', 'ba92f5b4-
2d11-453d-a403-e96b0029c9fe')
    principalType: principalType
  }
  scope: storage
}

resource storage_StorageTableDataContributor
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(storage.id, principalId,
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '0a9a7e1f-
b9d0-4cc4-a60d-0319b160aaa3'))
  properties: {
    principalId: principalId
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '0a9a7e1f-
b9d0-4cc4-a60d-0319b160aaa3')
    principalType: principalType
  }
  scope: storage
}

resource storage_StorageQueueDataContributor
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(storage.id, principalId,
```

```
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '974c5e8b-
45b9-4653-ba55-5f855dd0fb88'))
  properties: {
    principalId: principalId
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '974c5e8b-
45b9-4653-ba55-5f855dd0fb88')
    principalType: principalType
  }
  scope: storage
}

output blobEndpoint string = storage.properties.primaryEndpoints.blob

output queueEndpoint string = storage.properties.primaryEndpoints.queue

output tableEndpoint string = storage.properties.primaryEndpoints.table
```

The preceding Bicep is a module that provisions an Azure Storage account with the following defaults:

- `kind`: The kind of storage account. The default is `StorageV2`.
- `sku`: The SKU of the storage account. The default is `Standard_GRS`.
- `properties`: The properties of the storage account:
  - `accessTier`: The access tier of the storage account. The default is `Hot`.
  - `allowSharedKeyAccess`: A boolean value that indicates whether the storage account permits requests to be authorized with the account access key. The default is `false`.
  - `minimumTlsVersion`: The minimum supported TLS version for the storage account. The default is `TLS1_2`.
  - `networkAcls`: The network ACLs for the storage account. The default is `{ defaultAction: 'Allow' }`.

In addition to the storage account, it also provisions a blob container.

The following role assignments are added to the storage account to grant your application access. See the built-in Azure role-based access control (Azure RBAC) roles for more information:

Expand table

Role / ID	Description
Storage Blob Data Contributor `ba92f5b4-2d11-453d-a403-e96b0029c9fe`	Read, write, and delete Azure Storage containers and blobs.

Role / ID	Description
Storage Table Data Contributor `0a9a7e1f-b9d0-4cc4-a60d-0319b160aaa3`	Read, write, and delete Azure Storage tables and entities.
Storage Queue Data Contributor `974c5e8b-45b9-4653-ba55-5f855dd0fb88`	Read, write, and delete Azure Storage queues and queue messages.

The generated Bicep is a starting point and is influenced by changes to the provisioning infrastructure in C#. Customizations to the Bicep file directly will be overwritten, so make changes through the C# provisioning APIs to ensure they are reflected in the generated files.

Customize provisioning infrastructure

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This type enables the customization of the generated Bicep by providing a fluent API to configure the Azure resources—using the ConfigureInfrastructure<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API. For example, you can configure the `kind`, `sku`, `properties`, and more. The following example demonstrates how to customize the Azure Storage resource:

C#

```
builder.AddAzureStorage("storage")
    .ConfigureInfrastructure(infra =>
    {
        var storageAccount = infra.GetProvisionableResources()
                                  .OfType<StorageAccount>()
                                  .Single();

        storageAccount.AccessTier = StorageAccountAccessTier.Cool;
        storageAccount.Sku = new StorageSku { Name =
StorageSkuName.PremiumZrs };
        storageAccount.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
  - The `infra` parameter is an instance of the AzureResourceInfrastructure type.
  - The provisionable resources are retrieved by calling the GetProvisionableResources() method.

- The single StorageAccount is retrieved.
- The StorageAccount.AccessTier is assigned to StorageAccountAccessTier.Cool.
- The StorageAccount.Sku is assigned to a new StorageSku with a `Name` of PremiumZrs.
- A tag is added to the storage account with a key of `ExampleKey` and a value of `Example value`.

There are many more configuration options available to customize the Azure Storage resource. For more information, see Azure.Provisioning.Storage.

Connect to an existing Azure Storage account

You might have an existing Azure Storage account that you want to connect to. Instead of representing a new Azure Storage resource, you can add a connection string to the app host. To add a connection to an existing Azure Storage account, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var blobs = builder.AddConnectionString("blobs");

builder.AddProject<Projects.WebApplication>("web")
       .WithReference(blobs);

// After adding all resources, run the app...
```

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section. The app host injects this connection string as an environment variable into all dependent resources, for example:

JSON

```
{
    "ConnectionStrings": {
```

```
        "blobs": "https://{account_name}.blob.core.windows.net/"
    }
}
```

The dependent resource can access the injected connection string by calling the GetConnectionString method, and passing the connection name as the parameter, in this case `"blobs"`. The `GetConnectionString` API is shorthand for `IConfiguration.GetSection("ConnectionStrings")[name]`.

Add Azure Storage emulator resource

To add an Azure Storage emulator resource, chain a call on an `IResourceBuilder<AzureStorageResource>` to the RunAsEmulator API:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage")
                     .RunAsEmulator();

// After adding all resources, run the app...
```

When you call `RunAsEmulator`, it configures your storage resources to run locally using an emulator. The emulator in this case is Azurite. The Azurite open-source emulator provides a free local environment for testing your Azure Blob, Queue Storage, and Table Storage apps and it's a perfect companion to the .NET Aspire Azure hosting integration. Azurite isn't installed, instead, it's accessible to .NET Aspire as a container. When you add a container to the app host, as shown in the preceding example with the `mcr.microsoft.com/azure-storage/azurite` image, it creates and starts the container when the app host starts. For more information, see Container resource lifecycle.

Configure Azurite container

There are various configurations available to container resources, for example, you can configure the container's ports, environment variables, it's lifetime, and more.

Configure Azurite container ports

By default, the Azurite container when configured by .NET Aspire, exposes the following endpoints:

Expand table

Endpoint	Container port	Host port
`blob`	10000	dynamic
`queue`	10001	dynamic
`table`	10002	dynamic

The port that they're listening on is dynamic by default. When the container starts, the ports are mapped to a random port on the host machine. To configure the endpoint ports, chain calls on the container resource builder provided by the `RunAsEmulator` method as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage").RunAsEmulator(
                      azurite =>
                      {
                          azurite.WithBlobPort(27000)
                                 .WithQueuePort(27001)
                                 .WithTablePort(27002);
                      });

// After adding all resources, run the app...
```

The preceding code configures the Azurite container's existing `blob`, `queue`, and `table` endpoints to listen on ports `27000`, `27001`, and `27002`, respectively. The Azurite container's ports are mapped to the host ports as shown in the following table:

Expand table

Endpoint name	Port mapping (`container:host`)
`blob`	`10000:27000`
`queue`	`10001:27001`
`table`	`10002:27002`

Configure Azurite container with persistent lifetime

To configure the Azurite container with a persistent lifetime, call the WithLifetime method on the Azurite container resource and pass ContainerLifetime.Persistent:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage").RunAsEmulator(
                      azurite =>
                      {
                          azurite.WithLifetime(ContainerLifetime.Persistent);
                      });

// After adding all resources, run the app...
```

For more information, see Container resource lifetime.

Configure Azurite container with data volume

To add a data volume to the Azure Storage emulator resource, call the WithDataVolume method on the Azure Storage emulator resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage").RunAsEmulator(
                      azurite =>
                      {
                          azurite.WithDataVolume();
                      });

// After adding all resources, run the app...
```

The data volume is used to persist the Azurite data outside the lifecycle of its container. The data volume is mounted at the `/data` path in the Azurite container and when a `name` parameter isn't provided, the name is formatted as `.azurite/{resource name}`. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Configure Azurite container with data bind mount

To add a data bind mount to the Azure Storage emulator resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage").RunAsEmulator(
                      azurite =>
                      {
                          azurite.WithDataBindMount("../Azurite/Data");
                      });

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Azurite data across container restarts. The data bind mount is mounted at the `../Azurite/Data` path on the host machine relative to the app host directory (IDistributedApplicationBuilder.AppHostDirectory) in the Azurite container. For more information on data bind mounts, see Docker docs: Bind mounts.

Connect to storage resources

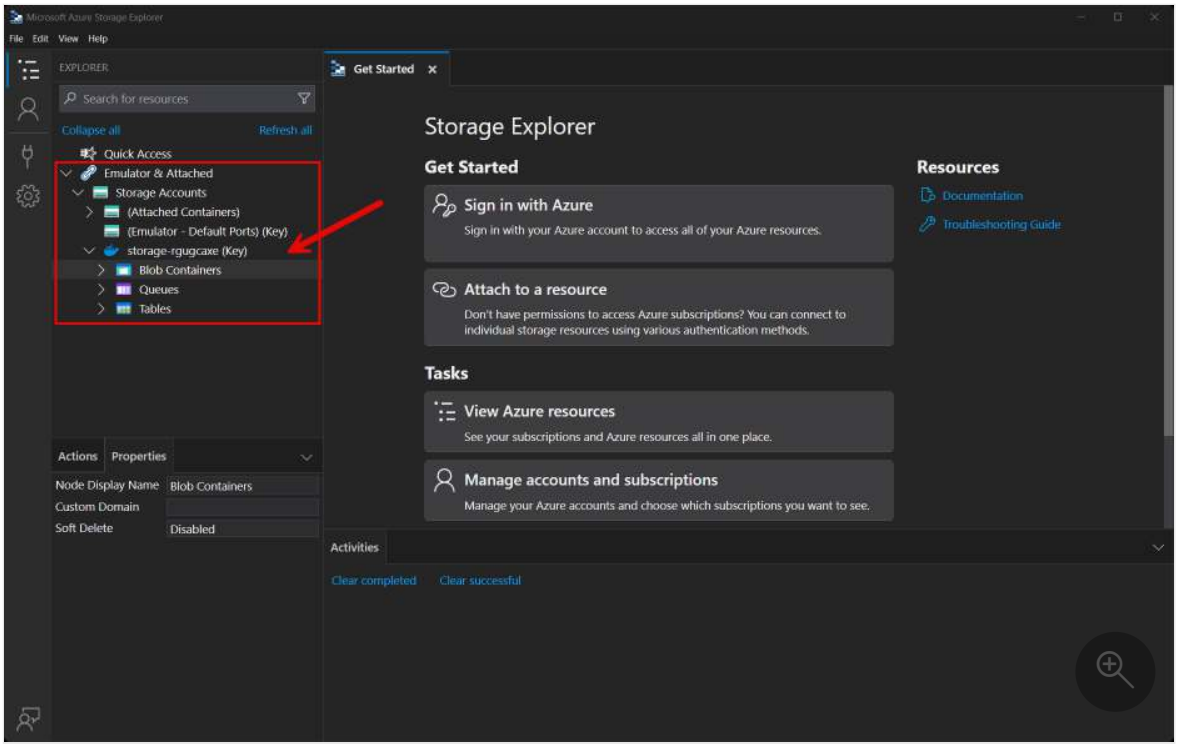
When the .NET Aspire app host runs, the storage resources can be accessed by external tools, such as the Azure Storage Explorer. If your storage resource is running locally using Azurite, it will automatically be picked up by the Azure Storage Explorer.

> **Note**
>
> The Azure Storage Explorer discovers Azurite storage resources assuming the default ports are used. If you've configured the Azurite container to use different ports, you'll need to configure the Azure Storage Explorer to connect to the correct ports.

To connect to the storage resource from Azure Storage Explorer, follow these steps:

1. Run the .NET Aspire app host.
2. Open the Azure Storage Explorer.

3. View the **Explorer** pane.
4. Select the **Refresh all** link to refresh the list of storage accounts.
5. Expand the **Emulator & Attached** node.
6. Expand the **Storage Accounts** node.
7. You should see a storage account with your resource's name as a prefix:



You're free to explore the storage account and its contents using the Azure Storage Explorer. For more information on using the Azure Storage Explorer, see Get started with Storage Explorer.

Add Azure Queue Storage resource

In your app host project, register the Azure Queue Storage integration by chaining a call to AddQueues on the `IResourceBuilder<IAzureStorageResource>` instance returned by AddAzureStorage. The following example demonstrates how to add an Azure Queue Storage resource named `storage` and a queue resource named `queues`:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var queues = builder.AddAzureStorage("storage")
                    .AddQueues("queues");

builder.AddProject<Projects.ExampleProject>()
```

preceding code:

Adds an Azure Storage resource named `storage`.
Adds a queue named `queues` to the storage resource.
Adds the `storage` resource to the

```
builder.AddAzureQueueClient("queue");
```

You can then retrieve the `QueueServiceClient` instance using dependency injection. For example, to retrieve the client from a service:

C#

```
public class ExampleService(QueueServiceClient client)
{
    // Use client...
}
```

Configuration

The .NET Aspire Azure Queue Storage integration provides multiple options to configure the `QueueServiceClient` based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling AddAzureQueueClient:

C#

```
builder.AddAzureQueueClient("queue");
```

Then the connection string is retrieved from the `ConnectionStrings` configuration section, and two connection formats are supported:

Service URI

The recommended approach is to use a `ServiceUri`, which works with the AzureStorageQueuesSettings.Credential property to establish a connection. If no credential is configured, the Azure.Identity.DefaultAzureCredential is used.

JSON

```
{
  "ConnectionStrings": {
    "queue": "https://{account_name}.queue.core.windows.net/"
```

```
  }
}
```

Connection string

Alternatively, an Azure Storage connection string can be used.

JSON

```
{
  "ConnectionStrings": {
    "queue": "AccountName=myaccount;AccountKey=myaccountkey"
  }
}
```

For more information, see Configure Azure Storage connection strings.

Use configuration providers

The .NET Aspire Azure Queue Storage integration supports Microsoft.Extensions.Configuration. It loads the AzureStorageQueuesSettings and QueueClientOptions from configuration by using the `Aspire:Azure:Storage:Queues` key. The following snippet is an example of a *appsettings.json* file that configures some of the options:

JSON

```
{
  "Aspire": {
    "Azure": {
      "Storage": {
        "Queues": {
          "DisableHealthChecks": true,
          "DisableTracing": false,
          "ClientOptions": {
            "Diagnostics": {
              "ApplicationId": "myapp"
            }
          }
        }
      }
    }
  }
}
```

For the complete Azure Storage Queues client integration JSON schema, see Aspire.Azure.Data.Queues/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<AzureStorageQueuesSettings> configureSettings` delegate to set up some or all the options inline, for example to configure health checks:

```
C#

builder.AddAzureQueueClient(
    "queue",
    settings => settings.DisableHealthChecks = true);
```

You can also set up the QueueClientOptions using `Action<IAzureClientBuilder<QueueServiceClient, QueueClientOptions>> configureClientBuilder` delegate, the second parameter of the `AddAzureQueueClient` method. For example, to set the first part of user-agent headers for all requests issues by this client:

```
C#

builder.AddAzureQueueClient(
    "queue",
    configureClientBuilder: clientBuilder =>
        clientBuilder.ConfigureOptions(
            options => options.Diagnostics.ApplicationId = "myapp"));
```

Client integration health checks

By default, .NET Aspire integrations enable health checks for all services. For more information, see .NET Aspire integrations overview.

The .NET Aspire Azure Queue Storage integration:

- Adds the health check when AzureStorageQueuesSettings.DisableHealthChecks is `false`, which attempts to connect to the Azure Queue Storage.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Azure Queue Storage integration uses the following log categories:

- `Azure.Core`
- `Azure.Identity`

Tracing

The .NET Aspire Azure Queue Storage integration emits the following tracing activities using OpenTelemetry:

- `Azure.Storage.Queues.QueueClient`

Metrics

The .NET Aspire Azure Queue Storage integration currently doesn't support metrics by default due to limitations with the Azure SDK.

See also

- Azure Queue Storage docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire Azure Data Tables integration

Article • 12/11/2024

Includes: ✅ Hosting integration and ✅ Client integration

Azure Table Storage is a service for storing structured NoSQL data. The .NET Aspire Azure Data Tables integration enables you to connect to existing Azure Table Storage instances or create new instances from .NET applications.

Hosting integration

The .NET Aspire Azure Storage hosting integration models the various storage resources as the following types:

- AzureStorageResource: Represents an Azure Storage resource.
- AzureStorageEmulatorResource: Represents an Azure Storage emulator resource (Azurite).
- AzureBlobStorageResource: Represents an Azure Blob storage resource.
- AzureQueueStorageResource: Represents an Azure Queue storage resource.
- AzureTableStorageResource: Represents an Azure Table storage resource.

To access these types and APIs for expressing them, add the 📦 Aspire.Hosting.Azure.Storage NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Storage
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Azure Storage resource

In your app host project, call AddAzureStorage to add and return an Azure Storage resource builder.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage");

// An Azure Storage resource is required to add any of the following:
//
// - Azure Blob storage resource.
// - Azure Queue storage resource.
// - Azure Table storage resource.

// After adding all resources, run the app...
```

When you add an `AzureStorageResource` to the app host, it exposes other useful APIs to add Azure Blob, Queue, and Table storage resources. In other words, you must add an `AzureStorageResource` before adding any of the other storage resources.

> **Important**
>
> When you call

If you're new to Bicep, it's a domain-specific language for defining Azure resources. With .NET Aspire, you don't need to write Bicep by-hand, instead the provisioning APIs generate Bicep for you. When you publish your app, the generated Bicep is output alongside the manifest file. When you add an Azure Storage resource, the following Bicep is generated:

Bicep

```
  sku: {
    name: 'Standard_GRS'
  }
  properties: {
    accessTier: 'Hot'
    allowSharedKeyAccess: false
    minimumTlsVersion: 'TLS1_2'
    networkAcls: {
      defaultAction: 'Allow'
    }
  }
  tags: {
    'aspire-resource-name': 'storage'
  }
}

resource blobs 'Microsoft.Storage/storageAccounts/blobServices@2024-01-01' =
{
  name: 'default'
  parent: storage
}

resource storage_StorageBlobDataContributor
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(storage.id, principalId,
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', 'ba92f5b4-
2d11-453d-a403-e96b0029c9fe'))
  properties: {
    principalId: principalId
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', 'ba92f5b4-
2d11-453d-a403-e96b0029c9fe')
    principalType: principalType
  }
  scope: storage
}

resource storage_StorageTableDataContributor
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(storage.id, principalId,
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '0a9a7e1f-
b9d0-4cc4-a60d-0319b160aaa3'))
  properties: {
    principalId: principalId
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '0a9a7e1f-
b9d0-4cc4-a60d-0319b160aaa3')
    principalType: principalType
  }
  scope: storage
}

resource storage_StorageQueueDataContributor
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(storage.id, principalId,
```

```
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '974c5e8b-
45b9-4653-ba55-5f855dd0fb88'))
  properties: {
    principalId: principalId
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '974c5e8b-
45b9-4653-ba55-5f855dd0fb88')
    principalType: principalType
  }
  scope: storage
}

output blobEndpoint string = storage.properties.primaryEndpoints.blob

output queueEndpoint string = storage.properties.primaryEndpoints.queue

output tableEndpoint string = storage.properties.primaryEndpoints.table
```

The preceding Bicep is a module that provisions an Azure Storage account with the following defaults:

- `kind`: The kind of storage account. The default is `StorageV2`.
- `sku`: The SKU of the storage account. The default is `Standard_GRS`.
- `properties`: The properties of the storage account:
  - `accessTier`: The access tier of the storage account. The default is `Hot`.
  - `allowSharedKeyAccess`: A boolean value that indicates whether the storage account permits requests to be authorized with the account access key. The default is `false`.
  - `minimumTlsVersion`: The minimum supported TLS version for the storage account. The default is `TLS1_2`.
  - `networkAcls`: The network ACLs for the storage account. The default is `{ defaultAction: 'Allow' }`.

In addition to the storage account, it also provisions a blob container.

The following role assignments are added to the storage account to grant your application access. See the built-in Azure role-based access control (Azure RBAC) roles for more information:

Expand table

Role / ID	Description
Storage Blob Data Contributor `ba92f5b4-2d11-453d-a403-e96b0029c9fe`	Read, write, and delete Azure Storage containers and blobs.

Role / ID	Description
Storage Table Data Contributor `0a9a7e1f-b9d0-4cc4-a60d-0319b160aaa3`	Read, write, and delete Azure Storage tables and entities.
Storage Queue Data Contributor `974c5e8b-45b9-4653-ba55-5f855dd0fb88`	Read, write, and delete Azure Storage queues and queue messages.

The generated Bicep is a starting point and is influenced by changes to the provisioning infrastructure in C#. Customizations to the Bicep file directly will be overwritten, so make changes through the C# provisioning APIs to ensure they are reflected in the generated files.

Customize provisioning infrastructure

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This type enables the customization of the generated Bicep by providing a fluent API to configure the Azure resources—using the ConfigureInfrastructure<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API. For example, you can configure the `kind`, `sku`, `properties`, and more. The following example demonstrates how to customize the Azure Storage resource:

C#

```
builder.AddAzureStorage("storage")
    .ConfigureInfrastructure(infra =>
    {
        var storageAccount = infra.GetProvisionableResources()
                                  .OfType<StorageAccount>()
                                  .Single();

        storageAccount.AccessTier = StorageAccountAccessTier.Cool;
        storageAccount.Sku = new StorageSku { Name =
StorageSkuName.PremiumZrs };
        storageAccount.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
  - The `infra` parameter is an instance of the AzureResourceInfrastructure type.
  - The provisionable resources are retrieved by calling the GetProvisionableResources() method.

- The single StorageAccount is retrieved.
- The StorageAccount.AccessTier is assigned to StorageAccountAccessTier.Cool.
- The StorageAccount.Sku is assigned to a new StorageSku with a `Name` of PremiumZrs.
- A tag is added to the storage account with a key of `ExampleKey` and a value of `Example value`.

There are many more configuration options available to customize the Azure Storage resource. For more information, see Azure.Provisioning.Storage.

Connect to an existing Azure Storage account

You might have an existing Azure Storage account that you want to connect to. Instead of representing a new Azure Storage resource, you can add a connection string to the app host. To add a connection to an existing Azure Storage account, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var blobs = builder.AddConnectionString("blobs");

builder.AddProject<Projects.WebApplication>("web")
       .WithReference(blobs);

// After adding all resources, run the app...
```

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section. The app host injects this connection string as an environment variable into all dependent resources, for example:

JSON

```
{
    "ConnectionStrings": {
```

```
        "blobs": "https://{account_name}.blob.core.windows.net/"
    }
}
```

The dependent resource can access the injected connection string by calling the GetConnectionString method, and passing the connection name as the parameter, in this case `"blobs"`. The `GetConnectionString` API is shorthand for `IConfiguration.GetSection("ConnectionStrings")[name]`.

Add Azure Storage emulator resource

To add an Azure Storage emulator resource, chain a call on an `IResourceBuilder<AzureStorageResource>` to the RunAsEmulator API:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage")
                     .RunAsEmulator();

// After adding all resources, run the app...
```

When you call `RunAsEmulator`, it configures your storage resources to run locally using an emulator. The emulator in this case is Azurite. The Azurite open-source emulator provides a free local environment for testing your Azure Blob, Queue Storage, and Table Storage apps and it's a perfect companion to the .NET Aspire Azure hosting integration. Azurite isn't installed, instead, it's accessible to .NET Aspire as a container. When you add a container to the app host, as shown in the preceding example with the `mcr.microsoft.com/azure-storage/azurite` image, it creates and starts the container when the app host starts. For more information, see Container resource lifecycle.

Configure Azurite container

There are various configurations available to container resources, for example, you can configure the container's ports, environment variables, it's lifetime, and more.

Configure Azurite container ports

By default, the Azurite container when configured by .NET Aspire, exposes the following endpoints:

Expand table

Endpoint	Container port	Host port
`blob`	10000	dynamic
`queue`	10001	dynamic
`table`	10002	dynamic

The port that they're listening on is dynamic by default. When the container starts, the ports are mapped to a random port on the host machine. To configure the endpoint ports, chain calls on the container resource builder provided by the `RunAsEmulator` method as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage").RunAsEmulator(
                      azurite =>
                      {
                          azurite.WithBlobPort(27000)
                                 .WithQueuePort(27001)
                                 .WithTablePort(27002);
                      });

// After adding all resources, run the app...
```

The preceding code configures the Azurite container's existing `blob`, `queue`, and `table` endpoints to listen on ports `27000`, `27001`, and `27002`, respectively. The Azurite container's ports are mapped to the host ports as shown in the following table:

Expand table

Endpoint name	Port mapping (`container:host`)
`blob`	`10000:27000`
`queue`	`10001:27001`
`table`	`10002:27002`

Configure Azurite container with persistent lifetime

To configure the Azurite container with a persistent lifetime, call the WithLifetime method on the Azurite container resource and pass ContainerLifetime.Persistent:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage").RunAsEmulator(
                      azurite =>
                      {
                          azurite.WithLifetime(ContainerLifetime.Persistent);
                      });

// After adding all resources, run the app...
```

For more information, see Container resource lifetime.

Configure Azurite container with data volume

To add a data volume to the Azure Storage emulator resource, call the WithDataVolume method on the Azure Storage emulator resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage").RunAsEmulator(
                      azurite =>
                      {
                          azurite.WithDataVolume();
                      });

// After adding all resources, run the app...
```

The data volume is used to persist the Azurite data outside the lifecycle of its container. The data volume is mounted at the `/data` path in the Azurite container and when a `name` parameter isn't provided, the name is formatted as `.azurite/{resource name}`. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Configure Azurite container with data bind mount

To add a data bind mount to the Azure Storage emulator resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("storage").RunAsEmulator(
                      azurite =>
                      {
                          azurite.WithDataBindMount("../Azurite/Data");
                      });

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Azurite data across container restarts. The data bind mount is mounted at the `../Azurite/Data` path on the host machine relative to the app host directory (IDistributedApplicationBuilder.AppHostDirectory) in the Azurite container. For more information on data bind mounts, see Docker docs: Bind mounts.

Connect to storage resources

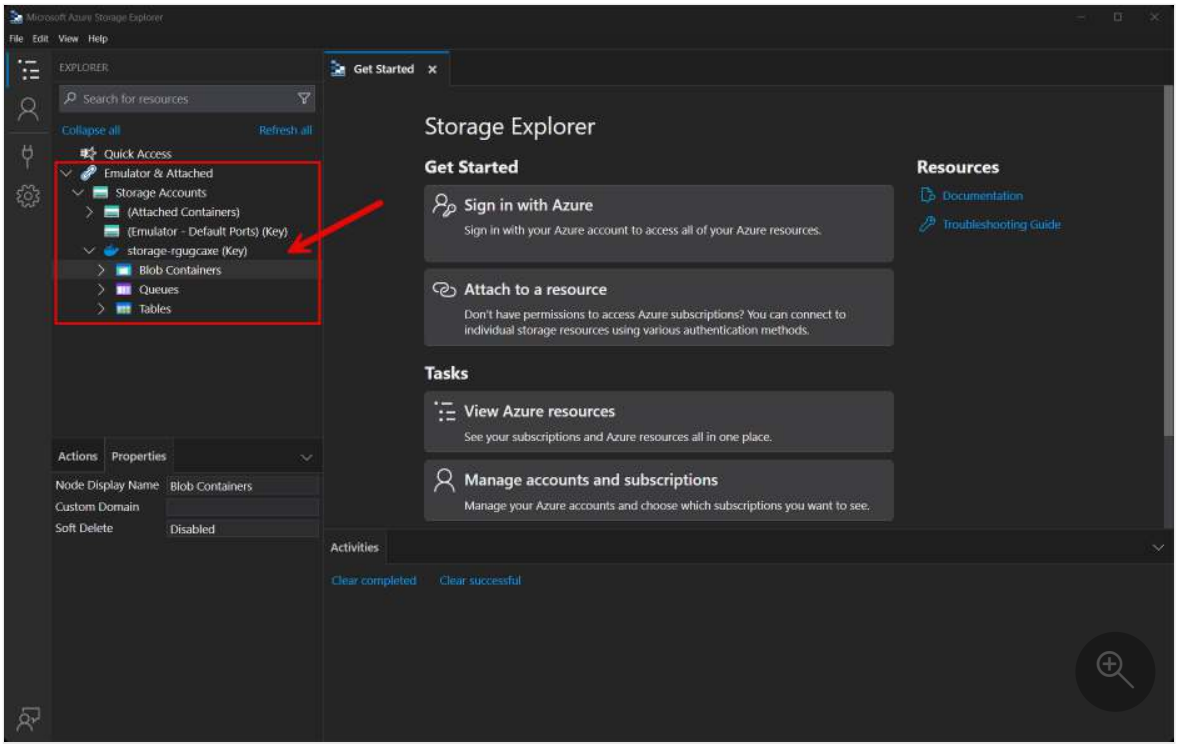
When the .NET Aspire app host runs, the storage resources can be accessed by external tools, such as the Azure Storage Explorer. If your storage resource is running locally using Azurite, it will automatically be picked up by the Azure Storage Explorer.

> **Note**
>
> The Azure Storage Explorer discovers Azurite storage resources assuming the default ports are used. If you've configured the Azurite container to use different ports, you'll need to configure the Azure Storage Explorer to connect to the correct ports.

To connect to the storage resource from Azure Storage Explorer, follow these steps:

1. Run the .NET Aspire app host.
2. Open the Azure Storage Explorer.

3. View the **Explorer** pane.
4. Select the **Refresh all** link to refresh the list of storage accounts.
5. Expand the **Emulator & Attached** node.
6. Expand the **Storage Accounts** node.
7. You should see a storage account with your resource's name as a prefix:



You're free to explore the storage account and its contents using the Azure Storage Explorer. For more information on using the Azure Storage Explorer, see Get started with Storage Explorer.

Add Azure Table Storage resource

In your app host project, register the Azure Table Storage integration by chaining a call to AddTables on the `IResourceBuilder<IAzureStorageResource>` instance returned by AddAzureStorage. The following example demonstrates how to add an Azure Table Storage resource named `storage` and a table resource named `tables`:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var tables = builder.AddAzureStorage("storage")
                    .AddTables("tables");

builder.AddProject<Projects.ExampleProject>()
```

```
        .WithReference(tables)
        .WaitFor(tables);

// After adding all resources, run the app...
```

The preceding code:

- Adds an Azure Storage resource named `storage`.
- Adds a table storage resource named `tables` to the storage resource.
- Adds the `storage` resource to the `ExampleProject` and waits for it to be ready before starting the project.

Hosting integration health checks

The Azure Storage hosting integration automatically adds a health check for the storage resource. It's added only when running as an emulator, and verifies the Azurite container is running and that a connection can be established to it. The hosting integration relies on the AspNetCore.HealthChecks.Azure.Storage.Blobs NuGet package.

Client integration

To get started with the .NET Aspire Azure Data Tables client integration, install the Aspire.Azure.Data.Tables NuGet package in the client-consuming project, that is, the project for the application that uses the Azure Data Tables client. The Azure Data Tables client integration registers a TableServiceClient instance that you can use to interact with Azure Table Storage.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Azure.Data.Tables
```

Add Azure Table Storage client

In the *Program.cs* file of your client-consuming project, call the AddAzureTableClient extension method on any IHostApplicationBuilder to register a `TableServiceClient` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddAzureTableClient("tables");
```

You can then retrieve the `TableServiceClient` instance using dependency injection. For example, to retrieve the client from a service:

C#

```
public class ExampleService(TableServiceClient client)
{
    // Use client...
}
```

Configuration

The .NET Aspire Azure Table Storage integration provides multiple options to configure the `TableServiceClient` based on the requirements and conventions of your project.

Use configuration providers

The .NET Aspire Azure Table Storage integration supports Microsoft.Extensions.Configuration. It loads the AzureDataTablesSettings and TableClientOptions from configuration by using the `Aspire:Azure:Data:Tables` key. The following snippet is an example of a *appsettings.json* file that configures some of the options:

JSON

```
{
  "Aspire": {
    "Azure": {
      "Data": {
        "Tables": {
          "ServiceUri": "YOUR_URI",
          "DisableHealthChecks": true,
          "DisableTracing": false,
          "ClientOptions": {
            "EnableTenantDiscovery": true
          }
        }
      }
    }
  }
}
```

For the complete Azure Data Tables client integration JSON schema, see Aspire.Azure.Data.Tables/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<AzureDataTablesSettings> configureSettings` delegate to set up some or all the options inline, for example to configure the `ServiceUri`:

C#

```
builder.AddAzureTableClient(
    "tables",
    settings => settings.DisableHealthChecks = true);
```

You can also set up the TableClientOptions using `Action<IAzureClientBuilder<TableServiceClient, TableClientOptions>> configureClientBuilder` delegate, the second parameter of the `AddAzureTableClient` method. For example, to set the `TableServiceClient` ID to identify the client:

C#

```
builder.AddAzureTableClient(
    "tables",
    configureClientBuilder: clientBuilder =>
        clientBuilder.ConfigureOptions(
            options => options.EnableTenantDiscovery = true));
```

Client integration health checks

By default, .NET Aspire integrations enable health checks for all services. For more information, see .NET Aspire integrations overview.

The .NET Aspire Azure Data Tables integration:

- Adds the health check when AzureDataTablesSettings.DisableHealthChecks is `false`, which attempts to connect to the Azure Table Storage.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Azure Data Tables integration uses the following log categories:

- `Azure.Core`
- `Azure.Identity`

Tracing

The .NET Aspire Azure Data Tables integration emits the following tracing activities using OpenTelemetry:

- `Azure.Data.Tables.TableServiceClient`

Metrics

The .NET Aspire Azure Data Tables integration currently doesn't support metrics by default due to limitations with the Azure SDK.

See also

- Azure Table Storage docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire Azure Web PubSub integration

Article • 03/18/2025

Includes:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var webPubSub = builder.AddAzureWebPubSub("web-pubsub");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(webPubSub);

// After adding all resources, run the app...
```

The preceding code adds an Azure Web PubSub resource named `web-pubsub` to the app host project. The WithReference method passes the connection information to the `ExampleProject` project.

> **ⓘ Important**
>
> When you call `AddAzureWebPubSub`, it implicitly calls AddAzureProvisioning(IDistributedApplicationBuilder)—which adds support for generating Azure resources dynamically during app startup. The app must configure the appropriate subscription and location. For more information, see Local provisioning: Configuration.

Add an Azure Web PubSub hub resource

To add an Azure Web PubSub hub resource to your app host project, chain a call to the AddHub(IResourceBuilder<AzureWebPubSubResource>, String) method providing a name:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var worker = builder.AddProject<Projects.WorkerService>("worker")
                    .WithExternalHttpEndpoints();

var webPubSub = builder.AddAzureWebPubSub("web-pubsub");
var messagesHub = webPubSub.AddHub("messages");

// After adding all resources, run the app...
```

The preceding code adds an Azure Web PubSub hub resource named `messages`, which enables the addition of event handlers. To add an event handler, call the AddEventHandler:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var worker = builder.AddProject<Projects.WorkerService>("worker")
                    .WithExternalHttpEndpoints();

var webPubSub = builder.AddAzureWebPubSub("web-pubsub");
var messagesHub = webPubSub.AddHub("messages");

messagesHub.AddEventHandler(
    $"{worker.GetEndpoint("https")}/eventhandler/",
    systemEvents: ["connected"]);

// After adding all resources, run the app...
```

The preceding code adds a worker service project named `worker` with an external HTTP endpoint. The hub named `messages` resource is added to the `web-pubsub` resource, and an event handler is added to the `messagesHub` resource. The event handler URL is set to the worker service's external HTTP endpoint. For more information, see Azure Web PubSub event handlers.

Generated provisioning Bicep

When you publish your app, .NET Aspire provisioning APIs generate Bicep alongside the manifest file. Bicep is a domain-specific language for defining Azure resources. For more information, see Bicep Overview.

When you add an Azure Web PubSub resource, the following Bicep is generated:

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location

param sku string = 'Free_F1'

param capacity int = 1

param principalType string

param principalId string

param messages_url_0 string

resource web_pubsub 'Microsoft.SignalRService/webPubSub@2024-03-01' = {
  name: take('webpubsub-${uniqueString(resourceGroup().id)}', 63)
  location: location
  sku: {
```

```
    name: sku
    capacity: capacity
  }
  tags: {
    'aspire-resource-name': 'web-pubsub'
  }
}

resource web_pubsub_WebPubSubServiceOwner
'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(web_pubsub.id, principalId,
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '12cf5a90-
567b-43ae-8102-96cf46c7d9b4'))
  properties: {
    principalId: principalId
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '12cf5a90-
567b-43ae-8102-96cf46c7d9b4')
    principalType: principalType
  }
  scope: web_pubsub
}

resource messages 'Microsoft.SignalRService/webPubSub/hubs@2024-03-01' = {
  name: 'messages'
  properties: {
    eventHandlers: [
      {
        urlTemplate: messages_url_0
        userEventPattern: '*'
        systemEvents: [
          'connected'
        ]
      }
    ]
  }
  parent: web_pubsub
}

output endpoint string = 'https://${web_pubsub.properties.hostName}'
```

The preceding Bicep is a module that provisions an Azure Web PubSub resource with the following defaults:

- `location`: The location of the resource group.
- `sku`: The SKU of the Web PubSub resource, defaults to `Free_F1`.
- `principalId`: The principal ID of the Web PubSub resource.
- `principalType`: The principal type of the Web PubSub resource.
- `messages_url_0`: The URL of the event handler for the `messages` hub.
- `messages`: The name of the hub resource.

- `web_pubsub`: The name of the Web PubSub resource.
- `web_pubsub_WebPubSubServiceOwner`: The role assignment for the Web PubSub resource owner. For more information, see Azure Web PubSub Service Owner.
- `endpoint`: The endpoint of the Web PubSub resource.

The generated Bicep is a starting point and is influenced by changes to the provisioning infrastructure in C#. Customizations to the Bicep file directly will be overwritten, so make changes through the C# provisioning APIs to ensure they are reflected in the generated files.

Customize provisioning infrastructure

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This type enables the customization of the generated Bicep by providing a fluent API to configure the Azure resources—using the ConfigureInfrastructure<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API. For example, you can configure the `kind`, `consistencyPolicy`, `locations`, and more. The following example demonstrates how to customize the Azure Cosmos DB resource:

C#

```
builder.AddAzureWebPubSub("web-pubsub")
    .ConfigureInfrastructure(infra =>
    {
        var webPubSubService = infra.GetProvisionableResources()
                                    .OfType<WebPubSubService>()
                                    .Single();

        webPubSubService.Sku.Name = "Standard_S1";
        webPubSubService.Sku.Capacity = 5;
        webPubSubService.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
  - The `infra` parameter is an instance of the AzureResourceInfrastructure type.
  - The provisionable resources are retrieved by calling the GetProvisionableResources() method.
  - The single WebPubSubService resource is retrieved.
  - The WebPubSubService.Sku object has its name and capacity properties set to `Standard_S1` and `5`, respectively.
  - A tag is added to the Web PubSub resource with a key of `ExampleKey` and a value of `Example value`.

There are many more configuration options available to customize the Web PubSub resource. For more information, see Azure.Provisioning.WebPubSub. For more information, see Azure.Provisioning customization.

Connect to an existing Azure Web PubSub instance

You might have an existing Azure Web PubSub service that you want to connect to. You can chain a call to annotate that your AzureWebPubSubResource is an existing resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var existingPubSubName = builder.AddParameter("existingPubSubName");
var existingPubSubResourceGroup =
builder.AddParameter("existingPubSubResourceGroup");

var webPubSub = builder.AddAzureWebPubSub("web-pubsub")
                       .AsExisting(existingPubSubName,
existingPubSubResourceGroup);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(webPubSub);

// After adding all resources, run the app...
```

For more information on treating Azure Web PubSub resources as existing resources, see Use existing Azure resources.

Alternatively, instead of representing an Azure Web PubSub resource, you can add a connection string to the app host. Which is a weakly-typed approach that's based solely on a `string` value. To add a connection to an existing Azure Web PubSub service, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var webPubSub = builder.ExecutionContext.IsPublishMode
    ? builder.AddAzureWebPubSub("web-pubsub")
    : builder.AddConnectionString("web-pubsub");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(webPubSub);

// After adding all resources, run the app...
```

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section:

JSON

```
{
  "ConnectionStrings": {
    "web-pubsub": "https://{account_name}.webpubsub.azure.com"
  }
}
```

For more information, see Add existing Azure resources with connection strings.

Client integration

The .NET Aspire Azure Web PubSub client integration is used to connect to an Azure Web PubSub service using the WebPubSubServiceClient. To get started with the .NET Aspire Azure Web PubSub service client integration, install the Aspire.Azure.Messaging.WebPubSub NuGet package in the application.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Azure.Messaging.WebPubSub
```

Supported Web PubSub client types

The following Web PubSub client types are supported by the library:

Expand table

Azure client type	Azure options class	.NET Aspire settings class
WebPubSubServiceClient	WebPubSubServiceClientOptions	AzureMessagingWebPubSubSettings

Add Web PubSub client

In the *Program.cs* file of your client-consuming project, call the AddAzureWebPubSubServiceClient extension method to register a `WebPubSubServiceClient` for use via the dependency injection container. The method takes a connection name parameter:

C#

```
builder.AddAzureWebPubSubServiceClient(
    connectionName: "web-pubsub");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Web PubSub resource in the app host project. For more information, see **Add an Azure Web PubSub resource**.

After adding the `WebPubSubServiceClient`, you can retrieve the client instance using dependency injection. For example, to retrieve your data source object from an example service define it as a constructor parameter and ensure the `ExampleService` class is registered with the dependency injection container:

C#

```
public class ExampleService(WebPubSubServiceClient client)
{
    // Use client...
}
```

For more information, see:

- Azure.Messaging.WebPubSub documentation for examples on using the `WebPubSubServiceClient`.
- Dependency injection in .NET for details on dependency injection.

Add keyed Web PubSub client

There might be situations where you want to register multiple `WebPubSubServiceClient` instances with different connection names. To register keyed Web PubSub clients, call the AddKeyedAzureWebPubSubServiceClient method:

C#

```
builder.AddKeyedAzureWebPubSubServiceClient(name: "messages");
builder.AddKeyedAzureWebPubSubServiceClient(name: "commands");
```

> **ⓘ Important**
>
> When using keyed services, it's expected that your Web PubSub resource configured two named hubs, one for the `messages` and one for the `commands`.

Then you can retrieve the client instances using dependency injection. For example, to retrieve the clients from a service:

C#

```
public class ExampleService(
    [KeyedService("messages")] WebPubSubServiceClient messagesClient,
    [KeyedService("commands")] WebPubSubServiceClient commandsClient)
{
    // Use clients...
}
```

For more information, see Keyed services in .NET.

Configuration

The .NET Aspire Azure Web PubSub library provides multiple options to configure the Azure Web PubSub connection based on the requirements and conventions of your project. Either an `Endpoint` or a `ConnectionString` must be supplied.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `AddAzureWebPubSubServiceClient`:

C#

```
builder.AddAzureWebPubSubServiceClient(
    "web-pubsub",
    settings => settings.HubName = "your_hub_name");
```

The connection information is retrieved from the `ConnectionStrings` configuration section. Two connection formats are supported:

- **Service endpoint (recommended)**: Uses the service endpoint with `DefaultAzureCredential`.

  JSON

  ```
  {
    "ConnectionStrings": {
      "web-pubsub": "https://{account_name}.webpubsub.azure.com"
    }
  }
  ```

- **Connection string**: Includes an access key.

  JSON

  ```
  {
    "ConnectionStrings": {
      "web-pubsub":
  "Endpoint=https://{account_name}.webpubsub.azure.com;AccessKey=
  {account_key}"
    }
  }
  ```

Use configuration providers

The library supports Microsoft.Extensions.Configuration. It loads settings from configuration using the `Aspire:Azure:Messaging:WebPubSub` key:

JSON

```
{
  "Aspire": {
    "Azure": {
      "Messaging": {
        "WebPubSub": {
          "DisableHealthChecks": true,
          "HubName": "your_hub_name"
        }
      }
```

For the complete Azure OpenAI client integration JSON schema, see Aspire.Azure.Messaging.WebPubSub/ConfigurationSchema.json .

You can configure settings inline:

C

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques shown in the Configuration section.

The .NET Aspire Azure Web PubSub

- `Azure`
- `Azure.Core`

Metrics

The .NET Aspire Azure Web PubSub integration currently doesn't support metrics by default due to limitations with the Azure SDK for .NET. If that changes in the future, this section will be updated to reflect those changes.

See also

- Azure Web PubSub
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire Dapr integration

Article • 01/23/2025

Distributed Application Runtime (Dapr) offers developer APIs that serve as a conduit for interacting with other services and dependencies and abstract the application from the specifics of those services and dependencies. Dapr and .NET Aspire can work together to improve your local development experience. By using Dapr with .NET Aspire, you can focus on writing and implementing .NET-based distributed applications instead of local on-boarding.

In this guide, you'll learn how to take advantage of Dapr's abstraction and .NET Aspire's opinionated configuration of cloud technologies to build simple, portable, resilient, and secured microservices at scale.

Comparing .NET Aspire and Dapr

At first sight Dapr and .NET Aspire may look like they have overlapping functionality, and they do. However, they take different approaches. .NET Aspire is opinionated on how to build distributed applications on a cloud platform and focuses on improving the local development experience. Dapr is a runtime that abstracts away the common complexities of the underlying cloud platform both during development and in production. It relies on sidecars to provide abstractions for things like configuration, secret management, and messaging. The underlying technology can be easily switched out through configuration files, while your code does not need to change.

Expand table

Aspect	.NET Aspire	Dapr
Purpose	Designed to make it easier to develop cloud-native solutions on local development computers.	Designed to make it easier to develop and run distributed apps with common APIs that can be easily swapped.
APIs	Developers must call resource APIs using their specific SDKs	Developers call APIs in the Dapr sidecar, which forwards the call to the correct API. It's easy to swap resource APIs without changing code in your microservices.
Languages	You write microservices in .NET languages, Go, Python, Javascript, and others.	You can call Dapr sidecar functions in any language that supports HTTP/gRPC interfaces.

Aspect	.NET Aspire	Dapr
Security policies	Doesn't include security policies but can securely configure connections between inter-dependent resources.	Includes customizable security policies that control which microservices have access to other services or resources.
Deployment	There are deployment tools for Azure and Kubernetes.	Doesn't include deployment tools. Apps are usually deployed with Continuous Integration/Continuous Development (CI/CD) systems.
Dashboard	Provides a comprehensive view of the resources and their telemetry and supports listening on any OTEL supported resource.	Limited to Dapr resources only.

.NET Aspire makes setting up and debugging Dapr applications easier by providing a straightforward API to configure Dapr sidecars, and by exposing the sidecars as resources in the dashboard.

Explore Dapr components with .NET Aspire

Dapr provides many built-in components, and when you use Dapr with .NET Aspire you can easily explore and configure these components. Don't confuse these components with .NET Aspire integrations. For example, consider the following:

- Dapr—State stores: Call AddDaprStateStore to add a configured state store to your .NET Aspire project.
- Dapr—Pub Sub: Call AddDaprPubSub to add a configured pub sub to your .NET Aspire project.
- Dapr—Components: Call AddDaprComponent to add a configured integration to your .NET Aspire project.

Install Dapr

This integration requires Dapr version 1.13 or later. To install Dapr, see Install the Dapr CLI. After installing the Dapr CLI, run the `dapr init`, as described in Initialize Dapr in your local environment.

> **Important**

If you attempt to run the .NET Aspire solution without the Dapr CLI, you'll receive the following error:

```plaintext
Unable to locate the Dapr CLI.
```

Hosting integration

In your .NET Aspire solution, to integrate Dapr and access its types and APIs, add the Aspire.Hosting.Dapr NuGet package in the app host project.

.NET CLI

```
dotnet add package Aspire.Hosting.Dapr
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Dapr sidecar to .NET Aspire resources

Dapr uses the sidecar pattern. The Dapr sidecar runs alongside your app as a lightweight, portable, and stateless HTTP server that listens for incoming HTTP requests from your app.

To add a sidecar to a .NET Aspire resource, call the WithDaprSidecar method on it. The `appId` parameter is the unique identifier for the Dapr application, but it's optional. If you don't provide an `appId`, the parent resource name is used instead.

C#

```
using Aspire.Hosting.Dapr;

var builder = DistributedApplication.CreateBuilder(args);

var apiService = builder
    .AddProject<Projects.Dapr_ApiService>("apiservice")
    .WithDaprSidecar();
```

Configure Dapr sidecars

The `WithDaprSidecar` method offers overloads to configure your Dapr sidecar options like `AppId` and various ports. In the following example, the Dapr sidecar is configured with specific ports for GRPC, HTTP, metrics, and a specific app ID.

C#

```
DaprSidecarOptions sidecarOptions = new()
{
    AppId = "FirstSidecar",
    DaprGrpcPort = 50001,
    DaprHttpPort = 3500,
    MetricsPort = 9090
};

builder.AddProject<Projects.Dapr_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(apiService)
    .WithDaprSidecar(sidecarOptions);
```

Complete Dapr app host example

Putting everything together, consider the following example of a .NET Aspire app host project that includes:

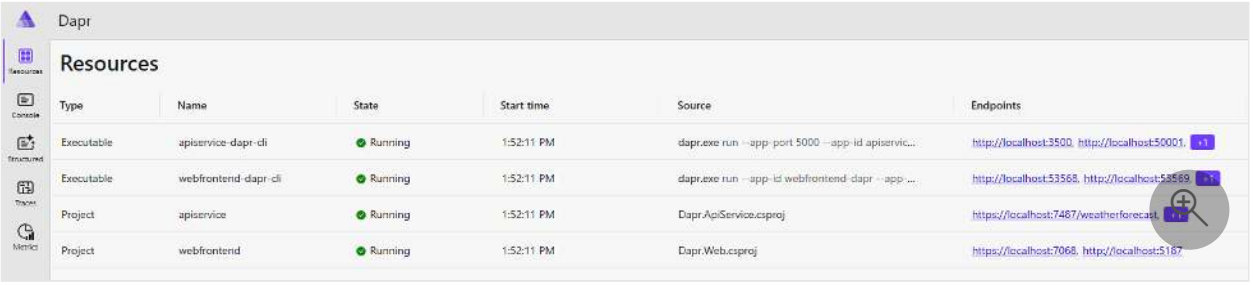
- A backend API service that declares a Dapr sidecar with defaults.
- A web frontend project that declares a Dapr sidecar with specific options, such as explict ports.

C#

```
using Aspire.Hosting.Dapr;

var builder = DistributedApplication.CreateBuilder(args);

var apiService = builder
    .AddProject<Projects.Dapr_ApiService>("apiservice")
    .WithDaprSidecar();

DaprSidecarOptions sidecarOptions = new()
{
    AppId = "FirstSidecar",
    DaprGrpcPort = 50001,
    DaprHttpPort = 3500,
    MetricsPort = 9090
};
```

```
builder.AddProject<Projects.Dapr_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(apiService)
    .WithDaprSidecar(sidecarOptions);

builder.Build().Run();
```

When you start the .NET Aspire solution, the dashboard shows the Dapr sidecar as a resource, with its status and logs.



Use Dapr sidecars in consuming .NET Aspire projects

To use Dapr APIs from .NET Aspire resources, you can use the Dapr.AspNetCore/ NuGet package. The Dapr SDK provides a set of APIs to interact with Dapr sidecars.

> **Note**
>
> Use the `Dapr.AspNetCore` library for the Dapr integration with ASP.NET (DI integration, registration of subscriptions, etc.). Non-ASP.NET apps (such as console apps) can just use the Dapr.Client to make calls through the Dapr sidecar.

.NET CLI

.NET CLI

```
dotnet add package Dapr.AspNetCore
```

Add Dapr client

Once installed into an ASP.NET Core project, the SDK can be added to the service builder.

C#

```
builder.Services.AddDaprClient();
```

Invoke Dapr methods

An instance of `DaprClient` can now be injected into your services to interact with the Dapr sidecar through the Dapr SDK:

C#

```
using Dapr.Client;

namespace Dapr.Web;

public class WeatherApiClient(DaprClient client)
{
    public async Task<WeatherForecast[]> GetWeatherAsync(
        int maxItems = 10, CancellationToken cancellationToken = default)
    {
        List<WeatherForecast>? forecasts =
            await client.InvokeMethodAsync<List<WeatherForecast>>(
                HttpMethod.Get,
                "apiservice",
                "weatherforecast",
                cancellationToken);

        return forecasts?.Take(maxItems)?.ToArray() ?? [];
    }
}

public record WeatherForecast(DateOnly Date, int TemperatureC, string?
Summary)
{
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
}
```

`InvokeMethodAsync` is the method that sends an HTTP request to the Dapr sidecar. It is a generic method that takes:

- An HTTP verb.
- The Dapr app ID of the service to call.
- The method name.
- A cancellation token.

Depending on the HTTP verb, it can also take a request body and headers. The generic type parameter is the type of the response body.

The full *Program.cs* file for the frontend project shows:

- The Dapr client being added to the service builder.
- The `WeatherApiClient` class that uses the Dapr client to call the backend service.

C#

```
using Dapr.Web;
using Dapr.Web.Components;

var builder = WebApplication.CreateBuilder(args);

// Add service defaults & Aspire components.
builder.AddServiceDefaults();

// Add services to the container.
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();

builder.Services.AddOutputCache();

builder.Services.AddDaprClient();

builder.Services.AddTransient<WeatherApiClient>();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseStaticFiles();
app.UseAntiforgery();

app.UseOutputCache();

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();

app.MapDefaultEndpoints();

app.Run();
```

For example, in a Blazor project, you can inject the `WeatherApiClient` class into a razor page and use it to call the backend service:

C#

```
@page "/weather"
@attribute [StreamRendering(true)]
@attribute [OutputCache(Duration = 5)]

@inject WeatherApiClient WeatherApi

<PageTitle>Weather</PageTitle>

<h1>Weather</h1>

<p>This component demonstrates showing data loaded from a backend API
service.</p>

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
        </tbody>
    </table>
}

@code {
    private WeatherForecast[]? forecasts;

    protected override async Task OnInitializedAsync()
    {
        forecasts = await WeatherApi.GetWeatherAsync();
    }
}
```

When the Dapr SDK is used, the Dapr sidecar is called over HTTP. The Dapr sidecar then forwards the request to the target service. While the target service runs in a separate process from the sidecar, the integration related to the service runs in the Dapr sidecar and is responsible for service discovery and routing the request to the target service.

Next steps

- Dapr
- Dapr documentation
- Dapr GitHub repo
- .NET Aspire Dapr sample app
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire Elasticsearch integration (Preview)

Article • 02/14/2025

Includes: ✅ Hosting integration and ✅ Client integration

Elasticsearch is a distributed, RESTful search and analytics engine, scalable data store, and vector database capable of addressing a growing number of use cases. The .NET Aspire Elasticsearch integration enables you to connect to existing Elasticsearch instances, or create new instances from .NET with the docker.io/library/elasticsearch container image.

Hosting integration

The Elasticsearch hosting integration models an Elasticsearch instance as the ElasticsearchResource type. To access this type and APIs that allow you to add it to your Aspire.Hosting.Elasticsearch NuGet package in the app host project.

.NET CLI

```
.NET CLI
dotnet add package Aspire.Hosting.Elasticsearch
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Elasticsearch resource

In your app host project, call AddElasticsearch on the `builder` instance to add an Elasticsearch resource:

```
C#
var builder = DistributedApplication.CreateBuilder(args);

var elasticsearch = builder.AddElasticsearch("elasticsearch");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(elasticsearch);
```

```
// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/library/elasticsearch` image, it creates a new Elasticsearch instance on your local machine. A reference to your Elasticsearch resource (the `elasticsearch` variable) is added to the `ExampleProject`. The Elasticsearch resource includes default credentials with a `username` of `"elastic"` and randomly generated `password` using the CreateDefaultPasswordParameter method when a password wasn't provided.

The WithReference method configures a connection in the `ExampleProject` named `"elasticsearch"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing Elasticsearch instance, call AddConnectionString instead. For more information, see Reference existing resources.

Add Elasticsearch resource with data volume

To add a data volume to the Elasticsearch resource, call the WithDataVolume method on the Elasticsearch resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var elasticsearch = builder.AddElasticsearch("elasticsearch")
                           .WithDataVolume(isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
        .WithReference(elasticsearch);

// After adding all resources, run the app...
```

The data volume is used to persist the Elasticsearch data outside the lifecycle of its container. The data volume is mounted at the `/usr/share/elasticsearch/data` path in the Elasticsearch container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add Elasticsearch resource with data bind mount

To add a data bind mount to the Elasticsearch resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var elasticsearch = builder.AddElasticsearch("elasticsearch")
                           .WithDataBindMount(
                               source: @"C:\Elasticsearch\Data",
                               isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
        .WithReference(elasticsearch);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Elasticsearch data across container restarts. The data bind mount is mounted at the `C:\Elasticsearch\Data` on Windows (or `/Elasticsearch/Data` on Unix) path on the host machine in the Elasticsearch container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add Elasticsearch resource with password parameter

When you want to explicitly provide the password used by the container image, you can provide these credentials as parameters. Consider the following alternative example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var password = builder.AddParameter("password", secret: true);
var elasticsearch = builder.AddElasticsearch("elasticsearch", password);
```

```
builder.AddProject<Projects.ExampleProject>()
       .WithReference(elasticsearch);

// After adding all resources, run the app...
```

For more information on providing parameters, see External parameters.

Hosting integration health checks

The Elasticsearch hosting integration automatically adds a health check for the Elasticsearch resource. The health check verifies that the Elasticsearch instance is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Elasticsearch NuGet package.

Client integration

To get started with the .NET Aspire Elasticsearch client integration, install the Aspire.Elastic.Clients.Elasticsearch NuGet package in the client-consuming project, that is, the project for the application that uses the Elasticsearch client. The Elasticsearch client integration registers an ElasticsearchClient instance that you can use to interact with Elasticsearch.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Elastic.Clients.Elasticsearch
```

Add Elasticsearch client

In the *Program.cs* file of your client-consuming project, call the AddElasticsearchClient extension method on any IHostApplicationBuilder to register an `ElasticsearchClient` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddElasticsearchClient(connectionName: "elasticsearch");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Elasticsearch resource in the app host project. For more information, see **Add Elasticsearch resource**.

You can then retrieve the `ElasticsearchClient` instance using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(ElasticsearchClient client)
{
    // Use client...
}
```

Add keyed Elasticsearch client

There might be situations where you want to register multiple `ElasticsearchClient` instances with different connection names. To register keyed Elasticsearch clients, call the AddKeyedElasticsearchClient:

C#

```
builder.AddKeyedElasticsearchClient(name: "products");
builder.AddKeyedElasticsearchClient(name: "orders");
```

Then you can retrieve the `ElasticsearchClient` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("products")] ElasticsearchClient productsClient,
    [FromKeyedServices("orders")] ElasticsearchClient ordersClient)
{
    // Use clients...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire Elasticsearch client integration provides multiple options to configure the server connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddElasticsearchClient`:

C#

```
builder.AddElasticsearchClient("elasticsearch");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "elasticsearch": "http://elastic:password@localhost:27011"
  }
}
```

Use configuration providers

The .NET Aspire Elasticsearch Client integration supports Microsoft.Extensions.Configuration. It loads the ElasticClientsElasticsearchSettings from configuration by using the `Aspire:Elastic:Clients:Elasticsearch` key. Consider the following example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "Elastic": {
      "Clients": {
        "Elasticsearch": {
            "DisableHealthChecks": false,
            "DisableTracing": false,
            "HealthCheckTimeout": "00:00:03",
            "ApiKey": "<Valid ApiKey>",
            "Endpoint": "http://elastic:password@localhost:27011",
            "CloudId": "<Valid CloudId>"
        }
```

```
      }
    }
  }
}
```

For the complete Elasticsearch client integration JSON schema, see Aspire.Elastic.Clients.Elasticsearch/ConfigurationSchema.json.

Use inline delegates

Also you can pass the `Action<ElasticClientsElasticsearchSettings> configureSettings` delegate to set up some or all the options inline, for example to set the API key from code:

C#

```
builder.AddElasticsearchClient(
    "elasticsearch",
    static settings =>
        settings.Endpoint = new
Uri("http://elastic:password@localhost:27011"));
```

Use a `CloudId` and an `ApiKey` with configuration providers

When using Elastic Cloud, you can provide the `CloudId` and `ApiKey` in `Aspire:Elastic:Clients:Elasticsearch` section when calling `builder.AddElasticsearchClient`.

C#

```
builder.AddElasticsearchClient("elasticsearch");
```

Consider the following example *appsettings.json* that configures the options:

JSON

```
{
  "Aspire": {
    "Elastic": {
      "Clients": {
        "Elasticsearch": {
            "ApiKey": "<Valid ApiKey>",
            "CloudId": "<Valid CloudId>"
        }
      }
```

```
    }
  }
}
```

Use a `CloudId` and an `ApiKey` with inline delegates

C#

```
builder.AddElasticsearchClient(
    "elasticsearch",
    static settings =>
    {
        settings.ApiKey = "<Valid ApiKey>";
        settings.CloudId = "<Valid CloudId>";
    });
```

Client integration health checks

By default, .NET Aspire integrations enable health checks for all services. For more information, see .NET Aspire integrations overview.

The .NET Aspire Elasticsearch integration uses the configured client to perform a `PingAsync`. If the result is an HTTP 200 OK, the health check is considered healthy, otherwise it's unhealthy. Likewise, if there's an exception, the health check is considered unhealthy with the error propagating through the health check failure.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Tracing

The .NET Aspire Elasticsearch integration will emit the following tracing activities using OpenTelemetry:

- `Elastic.Transport`

See also

- Elasticsearch .NET
- .NET Aspire integrations
- .NET Aspire GitHub repo

Entity Framework Core overview

Article • 03/25/2025

In a cloud-native solution, such as those .NET Aspire is built to create, microservices often need to store data in relational databases. .NET Aspire includes integrations that you can use to ease that task, some of which use the Entity Framework Core (EF Core) object-relational mapper (O/RM) approach to streamline the process.

Developers use O/RMs to work with databases using code objects instead of SQL queries. EF Core automatically codes database interactions by generating SQL queries based on Language-Integrated Query (LINQ) queries. EF Core supports various database providers, including SQL Server, PostgreSQL, and MySQL, so it's easy to interact with relational databases while following object-oriented principles.

The most commonly used .NET Aspire EF Core client integrations are:

- Cosmos DB Entity Framework Core integration
- MySQL Pomelo Entity Framework Core integration
- Oracle Entity Framework Core integration
- PostgreSQL Entity Framework Core integration
- SQL Server Entity Framework Core integration

Overview of EF Core

O/RMs create a model that matches the schema and relationships defined in the database. Code against this model to query the data, create new records, or make other changes. In EF Core the model consists of:

- A set of entity classes, each of which represents a table in the database and its columns.
- A context class that represents the whole database.

An entity class might look like this:

C#

```
using System.ComponentModel.DataAnnotations;

namespace SupportDeskProject.Data;

public sealed class SupportTicket
{
    public int Id { get; set; }
    [Required]
```

> **Note**
>
> EF Core also supports creating, modifying, and deleted records and complex queries. For more information, see Querying Data and Saving Data

How .NET Aspire can help

.NET Aspire is designed to help build observable, production-ready, cloud-native solutions that consist of multiple microservices. It orchestrates multiple projects, each of which may be a microservice written by a dedicated team, and connects them to each other. It provides integrations that make it easy to connect to common services, such as databases.

If you want to use EF Core in any of your microservices, .NET Aspire can help by:

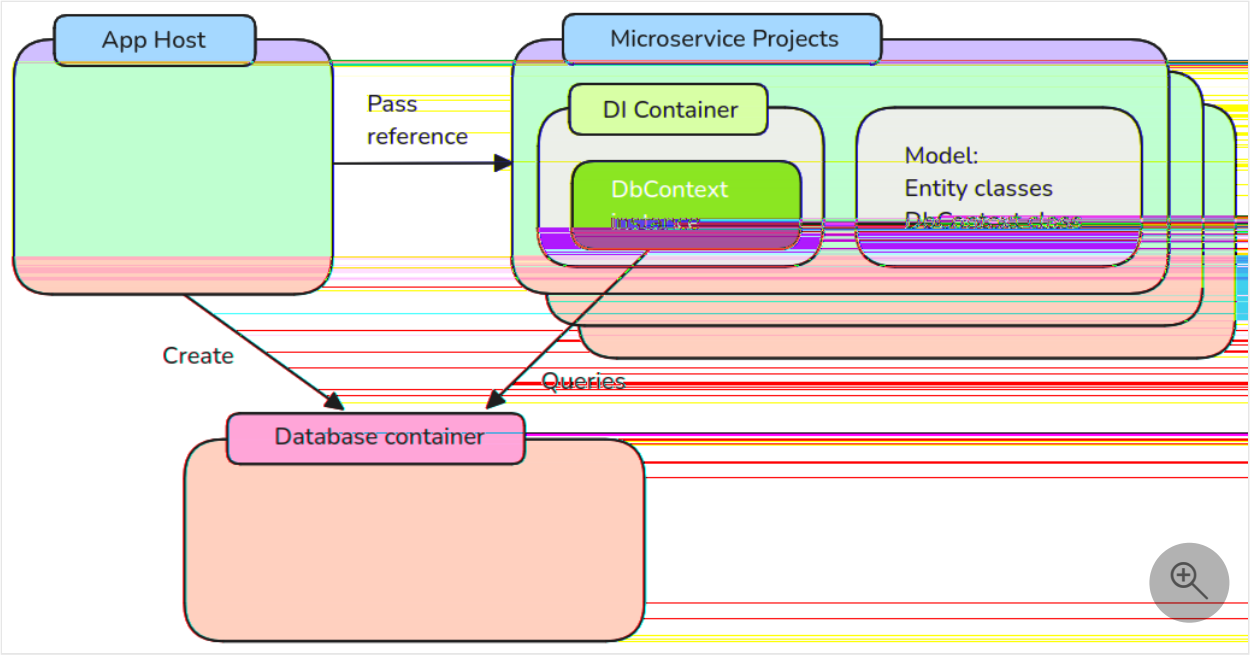
- Managing the database container, or a connection to an existing database, centrally in the App Host project and passing its reference to any project that uses it.

  > **Important**
  >
  > In .NET Aspire, EF Core is implemented by client integrations, not hosting integrations. The centralized management of the database in the App Host doesn't involve EF Core, which runs in consuming microservice projects instead. For more information, see Cosmos DB Hosting integration, MySQL Pomelo Hosting integration, Oracle Hosting integration, PostgreSQL Hosting integration, or SQL Server Hosting integration.

- Providing EF Core-aware integrations that make it easy to create contexts in microservice projects. There are EF Core integrations for SQL Server, MySQL, PostgreSQL, Oracle, Cosmos DB, and other popular database systems.

To use EF Core in your microservice, you must:

- Define the EF Core model with entity classes and context classes.
- Create an instance of the data context, using the reference passed from the App Host, and add it to the Dependency Injection (DI) container.
- When you want to interact with the database, obtain the context from DI and use it to execute LINQ queries against the database as normal for any EF Core code.



Both defining the EF Core model and querying the database are the same in .NET Aspire projects as in any other EF Core app. However, creating the data context differs. In the rest of this article, you'll learn how to create an configure EF Core contexts in .NET Aspire project.

Use .NET Aspire to create an EF Core context

In EF Core, a **context** is a class used to interact with the database. Contexts inherit from the DbContext class. They provide access to the database through properties of type `DbSet<T>`, where each `DbSet` represents a table or collection of entities in the database. The context also manages database connections, tracks changes to entities, and handles operations like saving data and executing queries.

The .NET Aspire EF Core client integrations each include extension methods named `Add{DatabaseSystem}DbContext`, where **{DatabaseSystem}** is the name identifying the database product you're using. For example, consider the SQL Server EF Core client integration, the method is named AddSqlServerDbContext and for the PostgreSQL client integration, the method is named AddNpgsqlDbContext.

These .NET Aspire add context methods:

- Check that a context of the same type isn't already registered in the dependency injection (DI) container.
- Use the connection name you pass to the method to get the connection string from the application builder. This connection name must match the name used when adding the corresponding resource to the app host project.
- Apply the `DbContext` options, if you passed them.

- Add the specified `DbContext` to the DI container with context pooling enabled.
- Apply the recommended defaults, unless you've disabled them through the .NET Aspire EF Core settings:
  - Enable tracing.
  - Enable health checks.
  - Enable connection resiliency.

Use these .NET Aspire add context methods when you want a simple way to create a context and don't yet need advanced EF Core customization.

C#

```
builder.AddSqlServerDbContext<ExampleDbContext>(connectionName: "database");
```

> **Tip**
>
> For more information about SQL Server hosting and client integrations, see .NET Aspire SQL Server Entity Framework Core integration.

You obtain the `ExampleDbContext` object from the DI container in the same way as for any other service:

C#

```
public class ExampleService(ExampleDbContext context)
{
    // Use context...
}
```

Use EF Core to add and enrich context

Alternatively, you can add a context to the DI container using the standard EF Core AddDbContextPool method, as commonly used in non-.NET Aspire projects:

C#

```
builder.Services.AddDbContextPool<ExampleDbContext>(options =>
{
    var connectionString =
builder.Configuration.GetConnectionString("database")
        ?? throw new InvalidOperationException("Connection string 'database'
not found.");
```

```
    options.UseSqlServer(connectionString);
});
```

You have more flexibility when you create the context in this way, for example:

- You can reuse existing configuration code for the context without rewriting it for .NET Aspire.
- You can choose not to use EF Core context pooling, which may be necessary in some circumstances. For more information, see Use EF Core context pooling in .NET Aspire
- You can use EF Core context factories or change the lifetime for the EF Core services. For more information, see Use EF Core context factories in .NET Aspire
- You can use dynamic connection strings. For more information, see Use EF Core with dynamic connection strings in .NET Aspire
- You can use EF Core interceptors that depend on DI services to modify database operations. For more information, see Use EF Core interceptors in .NET Aspire

By default, a context configured this way doesn't include .NET Aspire features, such as telemetry and health checks. To add those features, each .NET Aspire EF Core client integration includes a method named `Enrich\<DatabaseSystem\>DbContext`. These enrich context methods:

- Apply an EF Core settings object, if you passed one.
- Configure connection retry settings.
- Apply the recommended defaults, unless you've disabled them through the .NET Aspire EF Core settings:
  - Enable tracing.
  - Enable health checks.
  - Enable connection resiliency.

> **Note**
>
> You must add a context to the DI container before you call an enrich method.

C#

```
builder.EnrichSqlServerDbContext<ExampleDbContext>(
    configureSettings: settings =>
    {
        settings.DisableRetry = false;
        settings.CommandTimeout = 30; // seconds
    });
```

Obtain the context from the DI container using the same code as the previous example:

C#

```
public class ExampleService(ExampleDbContext context)
{
    // Use context...
}
```

Use EF Core interceptors with .NET Aspire

EF Core interceptors allow developers to hook into and modify database operations at various points during the execution of database queries and commands. You can use them to log, modify, or suppress operations with your own code. Your interceptor must implement one or more interface from the IInterceptor interface.

Interceptors that depend on DI services are not supported by the .NET Aspire `Add\<DatabaseSystem\>DbContext` methods. Use the EF Core AddDbContextPool method and call the AddInterceptors method in the options builder:

C#

```
builder.Services.AddDbContextPool<ExampleDbContext>((serviceProvider,
options) =>
    {

options.UseSqlServer(builder.Configuration.GetConnectionString("database"));

options.AddInterceptors(serviceProvider.GetRequiredService<ExampleIntercepto
r>());
    });

builder.EnrichSqlServerDbContext<ExampleDbContext>(
    configureSettings: settings =>
    {
        settings.DisableRetry = false;
        settings.CommandTimeout = 30; // seconds
    });
```

> ⓘ **Note**
>
> For more information about EF Core interceptors and their use, see Interceptors.

Use EF Core with dynamic connection strings in .NET Aspire

Most microservices always connect to the same database with the same credentials and other settings, so they always use the same connection string unless there's a major change in infrastructure. However, you may need to change the connection string for each request. For example:

- You might offer your service to multiple tenants and need to use a different database depending on which customer made the request.
- You might need to authenticate the request with a different database user account depending on which customer made the request.

For these requirements, you can use code to formulate a **dynamic connection string** and then use it to reach the database and run queries. However, this technique isn't supported by the .NET Aspire `Add\<DatabaseSystem\>DbContext` methods. Instead you must use the EF Core method to create the context and then enrich it:

C#

```
var connectionStringWithPlaceHolder =
builder.Configuration.GetConnectionString("database")
    ?? throw new InvalidOperationException("Connection string 'database' not
found.");

var connectionString = connectionStringWithPlaceHolder.Replace("
{DatabaseName}", "ContosoDatabase");

builder.Services.AddDbContext<ExampleDbContext>(options =>
    options.UseSqlServer(connectionString
        ?? throw new InvalidOperationException("Connection string 'database'
not found.")));

builder.EnrichSqlServerDbContext<ExampleDbContext>(
    configureSettings: settings =>
    {
        settings.DisableRetry = false;
        settings.CommandTimeout = 30; // seconds
    });
```

The above code replaces the place holder `{DatabaseName}` in the connection string with the string `ContosoDatabase`, at run time, before it creates the context and enriches it.

Use EF Core context factories in .NET Aspire

An EF Core context is an object designed to be used for a single unit of work. For example, if you want to add a new customer to the database, you might need to add a row in the **Customers** table and a row in the **Addresses** table. You should get the EF Core context, add the new customer and address entities to it, call SaveChangesAsync, and then dispose the context.

In many types of web application, such as ASP.NET applications, each HTTP request closely corresponds to a single unit of work against the database. If your .NET Aspire microservice is an ASP.NET application or a similar web application, you can use the standard EF Core AddDbContextPool method described above to register a context that is tied to the current HTTP request. Remember to call the .NET Aspire `Enrich\<DatabaseSystem\>DbContext` method to gain health checks, tracing, and other features. When you use this approach, the context lifetime is tied to the web request. You don't have to call the Dispose method when the unit of work is complete.

Other application types, such as ASP.NET Core Blazor, don't necessarily align each request with a unit of work, because they use dependency injection with a different service scope. In such apps, you may need to perform multiple units of work, each with a different context, within a single HTTP request and response. To implement this approach, you can register a context factory, by calling the EF Core AddPooledDbContextFactory method. This method also partners well with the .NET Aspire `Enrich\<DatabaseSystem\>DbContext` methods:

C#

```
builder.Services.AddPooledDbContextFactory<ExampleDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("database")
        ?? throw new InvalidOperationException("Connection string 'database'
not found.")));

builder.EnrichSqlServerDbContext<ExampleDbContext>(
    configureSettings: settings =>
    {
        settings.DisableRetry = false;
        settings.CommandTimeout = 30; // seconds
    });
```

Notice that the above code adds and enriches a *context factory* in the DI container. When you retreive this from the container, you must add a line of code to create a *context* from it:

C#

```
public class ExampleService(IDbContextFactory<ExampleDbContext>
contextFactory)
{
    using (var context = contextFactory.CreateDbContext())
    {
        // Use context...
    }
}
```

Contexts created from factories in this way aren't disposed of automatically because they aren't tied to an HTTP request lifetime. You must make sure your code disposes of them. In this example, the `using` code block ensures the disposal.

Use EF Core context pooling in .NET Aspire

In EF Core a context is relatively quick to create and dispose of so most applications can set them up as needed without impacting their performance. However, the overhead is not zero so, if your microservice intensively creates contexts, you may observe suboptimal performance. In such situations, consider using a context pool.

Context pooling is a feature of EF Core. Contexts are created as normal but, when you dispose of one, it isn't destroyed but reset and stored in a pool. The next time your code creates a context, the stored one is returned to avoid the extra overhead of creating a new one.

In a .NET Aspire consuming project, there are three ways to use context pooling:

- Use the .NET Aspire `Add\<DatabaseSystem\>DbContext` methods to create the context. These methods create a context pool automatically.
- Call the EF Core AddDbContextPool method instead of the EF Core AddDbContext method.

  C#

  ```
  builder.Services.AddDbContextPool<ExampleDbContext>(options =>

  options.UseSqlServer(builder.Configuration.GetConnectionString("databas
  e")
          ?? throw new InvalidOperationException("Connection string
  'database' not found.")));
  ```

- Call the EF Core AddPooledDbContextFactory method instead of the EF Core AddDbContextFactory method.

```
C#

builder.Services.AddPooledDbContextFactory<ExampleDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("databas
e")
        ?? throw new InvalidOperationException("Connection string
'database' not found.")));
```

Remember to enrich the context after using the last two methods, as described above.

> **ⓘ Important**
>
> Only the base context state is reset when it's returned to the pool. If you've manually changed the state of the `DbConnection` or another service, you must also manually reset it. Additionally, context pooling prevents you from using `OnConfiguring` to configure the context. See DbContext pooling for more information.

See also

- Entity Framework Core documentation hub
- Tutorial: Connect an ASP.NET Core app to SQL Server using .NET Aspire and Entity Framework Core
- Apply Entity Framework Core migrations in .NET Aspire
- DbContext Lifetime, Configuration, and Initialization
- Advanced Performance Topics
- Entity Framework Interceptors

Apply Entity Framework Core migrations in .NET Aspire

Article • 03/10/2025

Since .NET Aspire projects use a containerized architecture, databases are ephemeral and can be recreated at any time. Entity Framework Core (EF Core) uses a feature called migrations to create and update database schemas. Since databases are recreated when the app starts, you need to apply migrations to initialize the database schema each time your app starts. This is accomplished by registering a migration service project in your app that runs migrations during startup.

In this tutorial, you learn how to configure .NET Aspire projects to run EF Core migrations during app startup.

Prerequisites

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0
- An OCI compliant container runtime, such as:
  - Docker Desktop or Podman. For more information, see Container runtime.
- An Integrated Developer Environment (IDE) or code editor, such as:
  - Visual Studio 2022 version 17.9 or higher (Optional)
  - Visual Studio Code (Optional)
    - C# Dev Kit: Extension (Optional)
  - JetBrains Rider with .NET Aspire plugin (Optional)

For more information, see .NET Aspire setup and tooling, and .NET Aspire SDK.

Obtain the starter app

This tutorial uses a sample app that demonstrates how to apply EF Core migrations in .NET Aspire. Use Visual Studio to clone the sample app from GitHub or use the following command:

Bash

```
git clone https://github.com/MicrosoftDocs/aspire-docs-samples/
```

The sample app is in the *SupportTicketApi* folder. Open the solution in Visual Studio or VS Code and take a moment to review the sample app and make sure it runs before proceeding. The sample app is a rudimentary support ticket API, and it contains the following projects:

- **SupportTicketApi.Api**: The ASP.NET Core project that hosts the API.

- Runs EF Core migration command-line tool in the *SupportTicketApi.Api* directory. `dotnet ef` is run in this location because the API service is where the DB context is used.
- Creates a migration named *InitialCreate*.
- Creates the migration in the in the *Migrations* folder in the *SupportTicketApi.Data* project.

4. Modify the model so that it includes a new property. Open *SupportTicketApi.Data\Models\SupportTicket.cs* and add a new property to the `SupportTicket` class:

```
C#

public sealed class SupportTicket
{
    public int Id { get; set; }
    [Required]
    public string Title { get; set; } = string.Empty;
    [Required]
    public string Description { get; set; } = string.Empty;
    public bool Completed { get; set; }
}
```

5. Create another new migration to capture the changes to the model:

```
.NET CLI

dotnet ef migrations add AddCompleted --project
..\SupportTicketApi.Data\SupportTicketApi.Data.csproj
```

Now you've got some migrations to apply. Next, you'll create a migration service that applies these migrations during app startup.

Create the migration service

To run the migrations at startup, you need to create a service that applies the migrations.

1. Add a new Worker Service project to the solution. If using Visual Studio, right-click the solution in Solution Explorer and select **Add** > **New Project**. Select **Worker Service**, name the project *SupportTicketApi.MigrationService* and target **.NET 8.0**. If using the command line, use the following commands from the solution directory:

```
.NET CLI
```

```
dotnet new worker -n SupportTicketApi.MigrationService -f "net8.0"
dotnet sln add SupportTicketApi.MigrationService
```

2. Add the *SupportTicketApi.Data* and *SupportTicketApi.ServiceDefaults* project references to the *SupportTicketApi.MigrationService* project using Visual Studio or the command line:

.NET CLI

```
dotnet add SupportTicketApi.MigrationService reference
SupportTicketApi.Data
dotnet add SupportTicketApi.MigrationService reference
SupportTicketApi.ServiceDefaults
```

3. Add the Aspire.Microsoft.EntityFrameworkCore.SqlServer NuGet package reference to the *SupportTicketApi.MigrationService* project using Visual Studio or the command line:

.NET CLI

```
cd SupportTicketApi.MigrationService
dotnet add package Aspire.Microsoft.EntityFrameworkCore.SqlServer -v
"9.1.0"
```

4. Add the highlighted lines to the *Program.cs* file in the *SupportTicketApi.MigrationService* project:

C#

```
using SupportTicketApi.Data.Contexts;
using SupportTicketApi.MigrationService;

var builder = Host.CreateApplicationBuilder(args);

builder.AddServiceDefaults();

builder.Services.AddHostedService<Worker>();

builder.Services.AddOpenTelemetry()
    .WithTracing(tracing =>
tracing.AddSource(Worker.ActivitySourceName));
builder.AddSqlServerDbContext<TicketContext>("sqldata");

var host = builder.Build();
host.Run();
```

In the preceding code:

- The `AddServiceDefaults` extension method adds service defaults functionality.
- The `AddOpenTelemetry` extension method configures OpenTelemetry functionality.
- The `AddSqlServerDbContext` extension method adds the `TicketContext` service to the service collection. This service is used to run migrations and seed the database.

5. Replace the contents of the *Worker.cs* file in the *SupportTicketApi.MigrationService* project with the following code:

```
C#

using System.Diagnostics;

using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Storage;

using OpenTelemetry.Trace;

using SupportTicketApi.Data.Contexts;
using SupportTicketApi.Data.Models;

namespace SupportTicketApi.MigrationService;

public class Worker(
    IServiceProvider serviceProvider,
    IHostApplicationLifetime hostApplicationLifetime) :
BackgroundService
{
    public const string ActivitySourceName = "Migrations";
    private static readonly ActivitySource s_activitySource =
new(ActivitySourceName);

    protected override async Task ExecuteAsync(CancellationToken
cancellationToken)
    {
        using var activity = s_activitySource.StartActivity("Migrating
database", ActivityKind.Client);

        try
        {
            using var scope = serviceProvider.CreateScope();
            var dbContext =
scope.ServiceProvider.GetRequiredService<TicketContext>();

            await RunMigrationAsync(dbContext, cancellationToken);
            await SeedDataAsync(dbContext, cancellationToken);
```

```
        }
        catch (Exception ex)
        {
            activity?.RecordException(ex);
            throw;
        }

        hostApplicationLifetime.StopApplication();
    }

    private static async Task RunMigrationAsync(TicketContext
dbContext, CancellationToken cancellationToken)
    {
        var strategy = dbContext.Database.CreateExecutionStrategy();
        await strategy.ExecuteAsync(async () =>
        {
            // Run migration in a transaction to avoid partial
migration if it fails.
            await dbContext.Database.MigrateAsync(cancellationToken);
        });
    }

    private static async Task SeedDataAsync(TicketContext dbContext,
CancellationToken cancellationToken)
    {
        SupportTicket firstTicket = new()
        {
            Title = "Test Ticket",
            Description = "Default ticket, please ignore!",
            Completed = true
        };

        var strategy = dbContext.Database.CreateExecutionStrategy();
        await strategy.ExecuteAsync(async () =>
        {
            // Seed the database
            await using var transaction = await
dbContext.Database.BeginTransactionAsync(cancellationToken);
            await dbContext.Tickets.AddAsync(firstTicket,
cancellationToken);
            await dbContext.SaveChangesAsync(cancellationToken);
            await transaction.CommitAsync(cancellationToken);
        });
    }
}
```

In the preceding code:

- The `ExecuteAsync` method is called when the worker starts. It in turn performs the following steps:
  a. Gets a reference to the `TicketContext` service from the service provider.
  b. Calls `RunMigrationAsync` to apply any pending migrations.

c. Calls `SeedDataAsync` to seed the database with initial data.

d. Stops the worker with `StopApplication`.

- The `RunMigrationAsync` and `SeedDataAsync` methods both encapsulate their respective database operations using execution strategies to handle transient errors that may occur when interacting with the database. To learn more about execution strategies, see Connection Resiliency.

Add the migration service to the orchestrator

The migration service is created, but it needs to be added to the .NET Aspire app host so that it runs when the app starts.

1. In the *SupportTicketApi.AppHost* project, open the *Program.cs* file.

2. Add the following highlighted code to the `ConfigureServices` method:

   C#

   ```
   var builder = DistributedApplication.CreateBuilder(args);

   var sql = builder.AddSqlServer("sql", port: 14329)
                    .WithEndpoint(name: "sqlEndpoint", targetPort: 14330)
                    .AddDatabase("sqldata");

   builder.AddProject<Projects.SupportTicketApi_Api>("api")
       .WithReference(sql)
       .WaitFor(sql);

   builder.AddProject<Projects.SupportTicketApi_MigrationService>
   ("migrations")
       .WithReference(sql)
       .WaitFor(sql);

   builder.Build().Run();
   ```

   This enlists the *SupportTicketApi.MigrationService* project as a service in the .NET Aspire app host.

3. If the code cannot resolve the migration service project, add a reference to the migration service project in the AppHost project:

   .NET CLI

   ```
   dotnet add SupportTicketApi.AppHost reference
   SupportTicketApi.MigrationService
   ```

> **ⓘ Important**
>
> If you are using Visual Studio, and you selected the **Enlist in Aspire orchestration** option when creating the Worker Service project, similar code is added automatically with the service name `supporttickетapi-migrationservice`. Replace that code with the preceding code.

Remove existing seeding code

Since the migration service seeds the database, you should remove the existing data seeding code from the API project.

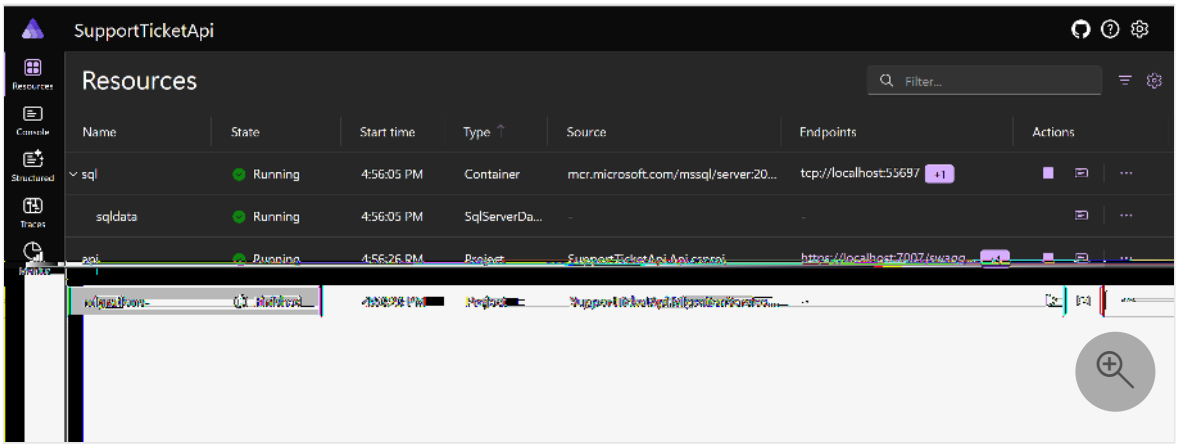
1. In the *SupportTicketApi.Api* project, open the *Program.cs* file.
2. Delete the highlighted lines.

```C#
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();

    using (var scope = app.Services.CreateScope())
    {
        var context =
scope.ServiceProvider.GetRequiredService<TicketContext>();
        context.Database.EnsureCreated();

        if(!context.Tickets.Any())
        {
            context.Tickets.Add(new SupportTicket { Title = "Initial
Ticket", Description = "Test ticket, please ignore." });
            context.SaveChanges();
        }
    }
}
```

Test the migration service

Now that the migration service is configured, run the app to test the migrations.

1. Run the app and observe the SupportTicketApi dashboard.
2. After a short wait, the `migrations` service state will display **Finished**.



3. Select the **Console logs** icon on the migration service to investigate the logs showing the SQL commands that were executed.

Get the code

You can find the completed sample app on GitHub.

More sample code

The Aspire Shop sample app uses this approach to apply migrations. See the `AspireShop.CatalogDbManager` project for the migration service implementation.

Seed data in a database using .NET Aspire

Article • 08/12/2024

In this article, you learn how to configure .NET Aspire projects to seed data in a database during app startup. .NET Aspire enables you to seed data using database scripts or Entity Framework Core for common platforms such as SQL Server, PostgreSQL and MySQL.

When to seed data

Seeding data pre-populates database tables with rows of data so they're ready for testing via your app. You may want to seed data for the following scenarios:

- Manually develop and test different features of your app against a meaningful set of data, such as a product catalog or list of customers.
- Run test suites to verify that features behave a specific way with a given set of data.

Manually seeding data is tedious and time consuming, so you should automate the process when possible. Use volumes to run database scripts for .NET Aspire projects during startup. You can also seed your database using tools like Entity Framework Core, which handles many underlying concerns for you.

Understand containerized databases

By default, .NET Aspire database integrations rely on containerized databases, which create the following challenges when trying to seed data:

- .NET Aspire destroys and recreates containers every time the app restarts, which means by default you have to re-seed your database every time the app restarts.
- Depending on your selected database technology, the new container instance may or may not create a default database, which means you might also have to create the database itself.
- Even if a default database exists, it most likely will not have the desired name or schema for your specific app.

.NET Aspire enables you to resolve these challenges using volumes and a few configurations to seed data effectively.

Seed data using volumes and SQL scripts

Volumes are the recommended way to automatically seed containerized databases when using SQL scripts. Volumes can store data for multiple containers at a time, offer high performance and are easy to back up or migrate. With .NET Aspire, you configure a volume for each resource container using the ContainerResourceBuilderExtensions.WithBindMount method, which accepts three parameters:

- **Source**: The source path of the volume mount, which is the physical location on your host.
- **Target**: The target path in the container of the data you want to persist.

Consider the following volume configuration code from a *Program.cs* file in a sample **AppHost** project:

C#

```
var todosDbName = "Todos";
var todosDb = builder.AddPostgres("postgres")
    .WithEnvironment("POSTGRES_DB", todosDbName)
    .WithBindMount(
        "../DatabaseContainers.ApiService/data/postgres",
        "/docker-entrypoint-initdb.d")
    .AddDatabase(todosDbName);
```

In this example, the `.WithBindMount` method parameters configure the following:

- `../DatabaseContainers.ApiService/data/postgres` sets a path to the SQL script in your local project that you want to run in the container to seed data.
- `/docker-entrypoint-initdb.d` sets the path to an entry point in the container so your script will be run during container startup.

The referenced SQL script located at `../DatabaseContainers.ApiService/data/postgres` creates and seeds a `Todos` table:

SQL

```
-- Postgres init script

-- Create the Todos table
CREATE TABLE IF NOT EXISTS Todos
(
    Id SERIAL PRIMARY KEY,
    Title text UNIQUE NOT NULL,
    IsComplete boolean NOT NULL DEFAULT false
```

```
);

-- Insert some sample data into the Todos table
INSERT INTO Todos (Title, IsComplete)
VALUES
    ('Give the dog a bath', false),
    ('Wash the dishes', false),
    ('Do the groceries', false)
ON CONFLICT DO NOTHING;
```

The script runs during startup every time a new container instance is created.

Database seeding examples

The following examples demonstrate how to seed data using SQL scripts and configurations applied using the `.WithBindMount` method for different database technologies:

> **Note**
>
> Visit the Database Container Sample App to view the full project and file structure for each database option.

SQL Server

The configuration code in the **.AppHost** *Program.cs* file mounts the required database files and folders and configures an entrypoint so that they run during startup.

C#

```
// SQL Server container is configured with an auto-generated password by
default
// but doesn't support any auto-creation of databases or running scripts
on startup so we have to do it manually.
var sqlserver = builder.AddSqlServer("sqlserver")
    // Mount the init scripts directory into the container.
    .WithBindMount("./sqlserverconfig", "/usr/config")
    // Mount the SQL scripts directory into the container so that the
init scripts run.
    .WithBindMount("../DatabaseContainers.ApiService/data/sqlserver",
"/docker-entrypoint-initdb.d")
    // Run the custom entrypoint script on startup.
    .WithEntrypoint("/usr/config/entrypoint.sh")
    // Configure the container to store data in a volume so that it
persists across instances.
```

```
    .WithDataVolume()
    // Keep the container running between app host sessions.
    .WithLifetime(ContainerLifetime.Persistent);
```

The *entrypoint.sh* script lives in the mounted `./sqlserverconfig` project folder and runs when the container starts. The script launches SQL Server and checks that it's running.

shell

```
#!/bin/bash

# Adapted from: https://github.com/microsoft/mssql-
docker/blob/80e2a51d0eb1693f2de014fb26d4a414f5a5add5/linux/preview/examp
les/mssql-customize/entrypoint.sh

# Start the script to create the DB and user
/usr/config/configure-db.sh &

# Start SQL Server
/opt/mssql/bin/sqlservr
```

The *init.sql* SQL script that lives in the mounted `../DatabaseContainers.ApiService/data/sqlserver` project folder creates the database and tables.

SQL

```
-- SQL Server init script

-- Create the AddressBook database
IF NOT EXISTS (SELECT * FROM sys.databases WHERE name = N'AddressBook')
BEGIN
  CREATE DATABASE AddressBook;
END;
GO

USE AddressBook;
GO

-- Create the Contacts table
IF OBJECT_ID(N'Contacts', N'U') IS NULL
BEGIN
    CREATE TABLE Contacts
    (
        Id        INT PRIMARY KEY IDENTITY(1,1) ,
        FirstName VARCHAR(255) NOT NULL,
        LastName  VARCHAR(255) NOT NULL,
        Email     VARCHAR(255) NULL,
        Phone     VARCHAR(255) NULL
```

```
    );
END;
GO

-- Ensure that either the Email or Phone column is populated
IF OBJECT_ID(N'chk_Contacts_Email_Phone', N'C') IS NULL
BEGIN
    ALTER TABLE Contacts
    ADD CONSTRAINT chk_Contacts_Email_Phone CHECK
    (
        Email IS NOT NULL OR Phone IS NOT NULL
    );
END;
GO

-- Insert some sample data into the Contacts table
IF (SELECT COUNT(*) FROM Contacts) = 0
BEGIN
    INSERT INTO Contacts (FirstName, LastName, Email, Phone)
    VALUES
        ('John', 'Doe', 'john.doe@example.com', '555-123-4567'),
        ('Jane', 'Doe', 'jane.doe@example.com', '555-234-5678');
END;
GO
```

Seed data using Entity Framework Core

You can also seed data in .NET Aspire projects using Entity Framework Core by explicitly running migrations during startup. Entity Framework Core handles underlying database connections and schema creation for you, which eliminates the need to use volumes or run SQL scripts during container startup.

> ⓘ **Important**
>
> These types of configurations should only be done during development, so make sure to add a conditional that checks your current environment context.

Add the following code to the *Program.cs* file of your **API Service** project.

SQL Server

C#

```
// Register DbContext class
builder.AddSqlServerDbContext<TicketContext>("sqldata");
```

```
var app = builder.Build();

app.MapDefaultEndpoints();

if (app.Environment.IsDevelopment())
{
    // Retrieve an instance of the DbContext class and manually run
migrations during startup
    using (var scope = app.Services.CreateScope())
    {
        var context =
scope.ServiceProvider.GetRequiredService<TicketContext>();
        context.Database.Migrate();
    }
}
```

Next steps

Database seeding is useful in a variety of app development scenarios. Try combining these techniques with the resource implementations demonstrated in the following tutorials:

- Tutorial: Connect an ASP.NET Core app to SQL Server using .NET Aspire and Entity Framework Core
- Tutorial: Connect an ASP.NET Core app to .NET Aspire storage integrations
- .NET Aspire orchestration overview

.NET Aspire Cosmos DB Entity Framework Core integration

Article • 02/26/2025

Includes: ✅ Hosting integration and ✅ Client integration

Azure Cosmos DB is a fully managed NoSQL database service for modern app development. The .NET Aspire Cosmos DB Entity Framework Core integration enables you to connect to existing Cosmos DB instances or create new instances from .NET with the Azure Cosmos DB emulator.

Hosting integration

The .NET Aspire Azure Cosmos DB hosting integration models the various Cosmos DB resources as the following types:

- AzureCosmosDBResource: Represents an Azure Cosmos DB resource.
- AzureCosmosDBEmulatorResource: Represents an Azure Cosmos DB emulator resource.

To access these types and APIs for expressing them, add the 📦 Aspire.Hosting.Azure.CosmosDB NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.CosmosDB
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Azure Cosmos DB resource

In your app host project, call AddAzureCosmosDB to add and return an Azure Cosmos DB resource builder.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db");

// After adding all resources, run the app...
```

When you add an AzureCosmosDBResource to the app host, it exposes other useful APIs to add databases and containers. In other words, you must add an `AzureCosmosDBResource` before adding any of the other Cosmos DB resources.

> **ⓘ Important**
>
> When you call AddAzureCosmosDB, it implicitly calls AddAzureProvisioning—which adds support for generating Azure resources dynamically during app startup. The app must configure the appropriate subscription and location. For more information, see Local provisioning: Configuration.

Generated provisioning Bicep

If you're new to Bicep, it's a domain-specific language for defining Azure resources. With .NET Aspire, you don't need to write Bicep by-hand, instead the provisioning APIs generate Bicep for you. When you publish your app, the generated Bicep is output alongside the manifest file. When you add an Azure Cosmos DB resource, the following Bicep is generated:

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location

param principalType string

param principalId string

resource cosmos 'Microsoft.DocumentDB/databaseAccounts@2024-08-15' = {
  name: take('cosmos-${uniqueString(resourceGroup().id)}', 44)
  location: location
  properties: {
    locations: [
      {
        locationName: location
        failoverPriority: 0
      }
    ]
    consistencyPolicy: {
```

```
      defaultConsistencyLevel: 'Session'
    }
    databaseAccountOfferType: 'Standard'
    disableLocalAuth: true
  }
  kind: 'GlobalDocumentDB'
  tags: {
    'aspire-resource-name': 'cosmos'
  }
}

resource cosmos_roleDefinition
'Microsoft.DocumentDB/databaseAccounts/sqlRoleDefinitions@2024-08-15'
existing = {
  name: '00000000-0000-0000-0000-000000000002'
  parent: cosmos
}

resource cosmos_roleAssignment
'Microsoft.DocumentDB/databaseAccounts/sqlRoleAssignments@2024-08-15' = {
  name: guid(principalId, cosmos_roleDefinition.id, cosmos.id)
  properties: {
    principalId: principalId
    roleDefinitionId: cosmos_roleDefinition.id
    scope: cosmos.id
  }
  parent: cosmos
}

output connectionString string = cosmos.properties.documentEndpoint
```

The preceding Bicep is a module that provisions an Azure Cosmos DB account with the following defaults:

- `kind`: The kind of Cosmos DB account. The default is `GlobalDocumentDB`.
- `consistencyPolicy`: The consistency policy of the Cosmos DB account. The default is `Session`.
- `locations`: The locations for the Cosmos DB account. The default is the resource group's location.

In addition to the Cosmos DB account, it also adds the current application to the `Data Contributor` role for the Cosmos DB account. The generated Bicep is a starting point and is influenced by changes to the provisioning infrastructure in C#. Customizations to the Bicep file directly will be overwritten, so make changes through the C# provisioning APIs to ensure they are reflected in the generated files.

Customize provisioning infrastructure

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This type enables the customization of the generated Bicep by providing a fluent API to configure the Azure resources—using the ConfigureInfrastructure<T>(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API. For example, you can configure the `kind`, `consistencyPolicy`, `locations`, and more. The following example demonstrates how to customize the Azure Cosmos DB resource:

C#

```
builder.AddAzureCosmosDB("cosmos-db")
    .ConfigureInfrastructure(infra =>
    {
        var cosmosDbAccount = infra.GetProvisionableResources()
                                   .OfType<CosmosDBAccount>()
                                   .Single();

        cosmosDbAccount.Kind = CosmosDBAccountKind.MongoDB;
        cosmosDbAccount.ConsistencyPolicy = new()
        {
            DefaultConsistencyLevel = DefaultConsistencyLevel.Strong,
        };
        cosmosDbAccount.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
  - The `infra` parameter is an instance of the AzureResourceInfrastructure type.
  - The provisionable resources are retrieved by calling the GetProvisionableResources() method.
  - The single CosmosDBAccount is retrieved.
  - The CosmosDBAccount.ConsistencyPolicy is assigned to a DefaultConsistencyLevel.Strong.
  - A tag is added to the Cosmos DB account with a key of `ExampleKey` and a value of `Example value`.

There are many more configuration options available to customize the Azure Cosmos DB resource. For more information, see Azure.Provisioning.CosmosDB. For more information, see Azure.Provisioning customization.

Connect to an existing Azure Cosmos DB account

You might have an existing Azure Cosmos DB account that you want to connect to. Instead of representing a new Azure Cosmos DB resource, you can add a connection

string to the app host. To add a connection to an existing Azure Cosmos DB account, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddConnectionString("cosmos-db");

builder.AddProject<Projects.WebApplication>("web")
       .WithReference(cosmos);

// After adding all resources, run the app...
```

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section. The app host injects this connection string as an environment variable into all dependent resources, for example:

JSON

```
{
    "ConnectionStrings": {
        "cosmos-db":
"AccountEndpoint=https://{account_name}.documents.azure.com:443/;AccountKey=
{account_key};"
    }
}
```

The dependent resource can access the injected connection string by calling the GetConnectionString method, and passing the connection name as the parameter, in this case `"cosmos-db"`. The `GetConnectionString` API is shorthand for `IConfiguration.GetSection("ConnectionStrings")[name]`.

Add Azure Cosmos DB database and container resources

To add an Azure Cosmos DB database resource, call the AddCosmosDatabase method on an `IResourceBuilder<AzureCosmosDBResource>` instance:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db");
cosmos.AddCosmosDatabase("db");

// After adding all resources, run the app...
```

When you call `AddCosmosDatabase`, it adds a database named `db` to your Cosmos DB resources and returns the newly created database resource. The database is created in the Cosmos DB account that's represented by the `AzureCosmosDBResource` that you added earlier. The database is a logical container for collections and users.

An Azure Cosmos DB container is where data is stored. When you create a container, you need to supply a partition key.

To add an Azure Cosmos DB container resource, call the AddContainer method on an `IResourceBuilder<AzureCosmosDBDatabaseResource>` instance:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db");
var db = cosmos.AddCosmosDatabase("db");
db.AddContainer("entries", "/id");

// After adding all resources, run the app...
```

The container is created in the database that's represented by the `AzureCosmosDBDatabaseResource` that you added earlier.

For more information, see Databases, containers, and items in Azure Cosmos DB.

Add Azure Cosmos DB emulator resource

To add an Azure Cosmos DB emulator resource, chain a call on an `IResourceBuilder<AzureCosmosDBResource>` to the RunAsEmulator API:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db")
                    .RunAsEmulator();
```

```
// After adding all resources, run the app...
```

When you call `RunAsEmulator`, it configures your Cosmos DB resources to run locally using an emulator. The emulator in this case is the Azure Cosmos DB Emulator. The Azure Cosmos DB Emulator provides a free local environment for testing your Azure Cosmos DB apps and it's a perfect companion to the .NET Aspire Azure hosting integration. The emulator isn't installed, instead, it's accessible to .NET Aspire as a container. When you add a container to the app host, as shown in the preceding example with the `mcr.microsoft.com/cosmosdb/emulator` image, it creates and starts the container when the app host starts. For more information, see Container resource lifecycle.

Configure Cosmos DB emulator container

There are various configurations available to container resources, for example, you can configure the container's ports, environment variables, it's lifetime, and more.

Configure Cosmos DB emulator container gateway port

By default, the Cosmos DB emulator container when configured by .NET Aspire, exposes the following endpoints:

Expand table

Endpoint	Container port	Host port
`https`	8081	dynamic

The port that it's listening on is dynamic by default. When the container starts, the port is mapped to a random port on the host machine. To configure the endpoint port, chain calls on the container resource builder provided by the `RunAsEmulator` method as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db").RunAsEmulator(
                     emulator =>
                     {
                         emulator.WithGatewayPort(7777);
                     });
```

```
// After adding all resources, run the app...
```

The preceding code configures the Cosmos DB emulator container's existing `https` endpoint to listen on port `8081`. The Cosmos DB emulator container's port is mapped to the host port as shown in the following table:

Expand table

Endpoint name	Port mapping (`container:host`)
`https`	`8081:7777`

Configure Cosmos DB emulator container with persistent lifetime

To configure the Cosmos DB emulator container with a persistent lifetime, call the WithLifetime method on the Cosmos DB emulator container resource and pass ContainerLifetime.Persistent:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db").RunAsEmulator(
                     emulator =>
                     {

emulator.WithLifetime(ContainerLifetime.Persistent);
                     });

// After adding all resources, run the app...
```

For more information, see Container resource lifetime.

Configure Cosmos DB emulator container with data volume

To add a data volume to the Azure Cosmos DB emulator resource, call the WithDataVolume method on the Azure Cosmos DB emulator resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db").RunAsEmulator(
                     emulator =>
                     {
```

```
                        emulator.WithDataVolume();
                    });

// After adding all resources, run the app...
```

The data volume is used to persist the Cosmos DB emulator data outside the lifecycle of its container. The data volume is mounted at the `/tmp/cosmos/appdata` path in the Cosmos DB emulator container and when a `name` parameter isn't provided, the name is generated. The emulator has its `AZURE_COSMOS_EMULATOR_ENABLE_DATA_PERSISTENCE` environment variable set to `true`. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Configure Cosmos DB emulator container partition count

To configure the partition count of the Cosmos DB emulator container, call the WithPartitionCount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db").RunAsEmulator(
                    emulator =>
                    {
                        emulator.WithPartitionCount(100); // Defaults to 25
                    });

// After adding all resources, run the app...
```

The preceding code configures the Cosmos DB emulator container to have a partition count of `100`. This is a shorthand for setting the `AZURE_COSMOS_EMULATOR_PARTITION_COUNT` environment variable.

Use Linux-based emulator (preview)

The next generation of the Azure Cosmos DB emulator is entirely Linux-based and is available as a Docker container. It supports running on a wide variety of processors and operating systems.

To use the preview version of the Cosmos DB emulator, call the RunAsPreviewEmulator method. Since this feature is in preview, you need to explicitly opt into the preview feature by suppressing the `ASPIRECOSMOSDB001` experimental diagnostic.

The preview emulator also supports exposing a "Data Explorer" endpoint which allows you to view the data stored in the Cosmos DB emulator via a web UI. To enable the Data Explorer, call the WithDataExplorer method.

C#

```
#pragma warning disable ASPIRECOSMOSDB001

var builder = DistributedApplication.CreateBuilder(args);

var cosmos = builder.AddAzureCosmosDB("cosmos-db").RunAsPreviewEmulator(
                     emulator =>
                     {
                         emulator.WithDataExplorer();
                     });

// After adding all resources, run the app...
```

The preceding code configures the Linux-based preview Cosmos DB emulator container, with the Data Explorer endpoint, to use at run time.

Hosting integration health checks

The Azure Cosmos DB hosting integration automatically adds a health check for the Cosmos DB resource. The health check verifies that the Cosmos DB is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.CosmosDb NuGet package.

Client integration

To get started with the .NET Aspire Microsoft Entity Framework Core Cosmos DB integration, install the Aspire.Microsoft.EntityFrameworkCore.Cosmos NuGet package in the client-consuming project, i.e., the project for the application that uses the Microsoft Entity Framework Core Cosmos DB client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Microsoft.EntityFrameworkCore.Cosmos
```

Add Cosmos DB context

In the Program.cs file of your client-consuming project, call the AddCosmosDbContext extension method to register a System.Data.Entity.DbContext for use via the dependency injection container. The method takes a connection name parameter.

```
C#

builder.AddCosmosDbContext<MyDbContext>("cosmosdb", "databaseName");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Cosmos DB resource in the app host project. In other words, when you call `AddAzureCosmosDB` and provide a name of `cosmosdb` that same name should be used when calling `AddCosmosDbContext`. For more information, see **Add Azure Cosmos DB resource**.

You can then retrieve the DbContext instance using dependency injection. For example, to retrieve the client from a service:

```
C#

public class ExampleService(MyDbContext context)
{
    // Use context...
}
```

For more information on using Entity Framework Core with Azure Cosmos DB, see the Examples for Azure Cosmos DB for NoSQL SDK for .NET.

Configuration

The .NET Aspire Microsoft Entity Framework Core Cosmos DB integration provides multiple options to configure the Azure Cosmos DB connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

`builder.AddCosmosDbContext`:

C#

```
builder.AddCosmosDbContext<MyDbContext>("CosmosConnection");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "CosmosConnection":
"AccountEndpoint=https://{account_name}.documents.azure.com:443/;AccountKey=
{account_key};"
  }
}
```

For more information, see the ConnectionString documentation.

Use configuration providers

The .NET Aspire Microsoft Entity Framework Core Cosmos DB integration supports Microsoft.Extensions.Configuration. It loads the EntityFrameworkCoreCosmosSettings from configuration files such as *appsettings.json*. Example _appsettings.json that configures some of the options:

JSON

```
{
  "Aspire": {
    "Microsoft": {
      "EntityFrameworkCore": {
        "Cosmos": {
          "DisableTracing": true
        }
      }
    }
  }
}
```

For the complete Cosmos DB client integration JSON schema, see Aspire.Microsoft.EntityFrameworkCore.Cosmos/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<EntityFrameworkCoreCosmosSettings> configureSettings` delegate to set up some or all the EntityFrameworkCoreCosmosSettings options inline, for example to disable tracing from code:

C#

```
builder.AddCosmosDbContext<MyDbContext>(
    "cosmosdb",
    settings => settings.DisableTracing = true);
```

Client integration health checks

The .NET Aspire Microsoft Entity Framework Core Cosmos DB integration currently doesn't implement health checks, though this may change in future releases.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Microsoft Entity Framework Core Cosmos DB integration uses the following log categories:

- `Azure-Cosmos-Operation-Request-Diagnostics`
- `Microsoft.EntityFrameworkCore.ChangeTracking`
- `Microsoft.EntityFrameworkCore.Database.Command`
- `Microsoft.EntityFrameworkCore.Infrastructure`
- `Microsoft.EntityFrameworkCore.Query`

Tracing

The .NET Aspire Microsoft Entity Framework Core Cosmos DB integration will emit the following tracing activities using OpenTelemetry:

- `Azure.Cosmos.Operation`
- `OpenTelemetry.Instrumentation.EntityFrameworkCore`

Metrics

The .NET Aspire Microsoft Entity Framework Core Cosmos DB integration currently supports the following metrics:

- `Microsoft.EntityFrameworkCore`
  - `ec_Microsoft_EntityFrameworkCore_active_db_contexts`
  - `ec_Microsoft_EntityFrameworkCore_total_queries`
  - `ec_Microsoft_EntityFrameworkCore_queries_per_second`
  - `ec_Microsoft_EntityFrameworkCore_total_save_changes`
  - `ec_Microsoft_EntityFrameworkCore_save_changes_per_second`
  - `ec_Microsoft_EntityFrameworkCore_compiled_query_cache_hit_rate`
  - `ec_Microsoft_Entity_total_execution_strategy_operation_failures`
  - `ec_Microsoft_E_execution_strategy_operation_failures_per_second`
  - `ec_Microsoft_EntityFramew_total_optimistic_concurrency_failures`
  - `ec_Microsoft_EntityF_optimistic_concurrency_failures_per_second`

See also

- Azure Cosmos DB docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire Azure PostgreSQL Entity Framework Core integration

Article • 01/31/2025

Includes: ✅ Hosting integration and ✅ Client integration

Azure Database for PostgreSQL—Flexible Server is a relational database service based on the open-source Postgres database engine. It's a fully managed database-as-a-service that can handle mission-critical workloads with predictable performance, security, high availability, and dynamic scalability. The .NET Aspire Azure PostgreSQL integration provides a way to connect to existing Azure PostgreSQL databases, or create new instances from .NET with the docker.io/library/postgres container image.

Hosting integration

The .NET Aspire Azure PostgreSQL hosting integration models a PostgreSQL flexible server and database as the AzurePostgresFlexibleServerResource and AzurePostgresFlexibleServerDatabaseResource types. Other types that are inherently available in the hosting integration are represented in the following resources:

- PostgresServerResource
- PostgresDatabaseResource
- PgAdminContainerResource
- PgWebContainerResource

To access these types and APIs for expressing them as resources in your app host project, install the Aspire.Hosting.Azure.PostgreSQL NuGet package:

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.PostgreSQL
```

For more information, see dotnet add package.

The Azure PostgreSQL hosting integration takes a dependency on the Aspire.Hosting.PostgreSQL NuGet package, extending it to support Azure. Everything that you can do with the .NET Aspire PostgreSQL integration and .NET Aspire PostgreSQL Entity Framework Core integration you can also do with this integration.

Add Azure PostgreSQL server resource

After you've installed the .NET Aspire Azure PostgreSQL hosting integration, call the AddAzurePostgresFlexibleServer extension method in your app host project:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddAzurePostgresFlexibleServer("postgres");
var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);
```

The preceding call to `AddAzurePostgresFlexibleServer` configures the PostgresSQL server resource to be deployed as an Azure Postgres Flexible Server.

> **Important**
>
> By default, `AddAzurePostgresFlexibleServer` configures Microsoft Entra ID authentication. This requires changes to applications that need to connect to these resources. For more information, see Client integration.

> **Tip**
>
> When you call AddAzurePostgresFlexibleServer, it implicitly calls AddAzureProvisioning—which adds support for generating Azure resources dynamically during app startup. The app must configure the appropriate subscription and location. For more information, see Local provisioning: Configuration.

Generated provisioning Bicep

If you're new to Bicep, it's a domain-specific language for defining Azure resources. With .NET Aspire, you don't need to write Bicep by hand, because the provisioning APIs generate Bicep for you. When you publish your app, the generated Bicep is output alongside the manifest file. When you add an Azure PostgreSQL resource, the following Bicep is generated:

Bicep

```
@description('The location for the resource(s) to be deployed.')
param location string = resourceGroup().location

param principalId string

param principalType string

param principalName string

resource postgres_flexible 'Microsoft.DBforPostgreSQL/flexibleServers@2024-
08-01' = {
  name: take('postgresflexible-${uniqueString(resourceGroup().id)}', 63)
  location: location
  properties: {
    authConfig: {
      activeDirectoryAuth: 'Enabled'
      passwordAuth: 'Disabled'
    }
    availabilityZone: '1'
    backup: {
      backupRetentionDays: 7
      geoRedundantBackup: 'Disabled'
    }
    highAvailability: {
      mode: 'Disabled'
    }
    storage: {
      storageSizeGB: 32
    }
    version: '16'
  }
  sku: {
    name: 'Standard_B1ms'
    tier: 'Burstable'
  }
  tags: {
    'aspire-resource-name': 'postgres-flexible'
  }
}

resource postgreSqlFirewallRule_AllowAllAzureIps
'Microsoft.DBforPostgreSQL/flexibleServers/firewallRules@2024-08-01' = {
  name: 'AllowAllAzureIps'
  properties: {
    endIpAddress: '0.0.0.0'
    startIpAddress: '0.0.0.0'
  }
  parent: postgres_flexible
}

resource postgres_flexible_admin
'Microsoft.DBforPostgreSQL/flexibleServers/administrators@2024-08-01' = {
  name: principalId
  properties: {
```

```
    principalName: principalName
    principalType: principalType
  }
  parent: postgres_flexible
  dependsOn: [
    postgres_flexible
    postgreSqlFirewallRule_AllowAllAzureIps
  ]
}

output connectionString string =
'Host=${postgres_flexible.properties.fullyQualifiedDomainName};Username=${pr
incipalName}'
```

The preceding Bicep is a module that provisions an Azure PostgreSQL flexible server with the following defaults:

- `authConfig`: The authentication configuration of the PostgreSQL server. The default is `ActiveDirectoryAuth` enabled and `PasswordAuth` disabled.
- `availabilityZone`: The availability zone of the PostgreSQL server. The default is `1`.
- `backup`: The backup configuration of the PostgreSQL server. The default is `BackupRetentionDays` set to `7` and `GeoRedundantBackup` set to `Disabled`.
- `highAvailability`: The high availability configuration of the PostgreSQL server. The default is `Disabled`.
- `storage`: The storage configuration of the PostgreSQL server. The default is `StorageSizeGB` set to `32`.
- `version`: The version of the PostgreSQL server. The default is `16`.
- `sku`: The SKU of the PostgreSQL server. The default is `Standard_B1ms`.
- `tags`: The tags of the PostgreSQL server. The default is `aspire-resource-name` set to the name of the Aspire resource, in this case `postgres-flexible`.

In addition to the PostgreSQL flexible server, it also provisions an Azure Firewall rule to allow all Azure IP addresses. Finally, an administrator is created for the PostgreSQL server, and the connection string is outputted as an output variable. The generated Bicep is a starting point and is influenced by changes to the provisioning infrastructure in C#. Customizations to the Bicep file directly will be overwritten, so make changes through the C# provisioning APIs to ensure they are reflected in the generated files.

Customize provisioning infrastructure

All .NET Aspire Azure resources are subclasses of the AzureProvisioningResource type. This type enables the customization of the generated Bicep by providing a fluent API to configure the Azure resources by using the ConfigureInfrastructure<T>

(IResourceBuilder<T>, Action<AzureResourceInfrastructure>) API. For example, you can configure the `kind`, `consistencyPolicy`, `locations`, and more. The following example demonstrates how to customize the PostgreSQL server resource:

C#

```
builder.AddAzurePostgresFlexibleServer("postgres")
    .ConfigureInfrastructure(infra =>
    {
        var flexibleServer = infra.GetProvisionableResources()
                                  .OfType<PostgreSqlFlexibleServer>()
                                  .Single();

        flexibleServer.Sku = new PostgreSqlFlexibleServerSku
        {
            Tier = PostgreSqlFlexibleServerSkuTier.Burstable,
        };
        flexibleServer.HighAvailability = new
PostgreSqlFlexibleServerHighAvailability
        {
            Mode =
PostgreSqlFlexibleServerHighAvailabilityMode.ZoneRedundant,
            StandbyAvailabilityZone = "2",
        };
        flexibleServer.Tags.Add("ExampleKey", "Example value");
    });
```

The preceding code:

- Chains a call to the ConfigureInfrastructure API:
  - The `infra` parameter is an instance of the AzureResourceInfrastructure type.
  - The provisionable resources are retrieved by calling the GetProvisionableResources() method.
  - The single PostgreSqlFlexibleServer is retrieved.
  - The `sku` is set with PostgreSqlFlexibleServerSkuTier.Burstable.
  - The high availability properties are set with PostgreSqlFlexibleServerHighAvailabilityMode.ZoneRedundant in standby availability zone `"2"`.
  - A tag is added to the flexible server with a key of `ExampleKey` and a value of `Example value`.

There are many more configuration options available to customize the PostgreSQL flexible server resource. For more information, see Azure.Provisioning.PostgreSql and Azure.Provisioning customization.

Connect to an existing Azure PostgreSQL flexible server

You might have an existing Azure PostgreSQL flexible server that you want to connect to. Instead of representing a new Azure PostgreSQL flexible server resource, you can add a connection string to the app host. To add a connection to an existing Azure PostgreSQL flexible server, call the AddConnectionString method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddConnectionString("postgres");

builder.AddProject<Projects.WebApplication>("web")
       .WithReference(postgres);

// After adding all resources, run the app...
```

> **Note**
>
> Connection strings are used to represent a wide range of connection information, including database connections, message brokers, endpoint URIs, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

The connection string is configured in the app host's configuration, typically under User Secrets, under the `ConnectionStrings` section. The app host injects this connection string as an environment variable into all dependent resources, for example:

JSON

```
{
    "ConnectionStrings": {
        "postgres": "Server=<PostgreSQL-server-
name>.postgres.database.azure.com;Database=<database-name>;Port=5432;Ssl
Mode=Require;User Id=<username>;"
    }
}
```

The dependent resource can access the injected connection string by calling the GetConnectionString method, and passing the connection name as the parameter, in this case `"postgres"`. The `GetConnectionString` API is shorthand for `IConfiguration.GetSection("ConnectionStrings")[name]`.

Run Azure PostgreSQL resource as a container

The Azure PostgreSQL hosting integration supports running the PostgreSQL server as a local container. This is beneficial for situations where you want to run the PostgreSQL server locally for development and testing purposes, avoiding the need to provision an Azure resource or connect to an existing Azure PostgreSQL server.

To run the PostgreSQL server as a container, call the RunAsContainer method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddAzurePostgresFlexibleServer("postgres")
                      .RunAsContainer();

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);
```

The preceding code configures an Azure PostgreSQL Flexible Server resource to run locally in a container.

> **Tip**
>
> The `RunAsContainer` method is useful for local development and testing. The API exposes an optional delegate that enables you to customize the underlying PostgresServerResource configuration. For example, you can add pgAdmin and pgWeb, add a data volume or data bind mount, and add an init bind mount. For more information, see the .NET Aspire PostgreSQL hosting integration section.

Configure the Azure PostgreSQL server to use password authentication

By default, the Azure PostgreSQL server is configured to use Microsoft Entra ID authentication. If you want to use password authentication, you can configure the server to use password authentication by calling the WithPasswordAuthentication method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var username = builder.AddParameter("username", secret: true);
var password = builder.AddParameter("password", secret: true);
```

```
var postgres = builder.AddAzurePostgresFlexibleServer("postgres")
                      .WithPasswordAuthentication(username, password);

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);
```

The preceding code configures the Azure PostgreSQL server to use password authentication. The `username` and `password` parameters are added to the app host as parameters, and the `WithPasswordAuthentication` method is called to configure the Azure PostgreSQL server to use password authentication. For more information, see External parameters.

Client integration

To get started with the .NET Aspire PostgreSQL Entity Framework Core client integration, install the Aspire.Npgsql.EntityFrameworkCore.PostgreSQL NuGet package in the client-consuming project, that is, the project for the application that uses the PostgreSQL client. The .NET Aspire PostgreSQL Entity Framework Core client integration registers your desired `DbContext` subclass instances that you can use to interact with PostgreSQL.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Npgsql.EntityFrameworkCore.PostgreSQL
```

Add Npgsql database context

In the *Program.cs* file of your client-consuming project, call the AddNpgsqlDbContext extension method on any IHostApplicationBuilder to register your DbContext subclass for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddNpgsqlDbContext<YourDbContext>(connectionName: "postgresdb");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the PostgreSQL server resource in the app host project. For more information, see Add PostgreSQL server resource.

After adding `YourDbContext` to the builder, you can get the `YourDbContext` instance using dependency injection. For example, to retrieve your data source object from an example service define it as a constructor parameter and ensure the `ExampleService` class is registered with the dependency injection container:

```
C#

public class ExampleService(YourDbContext context)
{
    // Use context...
}
```

For more information on dependency injection, see .NET dependency injection.

Enrich an Npgsql database context

You may prefer to use the standard Entity Framework method to obtain a database context and add it to the dependency injection container:

```
C#

builder.Services.AddDbContext<YourDbContext>(options =>

options.UseNpgsql(builder.Configuration.GetConnectionString("postgresdb")
        ?? throw new InvalidOperationException("Connection string
'postgresdb' not found.")));
```

> **Note**
>
> The connection string name that you pass to the GetConnectionString method must match the name used when adding the PostgreSQL server resource in the app host project. For more information, see Add PostgreSQL server resource.

You have more flexibility when you create the database context in this way, for example:

- You can reuse existing configuration code for the database context without rewriting it for .NET Aspire.
- You can use Entity Framework Core interceptors to modify database operations.
- You can choose not to use Entity Framework Core context pooling, which may perform better in some circumstances.

If you use this method, you can enhance the database context with .NET Aspire-style retries, health checks, logging, and telemetry features by calling the EnrichNpgsqlDbContext method:

C#

```
builder.EnrichNpgsqlDbContext<YourDbContext>(
    configureSettings: settings =>
    {
        settings.DisableRetry = false;
        settings.CommandTimeout = 30;
    });
```

The `settings` parameter is an instance of the NpgsqlEntityFrameworkCorePostgreSQLSettings class.

Configuration

The .NET Aspire PostgreSQL Entity Framework Core integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you provide the name of the connection string when calling the AddNpgsqlDbContext method:

C#

```
builder.AddNpgsqlDbContext<MyDbContext>("pgdb");
```

The connection string is retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
```

```
    "pgdb": "Host=myserver;Database=test"
  }
}
```

The `EnrichNpgsqlDbContext` won't make use of the `ConnectionStrings` configuration section since it expects a `DbContext` to be registered at the point it's called.

For more information, see the ConnectionString.

Use configuration providers

The .NET Aspire PostgreSQL Entity Framework Core integration supports Microsoft.Extensions.Configuration. It loads the NpgsqlEntityFrameworkCorePostgreSQLSettings from configuration files such as *appsettings.json* by using the `Aspire:Npgsql:EntityFrameworkCore:PostgreSQL` key. If you have set up your configurations in the `Aspire:Npgsql:EntityFrameworkCore:PostgreSQL` section you can just call the method without passing any parameter.

The following example shows an *appsettings.json* file that configures some of the available options:

JSON

```
{
  "Aspire": {
    "Npgsql": {
      "EntityFrameworkCore": {
        "PostgreSQL": {
          "ConnectionString": "Host=myserver;Database=postgresdb",
          "DisableHealthChecks": true,
          "DisableTracing": true
        }
      }
    }
  }
}
```

For the complete PostgreSQL Entity Framework Core client integration JSON schema, see Aspire.Npgsql.EntityFrameworkCore.PostgreSQL/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<NpgsqlEntityFrameworkCorePostgreSQLSettings>` delegate to set up some or all the options inline, for example to set the `ConnectionString`:

C#

```
builder.AddNpgsqlDbContext<YourDbContext>(
    "pgdb",
    static settings => settings.ConnectionString = "<YOUR CONNECTION
STRING>");
```

Configure multiple DbContext classes

If you want to register more than one DbContext with different configuration, you can use `$"Aspire:Npgsql:EntityFrameworkCore:PostgreSQL:{typeof(TContext).Name}"` configuration section name. The json configuration would look like:

JSON

```
{
  "Aspire": {
    "Npgsql": {
      "EntityFrameworkCore": {
        "PostgreSQL": {
          "ConnectionString": "<YOUR CONNECTION STRING>",
          "DisableHealthChecks": true,
          "DisableTracing": true,
          "AnotherDbContext": {
            "ConnectionString": "<ANOTHER CONNECTION STRING>",
            "DisableTracing": false
          }
        }
      }
    }
  }
}
```

Then calling the AddNpgsqlDbContext method with `AnotherDbContext` type parameter would load the settings from `Aspire:Npgsql:EntityFrameworkCore:PostgreSQL:AnotherDbContext` section.

C#

```
builder.AddNpgsqlDbContext<AnotherDbContext>();
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For

more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

By default, the .NET Aspire PostgreSQL Entity Framework Core integrations handles the following:

- Adds the DbContextHealthCheck, which calls EF Core's CanConnectAsync method. The name of the health check is the name of the `TContext` type.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire PostgreSQL Entity Framework Core integration uses the following Log categories:

- `Microsoft.EntityFrameworkCore.ChangeTracking`
- `Microsoft.EntityFrameworkCore.Database.Command`
- `Microsoft.EntityFrameworkCore.Database.Connection`
- `Microsoft.EntityFrameworkCore.Database.Transaction`
- `Microsoft.EntityFrameworkCore.Migrations`
- `Microsoft.EntityFrameworkCore.Infrastructure`
- `Microsoft.EntityFrameworkCore.Migrations`
- `Microsoft.EntityFrameworkCore.Model`
- `Microsoft.EntityFrameworkCore.Model.Validation`
- `Microsoft.EntityFrameworkCore.Query`
- `Microsoft.EntityFrameworkCore.Update`

Tracing

The .NET Aspire PostgreSQL Entity Framework Core integration will emit the following tracing activities using OpenTelemetry:

- `Npgsql`

Metrics

The .NET Aspire PostgreSQL Entity Framework Core integration will emit the following metrics using OpenTelemetry:

- Microsoft.EntityFrameworkCore:
  - `ec_Microsoft_EntityFrameworkCore_active_db_contexts`
  - `ec_Microsoft_EntityFrameworkCore_total_queries`
  - `ec_Microsoft_EntityFrameworkCore_queries_per_second`
  - `ec_Microsoft_EntityFrameworkCore_total_save_changes`
  - `ec_Microsoft_EntityFrameworkCore_save_changes_per_second`
  - `ec_Microsoft_EntityFrameworkCore_compiled_query_cache_hit_rate`
  - `ec_Microsoft_Entity_total_execution_strategy_operation_failures`
  - `ec_Microsoft_E_execution_strategy_operation_failures_per_second`
  - `ec_Microsoft_EntityFramew_total_optimistic_concurrency_failures`
  - `ec_Microsoft_EntityF_optimistic_concurrency_failures_per_second`
- Npgsql:
  - `ec_Npgsql_bytes_written_per_second`
  - `ec_Npgsql_bytes_read_per_second`
  - `ec_Npgsql_commands_per_second`
  - `ec_Npgsql_total_commands`
  - `ec_Npgsql_current_commands`
  - `ec_Npgsql_failed_commands`
  - `ec_Npgsql_prepared_commands_ratio`
  - `ec_Npgsql_connection_pools`
  - `ec_Npgsql_multiplexing_average_commands_per_batch`
  - `ec_Npgsql_multiplexing_average_write_time_per_batch`

Add Azure authenticated Npgsql client

By default, when you call `AddAzurePostgresFlexibleServer` in your PostgreSQL hosting integration, it requires Azure.Identity NuGet package to enable authentication:

.NET CLI

```
dotnet add package Azure.Identity
```

The PostgreSQL connection can be consumed using the client integration and Azure.Identity.

The following code snippets demonstrate how to use the DefaultAzureCredential class from the Azure.Identity package to authenticate with Microsoft Entra ID and retrieve a token to connect to the PostgreSQL database. The UsePasswordProvider method is used to provide the token to the data source builder.

EF Core version 8

C#

```
var dsBuilder = new
NpgsqlDataSourceBuilder(builder.Configuration.GetConnectionString("postgresd
b"));
if (string.IsNullOrEmpty(dsBuilder.ConnectionStringBuilder.Password))
{
    var credentials = new DefaultAzureCredential();
    var tokenRequest = new TokenRequestContext(["https://ossrdbms-
aad.database.windows.net/.default"]);

    dsBuilder.UsePasswordProvider(
        passwordProvider: _ => credentials.GetToken(tokenRequest).Token,
        passwordProviderAsync: async (_, ct) => (await
credentials.GetTokenAsync(tokenRequest, ct)).Token);
}

builder.AddNpgsqlDbContext<MyDb1Context>(
    "postgresdb",
    configureDbContextOptions: (options) =>
options.UseNpgsql(dsBuilder.Build()));
```

EF Core version 9+

With EF Core version 9, you can use the `ConfigureDataSource` method to configure the `NpgsqlDataSourceBuilder` that's used by the integration instead of building one outside of the integration and passing it in.

C#

```
builder.AddNpgsqlDbContext<MyDb1Context>(
    "postgresdb",
    configureDbContextOptions: (options) => options.UseNpgsql(npgsqlOptions
=>
        npgsqlOptions.ConfigureDataSource(dsBuilder =>
        {
            if
(string.IsNullOrEmpty(dsBuilder.ConnectionStringBuilder.Password))
            {
                var credentials = new DefaultAzureCredential();
                var tokenRequest = new
TokenRequestContext(["https://ossrdbms-aad.database.windows.net/.default"]);

                dsBuilder.UsePasswordProvider(
                    passwordProvider: _ =>
credentials.GetToken(tokenRequest).Token,
                    passwordProviderAsync: async (_, ct) => (await
credentials.GetTokenAsync(tokenRequest, ct)).Token);
            }
        }));
```

See also

- PostgreSQL docs
- Azure Database for PostgreSQL
- .NET Aspire PostgreSQL Entity Framework Core integration
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire Pomelo MySQL Entity Framework Core integration

Article • 02/07/2025

Includes: ✅ Hosting integration and ✅ Client integration

MySQL is an open-source Relational Database Management System (RDBMS) that uses Structured Query Language (SQL) to manage and manipulate data. It's employed in a many different environments, from small projects to large-scale enterprise systems and it's a popular choice to host data that underpins microservices in a cloud-native application. The .NET Aspire Pomelo MySQL Entity Framework Core integration enables you to connect to existing MySQL databases or create new instances from .NET with the mysql container image.

Hosting integration

The MySQL hosting integration models the server as the MySqlServerResource type and the database as the MySqlDatabaseResource type. To access these types and APIs, add the Aspire.Hosting.MySql NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.MySql
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add MySQL server resource and database resource

In your app host project, call AddMySql to add and return a MySQL resource builder. Chain a call to the returned resource builder to AddDatabase, to add a MySQL database resource.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var mysql = builder.AddMySql("mysql")
```

```
                    .WithLifetime(ContainerLifetime.Persistent);

var mysqldb = mysql.AddDatabase("mysqldb");

var myService = builder.AddProject<Projects.ExampleProject>()
                       .WithReference(mysqldb)
                       .WaitFor(mysqldb);

// After adding all resources, run the app...
```

> **Note**
>
> The SQL Server container is slow to start, so it's best to use a *persistent* lifetime to avoid unnecessary restarts. For more information, see Container resource lifetime.

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `mysql` image, it creates a new MySQL instance on your local machine. A reference to your MySQL resource builder (the `mysql` variable) is used to add a database. The database is named `mysqldb` and then added to the `ExampleProject`. The MySQL resource includes default credentials with a `username` of `root` and a random `password` generated using the CreateDefaultPasswordParameter method.

When the app host runs, the password is stored in the app host's secret store. It's added to the `Parameters` section, for example:

JSON

```
{
  "Parameters:mysql-password": "<THE_GENERATED_PASSWORD>"
}
```

The name of the parameter is `mysql-password`, but really it's just formatting the resource name with a `-password` suffix. For more information, see Safe storage of app secrets in development in ASP.NET Core and Add MySQL resource with parameters.

The WithReference method configures a connection in the `ExampleProject` named `mysqldb`.

> **Tip**
>
> If you'd rather connect to an existing MySQL server, call AddConnectionString instead. For more information, see Reference existing resources.

Add a MySQL resource with a data volume

To add a data volume to the SQL Server resource, call the WithDataVolume method on the SQL Server resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var mysql = builder.AddMySql("mysql")
                   .WithDataVolume();

var mysqldb = mysql.AddDatabase("mysqldb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(mysqldb)
       .WaitFor(mysqldb);

// After adding all resources, run the app...
```

The data volume is used to persist the MySQL server data outside the lifecycle of its container. The data volume is mounted at the `/var/lib/mysql` path in the SQL Server container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

> ⚠ **Warning**
>
> The password is stored in the data volume. When using a data volume and if the password changes, it will not work until you delete the volume.

Add a MySQL resource with a data bind mount

To add a data bind mount to the MySQL resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var mysql = builder.AddMySql("mysql")
                   .WithDataBindMount(source: @"C:\MySql\Data");

var db = sql.AddDatabase("mysqldb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(mysqldb)
```

```
        .WaitFor(mysqldb);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the MySQL data across container restarts. The data bind mount is mounted at the `C:\MySql\Data` on Windows (or `/MySql/Data` on Unix) path on the host machine in the MySQL container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add MySQL resource with parameters

When you want to provide a root MySQL password explicitly, you can pass it as a parameter. Consider the following alternative example:

C#

```
var password = builder.AddParameter("password", secret: true);

var mysql = builder.AddMySql("mysql", password)
                   .WithLifetime(ContainerLifetime.Persistent);

var mysqldb = mysql.AddDatabase("mysqldb");

var myService = builder.AddProject<Projects.ExampleProject>()
                       .WithReference(mysqldb)
                       .WaitFor(mysqldb);
```

For more information, see External parameters.

Add a PhpMyAdmin resource

phpMyAdmin is a popular web-based administration tool for MySQL. You can use it to browse and modify MySQL objects such as databases, tables, views, and indexes. To use phpMyAdmin within your .NET Aspire solution, call the WithPhpMyAdmin method.

This method adds a new container resource to the solution that hosts phpMyAdmin and connects it to the MySQL container:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var mysql = builder.AddMySql("mysql")
                   .WithPhpMyAdmin();

var db = sql.AddDatabase("mysqldb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(mysqldb)
       .WaitFor(mysqldb);

// After adding all resources, run the app...
```

When you run the solution, the .NET Aspire dashboard displays the phpMyAdmin resources with an endpoint. Select the link to the endpoint to view phpMyAdmin in a new browser tab.

Hosting integration health checks

The MySQL hosting integration automatically adds a health check for the MySQL resource. The health check verifies that the MySQL server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.MySql NuGet package.

Client integration

To get started with the .NET Aspire Pomelo MySQL Entity Framework integration, install the Aspire.Pomelo.EntityFrameworkCore.MySql NuGet package in the client-consuming project, that is, the project for the application that uses the MySQL Entity Framework Core client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Pomelo.EntityFrameworkCore.MySql
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add a MySQL database context

In the Program.cs file of your client-consuming project, call the AddMySqlDbContext extension method on any IHostApplicationBuilder to register a DbContext for use through the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddMySqlDbContext<ExampleDbContext>(connectionName: "mysqldb");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the SQL Server database resource in the app host project. In other words, when you call `AddDatabase` and provide a name of `mysqldb` that same name should be used when calling `AddMySqlDbContext`. For more information, see **Add MySQL server resource and database resource**.

To retrieve `ExampleDbContext` object from a service:

C#

```
public class ExampleService(ExampleDbContext context)
{
    // Use context...
}
```

For more information on dependency injection, see .NET dependency injection.

Enrich a MySQL database context

You may prefer to use the standard Entity Framework method to obtain a database context and add it to the dependency injection container:

C#

```
builder.Services.AddDbContext<ExampleDbContext>(options =>
    options.UseMySql(builder.Configuration.GetConnectionString("mysqldb")
        ?? throw new InvalidOperationException("Connection string 'mysqldb'
not found.")));
```

> **Note**
>
> The connection string name that you pass to the GetConnectionString method must match the name used when adding the MySQL resource in the app host project. For more information, see Add MySQL server resource and database resource.

You have more flexibility when you create the database context in this way, for example:

- You can reuse existing configuration code for the database context without rewriting it for .NET Aspire.
- You can use Entity Framework Core interceptors to modify database operations.
- You can choose not to use Entity Framework Core context pooling, which may perform better in some circumstances.

If you use this method, you can enhance the database context with .NET Aspire-style retries, health checks, logging, and telemetry features by calling the EnrichMySqlDbContext method:

C#

```
builder.EnrichMySqlDbContext<ExampleDbContext>(
    configureSettings: settings =>
    {
        settings.DisableRetry = false;
        settings.CommandTimeout = 30 // seconds
    });
```

The `settings` parameter is an instance of the PomeloEntityFrameworkCoreMySqlSettings class.

Configuration

The .NET Aspire Pomelo MySQL Entity Framework Core integration provides multiple options to configure the database connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddMySqlDatabaseDbContext<TContext>()`:

C#

```
builder.AddMySqlDatabaseDbContext<MyDbContext>("mysql");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "mysql": "Server=mysql;Database=mysqldb"
  }
}
```

The `EnrichMySqlDbContext` won't make use of the `ConnectionStrings` configuration section since it expects a `DbContext` to be registered at the point it's called.

For more information, see the MySqlConnector: ConnectionString documentation.

Use configuration providers

The .NET Aspire Pomelo MySQL Entity Framework Core integration supports Microsoft.Extensions.Configuration. It loads the PomeloEntityFrameworkCoreMySqlSettings from configuration files such as *appsettings.json* by using the `Aspire:Pomelo:EntityFrameworkCore:MySql` key.

The following example shows an *appsettings.json* that configures some of the available options:

JSON

```
{
  "Aspire": {
    "Pomelo": {
      "EntityFrameworkCore": {
        "MySql": {
          "ConnectionString": "YOUR_CONNECTIONSTRING",
          "DisableHealthChecks": true,
```

```
          "DisableTracing": true
        }
      }
    }
  }
}
```

For the complete MySQL integration JSON schema, see Aspire.Pomelo.EntityFrameworkCore.MySql/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<PomeloEntityFrameworkCoreMySqlSettings>` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddMySqlDbContext<MyDbContext>(
    "mysqldb",
    static settings => settings.DisableHealthChecks = true);
```

or

C#

```
builder.EnrichMySqlDbContext<MyDbContext>(
    static settings => settings.DisableHealthChecks = true);
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

The .NET Aspire Pomelo MySQL Entity Framework Core integration:

- Adds the health check when PomeloEntityFrameworkCoreMySqlSettings.DisableHealthChecks is `false`, which calls EF Core's CanConnectAsync method.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Pomelo MySQL Entity Framework Core integration uses the following log categories:

- `Microsoft.EntityFrameworkCore.ChangeTracking`
- `Microsoft.EntityFrameworkCore.Database.Command`
- `Microsoft.EntityFrameworkCore.Database.Connection`
- `Microsoft.EntityFrameworkCore.Database.Transaction`
- `Microsoft.EntityFrameworkCore.Infrastructure`
- `Microsoft.EntityFrameworkCore.Migrations`
- `Microsoft.EntityFrameworkCore.Model`
- `Microsoft.EntityFrameworkCore.Model.Validation`
- `Microsoft.EntityFrameworkCore.Query`
- `Microsoft.EntityFrameworkCore.Update`

Tracing

The .NET Aspire Pomelo MySQL Entity Framework Core integration will emit the following tracing activities using OpenTelemetry:

- `MySqlConnector`

Metrics

The .NET Aspire Pomelo MySQL Entity Framework Core integration currently supports the following metrics:

- MySqlConnector:
  - `db.client.connections.create_time`

- `db.client.connections.use_time`
- `db.client.connections.wait_time`
- `db.client.connections.idle.max`
- `db.client.connections.idle.min`
- `db.client.connections.max`
- `db.client.connections.pending_requests`
- `db.client.connections.timeouts`
- `db.client.connections.usage`

See also

- MySQL database
- Entity Framework Core docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire Oracle Entity Framework Core integration

Article • 01/26/2025

Includes: ✅ Hosting integration and ✅ Client integration

Oracle Database is a widely-used relational database management system owned and developed by Oracle. The .NET Aspire Oracle Entity Framework Core integration enables you to connect to existing Oracle servers or create new servers from .NET with the container-registry.orcale.com/databse/free container image.

Hosting integration

The .NET Aspire Oracle hosting integration models the server as the OracleDatabaseServerResource type and the database as the OracleDatabaseResource type. To access these types and APIs, add the Aspire.Hosting.Oracle NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Oracle
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Oracle server and database resources

In your app host project, call AddOracle to add and return an Oracle server resource builder. Chain a call to the returned resource builder to AddDatabase, to add an Oracle database to the server resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var oracle = builder.AddOracle("oracle")
                    .WithLifetime(ContainerLifetime.Persistent);
```

```
var oracledb = oracle.AddDatabase("oracledb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(oracledb);
       .WaitFor(oracledb);

// After adding all resources, run the app...
```

> **Note**
>
> The Oracle database container can be slow to start, so it's best to use a *persistent* lifetime to avoid unnecessary restarts. For more information, see Container resource lifetime.

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `container-registry.oracle.com/database/free` image, it creates a new Oracle server on your local machine. A reference to your Oracle resource builder (the `oracle` variable) is used to add a database. The database is named `oracledb` and then added to the `ExampleProject`. The Oracle resource includes a random `password` generated using the CreateDefaultPasswordParameter method.

The WithReference method configures a connection in the `ExampleProject` named `"oracledb"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing Oracle server, call AddConnectionString instead. For more information, see Reference existing resources.

Add Oracle resource with password parameter

The Oracle resource includes default credentials with a random password. Oracle supports configuration-based default passwords by using the environment variable `ORACLE_PWD`. When you want to provide a password explicitly, you can provide it as a parameter:

C#

```
var password = builder.AddParameter("password", secret: true);

var oracle = builder.AddOracle("oracle", password)
                    .WithLifetime(ContainerLifetime.Persistent);
```

```
var oracledb = oracle.AddDatabase("oracledb");

var myService = builder.AddProject<Projects.ExampleProject>()
                       .WithReference(oracledb)
                       .WaitFor(oracledb);
```

The preceding code gets a parameter to pass to the `AddOracle` API, and internally assigns the parameter to the `ORACLE_PWD` environment variable of the Oracle container. The `password` parameter is usually specified as a *user secret*:

JSON

```
{
  "Parameters": {
    "password": "Non-default-P@ssw0rd"
  }
}
```

For more information, see External parameters.

Add Oracle resource with data volume

To add a data volume to the Oracle resource, call the WithDataVolume method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var oracle = builder.AddOracle("oracle")
                    .WithDataVolume()
                    .WithLifetime(ContainerLifetime.Persistent);

var oracledb = oracle.AddDatabase("oracle");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(oracledb)
       .WaitFor(oracledb);

// After adding all resources, run the app...
```

The data volume is used to persist the Oracle data outside the lifecycle of its container. The data volume is mounted at the `/opt/oracle/oradata` path in the Oracle container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

> ⚠ **Warning**
>
> The password is stored in the data volume. When using a data volume and if the password changes, it will not work until you delete the volume.

Add Oracle resource with data bind mount

To add a data bind mount to the Oracle resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var oracle = builder.AddOracle("oracle")
                    .WithDataBindMount(source: @"C:\Oracle\Data");

var oracledb = oracle.AddDatabase("oracledb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(oracledb)
       .WaitFor(oracledb);

// After adding all resources, run the app...
```

> ⓘ **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Oracle data across container restarts. The data bind mount is mounted at the `C:\Oracle\Data` on Windows (or `/Oracle/Data` on Unix) path on the host machine in the Oracle container. For more information on data bind mounts, see Docker docs: Bind mounts.

Hosting integration health checks

The Oracle hosting integration automatically adds a health check for the Oracle resource. The health check verifies that the Oracle server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Oracle NuGet package.

Client integration

You need an Oracle database and connection string for accessing the database. To get started with the .NET Aspire Oracle client integration, install the Aspire.Oracle.EntityFrameworkCore NuGet package in the client-consuming project, that is, the project for the application that uses the Oracle client. The Oracle client integration registers a DbContext instance that you can use to interact with Oracle.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Oracle.EntityFrameworkCore
```

Add Oracle client

In the *Program.cs* file of your client-consuming project, call the AddOracleDatabaseDbContext extension method on any IHostApplicationBuilder to register a `DbContext` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddOracleDatabaseDbContext<ExampleDbContext>(connectionName:
"oracledb");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Oracle database resource in the app host project. In other words, when you call `AddDatabase` and provide a name of `oracledb` that same name should be used when calling `AddOracleDatabaseDbContext`. For more information, see Add Oracle server and database resources.

You can then retrieve the DbContext instance using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(ExampleDbContext context)
{
    // Use database context...
}
```

For more information on dependency injection, see .NET dependency injection.

Enrich Oracle database context

You may prefer to use the standard Entity Framework method to obtain a database context and add it to the dependency injection container:

C#

```
builder.Services.AddDbContext<ExampleDbContext>(options =>
    options.UseOracle(builder.Configuration.GetConnectionString("oracledb")
        ?? throw new InvalidOperationException("Connection string 'oracledb'
not found.")));
```

> **Note**
>
> The connection string name that you pass to the GetConnectionString method must match the name used when adding the Oracle resource in the app host project. For more information, see Add Oracle server and database resources.

You have more flexibility when you create the database context in this way, for example:

- You can reuse existing configuration code for the database context without rewriting it for .NET Aspire.
- You can use Entity Framework Core interceptors to modify database operations.
- You can choose not to use Entity Framework Core context pooling, which may perform better in some circumstances.

If you use this method, you can enhance the database context with .NET Aspire-style retries, health checks, logging, and telemetry features by calling the EnrichOracleDatabaseDbContext method:

C#

```
builder.EnrichOracleDatabaseDbContext<ExampleDbContext>(
    configureSettings: settings =>
    {
```

```
        settings.DisableRetry = false;
        settings.CommandTimeout = 30 // seconds
    });
```

The `settings` parameter is an instance of the OracleEntityFrameworkCoreSettings class.

Configuration

The .NET Aspire Oracle Entity Framework Core integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you provide the name of the connection string when calling `builder.AddOracleDatabaseDbContext<TContext>()`:

C#

```
builder.AddOracleDatabaseDbContext<ExampleDbContext>("oracleConnection");
```

The connection string is retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "oracleConnection": "Data Source=TORCL;User Id=OracleUser;Password=Non-default-P@ssw0rd;"
  }
}
```

The `EnrichOracleDatabaseDbContext` won't make use of the `ConnectionStrings` configuration section since it expects a `DbContext` to be registered at the point it is called.

For more information, see the ODP.NET documentation.

Use configuration providers

The .NET Aspire Oracle Entity Framework Core integration supports Microsoft.Extensions.Configuration from configuration files such as *appsettings.json* by

using the `Aspire:Oracle:EntityFrameworkCore` key. If you have set up your configurations in the `Aspire:Oracle:EntityFrameworkCore` section you can just call the method without passing any parameter.

The following is an example of an *appsettings.json* that configures some of the available options:

JSON

```
{
  "Aspire": {
    "Oracle": {
      "EntityFrameworkCore": {
        "DisableHealthChecks": true,
        "DisableTracing": true,
        "DisableRetry": false,
        "CommandTimeout": 30
      }
    }
  }
}
```

> **Tip**
>
> The `CommandTimeout` property is in seconds. When set as shown in the preceding example, the timeout is 30 seconds.

Use inline delegates

You can also pass the `Action<OracleEntityFrameworkCoreSettings>` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddOracleDatabaseDbContext<ExampleDbContext>(
    "oracle",
    static settings => settings.DisableHealthChecks  = true);
```

or

C#

```
builder.EnrichOracleDatabaseDbContext<ExampleDbContext>(
    static settings => settings.DisableHealthChecks  = true);
```

Configuration options

Here are the configurable options with corresponding default values:

Expand table

Name	Description
`ConnectionString`	The connection string of the Oracle database to connect to.
`DisableHealthChecks`	A boolean value that indicates whether the database health check is disabled or not.
`DisableTracing`	A boolean value that indicates whether the OpenTelemetry tracing is disabled or not.
`DisableRetry`	A boolean value that indicates whether command retries should be disabled or not.
`CommandTimeout`	The time in seconds to wait for the command to execute.

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

By default, the .NET Aspire Oracle Entity Framework Core integration handles the following:

- Checks if the OracleEntityFrameworkCoreSettings.DisableHealthChecks is `true`.
- If so, adds the DbContextHealthCheck, which calls EF Core's CanConnectAsync method. The name of the health check is the name of the `TContext` type.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not

metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Oracle Entity Framework Core integration uses the following log categories:

- `Microsoft.EntityFrameworkCore.ChangeTracking`
- `Microsoft.EntityFrameworkCore.Database.Command`
- `Microsoft.EntityFrameworkCore.Database.Connection`
- `Microsoft.EntityFrameworkCore.Database.Transaction`
- `Microsoft.EntityFrameworkCore.Infrastructure`
- `Microsoft.EntityFrameworkCore.Migrations`
- `Microsoft.EntityFrameworkCore.Model`
- `Microsoft.EntityFrameworkCore.Model.Validation`
- `Microsoft.EntityFrameworkCore.Query`
- `Microsoft.EntityFrameworkCore.Update`

Tracing

The .NET Aspire Oracle Entity Framework Core integration will emit the following tracing activities using OpenTelemetry:

- OpenTelemetry.Instrumentation.EntityFrameworkCore

Metrics

The .NET Aspire Oracle Entity Framework Core integration currently supports the following metrics:

- Microsoft.EntityFrameworkCore

See also

- Oracle Database
- Oracle Database Documentation
- Entity Framework Core docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire PostgreSQL Entity Framework Core integration

Article • 02/07/2025

Includes: ✅ Hosting integration and ✅ Client integration

PostgreSQL is a powerful, open source object-relational database system with many years of active development that has earned it a strong reputation for reliability, feature robustness, and performance. The .NET Aspire PostgreSQL Entity Framework Core integration provides a way to connect to existing PostgreSQL databases, or create new instances from .NET with the docker.io/library/postgres container image.

Hosting integration

The PostgreSQL hosting integration models various PostgreSQL resources as the following types.

- PostgresServerResource
- PostgresDatabaseResource
- PgAdminContainerResource
- PgWebContainerResource

To access these types and APIs for expressing them as resources in your app host project, install the Aspire.Hosting.PostgreSQL NuGet package:

.NET CLI

```
dotnet add package Aspire.Hosting.PostgreSQL
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add PostgreSQL server resource

In your app host project, call AddPostgres on the `builder` instance to add a PostgreSQL server resource then call AddDatabase on the `postgres` instance to add a database resource as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres");
var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/library/postgres` image, it creates a new PostgreSQL server instance on your local machine. A reference to your PostgreSQL server and your PostgreSQL database instance (the `postgresdb` variable) are used to add a dependency to the `ExampleProject`. The PostgreSQL server resource includes default credentials with a `username` of `"postgres"` and randomly generated `password` using the CreateDefaultPasswordParameter method.

The WithReference method configures a connection in the `ExampleProject` named `"messaging"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing PostgreSQL server, call AddConnectionString instead. For more information, see Reference existing resources.

Add PostgreSQL pgAdmin resource

When adding PostgreSQL resources to the `builder` with the `AddPostgres` method, you can chain calls to WithPgAdmin to add the **dpage/pgadmin4** container. This container is a cross-platform client for PostgreSQL databases, that serves a web-based admin dashboard. Consider the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres")
                      .WithPgAdmin();

var postgresdb = postgres.AddDatabase("postgresdb");
```

```
var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

The preceding code adds a container based on the `docker.io/dpage/pgadmin4` image. The container is used to manage the PostgreSQL server and database resources. The `WithPgAdmin` method adds a container that serves a web-based admin dashboard for PostgreSQL databases.

Configure the pgAdmin host port

To configure the host port for the pgAdmin container, call the WithHostPort method on the PostgreSQL server resource. The following example shows how to configure the host port for the pgAdmin container:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres")
                      .WithPgAdmin(pgAdmin => pgAdmin.WithHostPort(5050));

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

The preceding code adds and configures the host port for the pgAdmin container. The host port is otherwise randomly assigned.

Add PostgreSQL pgWeb resource

When adding PostgreSQL resources to the `builder` with the `AddPostgres` method, you can chain calls to WithPgWeb to add the **sosedoff/pgweb** container. This container is a cross-platform client for PostgreSQL databases, that serves a web-based admin dashboard. Consider the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres")
```

```
                        .WithPgWeb();

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

The preceding code adds a container based on the `docker.io/sosedoff/pgweb` image. All registered PostgresDatabaseResource instances are used to create a configuration file per instance, and each config is bound to the **pgweb** container bookmark directory. For more information, see PgWeb docs: Server connection bookmarks.

Configure the pgWeb host port

To configure the host port for the pgWeb container, call the WithHostPort method on the PostgreSQL server resource. The following example shows how to configure the host port for the pgAdmin container:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres")
                      .WithPgWeb(pgWeb => pgWeb.WithHostPort(5050));

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

The preceding code adds and configures the host port for the pgWeb container. The host port is otherwise randomly assigned.

Add PostgreSQL server resource with data volume

To add a data volume to the PostgreSQL server resource, call the WithDataVolume method on the PostgreSQL server resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);
```

```
var postgres = builder.AddPostgres("postgres")
                      .WithDataVolume(isReadOnly: false);

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

The data volume is used to persist the PostgreSQL server data outside the lifecycle of its container. The data volume is mounted at the `/var/lib/postgresql/data` path in the PostgreSQL server container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add PostgreSQL server resource with data bind mount

To add a data bind mount to the PostgreSQL server resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres")
                      .WithDataBindMount(
                          source: @"C:\PostgreSQL\Data",
                          isReadOnly: false);

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the PostgreSQL server data across container restarts. The data bind mount is mounted at the `C:\PostgreSQL\Data` on Windows (or `/PostgreSQL/Data` on Unix) path on the host machine in the PostgreSQL server container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add PostgreSQL server resource with init bind mount

To add an init bind mount to the PostgreSQL server resource, call the WithInitBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres")
                      .WithInitBindMount(@"C:\PostgreSQL\Init");

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

The init bind mount relies on the host machine's filesystem to initialize the PostgreSQL server database with the containers *init* folder. This folder is used for initialization, running any executable shell scripts or *.sql* command files after the *postgres-data* folder is created. The init bind mount is mounted at the `C:\PostgreSQL\Init` on Windows (or `/PostgreSQL/Init` on Unix) path on the host machine in the PostgreSQL server container.

Add PostgreSQL server resource with parameters

When you want to explicitly provide the username and password used by the container image, you can provide these credentials as parameters. Consider the following alternative example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var username = builder.AddParameter("username", secret: true);
var password = builder.AddParameter("password", secret: true);
```

```
var postgres = builder.AddPostgres("postgres", username, password);
var postgresdb = postgres.AddDatabase("postgresdb
```

For more information on providing parameters, see External parameters.

The PostgreSQL hosting integration automatically adds a health check for the PostgreSQL server resource. The health check verifies that the PostgreSQL server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Npgsql NuGet package.

To get started with the .NET Aspire PostgreSQL Entity Framework Core client integration, install the Aspire.Npgsql.EntityFrameworkCore.PostgreSQL NuGet package in the client-consuming project, that is, the project for the application that uses the PostgreSQL client. The .NET Aspire PostgreSQL Entity Framework Core client integration registers your desired `DbContext` subclass instances that you can use to interact with PostgreSQL.

.NET CLI

In the *Program.cs* file of your client-consuming project, call the AddNpgsqlDbContext extension method on any IHostApplicationBuilder to register your DbContext subclass for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddNpgsqlDbContext<YourDbContext>(connectionName: "postgresdb");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the PostgreSQL server resource in the app host project. For more information, see **Add PostgreSQL server resource**.

After adding `YourDbContext` to the builder, you can get the `YourDbContext` instance using dependency injection. For example, to retrieve your data source object from an example service define it as a constructor parameter and ensure the `ExampleService` class is registered with the dependency injection container:

C#

```
public class ExampleService(YourDbContext context)
{
    // Use context...
}
```

For more information on dependency injection, see .NET dependency injection.

Enrich an Npgsql database context

You may prefer to use the standard Entity Framework method to obtain a database context and add it to the dependency injection container:

C#

```
builder.Services.AddDbContext<YourDbContext>(options =>

options.UseNpgsql(builder.Configuration.GetConnectionString("postgresdb")
        ?? throw new InvalidOperationException("Connection string
'postgresdb' not found.")));
```

> **Note**

The connection string name that you pass to the GetConnectionString method must match the name used when adding the PostgreSQL server resource in the app host project. For more information, see Add PostgreSQL server resource.

You have more flexibility when you create the database context in this way, for example:

- You can reuse existing configuration code for the database context without rewriting it for .NET Aspire.
- You can use Entity Framework Core interceptors to modify database operations.
- You can choose not to use Entity Framework Core context pooling, which may perform better in some circumstances.

If you use this method, you can enhance the database context with .NET Aspire-style retries, health checks, logging, and telemetry features by calling the EnrichNpgsqlDbContext method:

```
C#

builder.EnrichNpgsqlDbContext<YourDbContext>(
    configureSettings: settings =>
    {
        settings.DisableRetry = false;
        settings.CommandTimeout = 30;
    });
```

The `settings` parameter is an instance of the NpgsqlEntityFrameworkCorePostgreSQLSettings class.

Configuration

The .NET Aspire PostgreSQL Entity Framework Core integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you provide the name of the connection string when calling the AddNpgsqlDbContext method:

```
C#
```

```
builder.AddNpgsqlDbContext<MyDbContext>("pgdb");
```

The connection string is retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "pgdb": "Host=myserver;Database=test"
  }
}
```

The `EnrichNpgsqlDbContext` won't make use of the `ConnectionStrings` configuration section since it expects a `DbContext` to be registered at the point it's called.

For more information, see the ConnectionString.

Use configuration providers

The .NET Aspire PostgreSQL Entity Framework Core integration supports Microsoft.Extensions.Configuration. It loads the NpgsqlEntityFrameworkCorePostgreSQLSettings from configuration files such as *appsettings.json* by using the `Aspire:Npgsql:EntityFrameworkCore:PostgreSQL` key. If you have set up your configurations in the `Aspire:Npgsql:EntityFrameworkCore:PostgreSQL` section you can just call the method without passing any parameter.

The following example shows an *appsettings.json* file that configures some of the available options:

JSON

```
{
  "Aspire": {
    "Npgsql": {
      "EntityFrameworkCore": {
        "PostgreSQL": {
          "ConnectionString": "Host=myserver;Database=postgresdb",
          "DisableHealthChecks": true,
          "DisableTracing": true
        }
      }
    }
  }
}
```

For the complete PostgreSQL Entity Framework Core client integration JSON schema, see Aspire.Npgsql.EntityFrameworkCore.PostgreSQL/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<NpgsqlEntityFrameworkCorePostgreSQLSettings>` delegate to set up some or all the options inline, for example to set the `ConnectionString`:

C#

```
builder.AddNpgsqlDbContext<YourDbContext>(
    "pgdb",
    static settings => settings.ConnectionString = "<YOUR CONNECTION
STRING>");
```

Configure multiple DbContext classes

If you want to register more than one DbContext with different configuration, you can use `$"Aspire:Npgsql:EntityFrameworkCore:PostgreSQL:{typeof(TContext).Name}"` configuration section name. The json configuration would look like:

JSON

```
{
  "Aspire": {
    "Npgsql": {
      "EntityFrameworkCore": {
        "PostgreSQL": {
          "ConnectionString": "<YOUR CONNECTION STRING>",
          "DisableHealthChecks": true,
          "DisableTracing": true,
          "AnotherDbContext": {
            "ConnectionString": "<ANOTHER CONNECTION STRING>",
            "DisableTracing": false
          }
        }
      }
    }
  }
}
```

Then calling the AddNpgsqlDbContext method with `AnotherDbContext` type parameter would load the settings from `Aspire:Npgsql:EntityFrameworkCore:PostgreSQL:AnotherDbContext` section.

C#

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

By default, the .NET Aspire PostgreSQL Entity Framework Core integrations handles the following:

- Adds the DbContextHealthCheck , which calls EF Core's CanConnectAsync method. The name of the health check is the name of the `TContext` type.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

.NE Aspire integrations automatically set up Logging, Tracing, and Metrics cor gurations, which are sometimes known as *the pillars of observability*. For more inf nation about integration observability and telemetry, see .NET Aspire integrations ov iew. Depending on the backing service, some integrations may only support some f se features. For example, some integrations support logging and tracing, but not cs. Telemetry features can also be disabled using the techniques presented in the guration section.

NET Aspire PostgreSQL ntity Framework Core integration uses the following Log ories:

- `Microsoft.EntityFrameworkCore. Tracking`
- `Microsoft.EntityFrameworkCore.Data mmand`
- `Microsoft.EntityFrameworkCore.Databas ion`

- `Microsoft.EntityFrameworkCore.Migrations`
- `Microsoft.EntityFrameworkCore.Model`
- `Microsoft.EntityFrameworkCore.Model.Validation`
- `Microsoft.EntityFrameworkCore.Query`
- `Microsoft.EntityFrameworkCore.Update`

Tracing

The .NET Aspire PostgreSQL Entity Framework Core integration will emit the following tracing activities using OpenTelemetry:

- `Npgsql`

Metrics

The .NET Aspire PostgreSQL Entity Framework Core integration will emit the following metrics using OpenTelemetry:

- Microsoft.EntityFrameworkCore:
  - `ec_Microsoft_EntityFrameworkCore_active_db_contexts`
  - `ec_Microsoft_EntityFrameworkCore_total_queries`
  - `ec_Microsoft_EntityFrameworkCore_queries_per_second`
  - `ec_Microsoft_EntityFrameworkCore_total_save_changes`
  - `ec_Microsoft_EntityFrameworkCore_save_changes_per_second`
  - `ec_Microsoft_EntityFrameworkCore_compiled_query_cache_hit_rate`
  - `ec_Microsoft_Entity_total_execution_strategy_operation_failures`
  - `ec_Microsoft_E_execution_strategy_operation_failures_per_second`
  - `ec_Microsoft_EntityFramew_total_optimistic_concurrency_failures`
  - `ec_Microsoft_EntityF_optimistic_concurrency_failures_per_second`
- Npgsql:
  - `ec_Npgsql_bytes_written_per_second`
  - `ec_Npgsql_bytes_read_per_second`
  - `ec_Npgsql_commands_per_second`
  - `ec_Npgsql_total_commands`
  - `ec_Npgsql_current_commands`
  - `ec_Npgsql_failed_commands`
  - `ec_Npgsql_prepared_commands_ratio`
  - `ec_Npgsql_connection_pools`
  - `ec_Npgsql_multiplexing_average_commands_per_batch`

- `ec_Npgsql_multiplexing_average_write_time_per_batch`

See also

- PostgreSQL docs
- .NET Aspire Azure PostgreSQL Entity Framework Core integration
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire SQL Server Entity Framework Core integration

Article • 02/12/2025

Includes: ✅ Hosting integration and ✅ Client integration

SQL Server is a relational database management system developed by Microsoft. The .NET Aspire SQL Server Entity Framework Core integration enables you to connect to existing SQL Server instances or create new instances from .NET with the mcr.microsoft.com/mssql/server container image.

Hosting integration

The SQL Server hosting integration models the server as the SqlServerServerResource type and the database as the SqlServerDatabaseResource type. To access these types and APIs, add the Aspire.Hosting.SqlServer NuGet package in the app host project.

.NET CLI

```
.NET CLI

dotnet add package Aspire.Hosting.SqlServer
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add SQL Server resource and database resource

In your app host project, call AddSqlServer to add and return a SQL Server resource builder. Chain a call to the returned resource builder to AddDatabase, to add SQL Server database resource.

```
C#

var builder = DistributedApplication.CreateBuilder(args);

var sql = builder.AddSqlServer("sql")
                 .WithLifetime(ContainerLifetime.Persistent);
```

```
var db = sql.AddDatabase("database");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(db)
       .WaitFor(db);

// After adding all resources, run the app...
```

> **Note**
>
> The SQL Server container is slow to start, so it's best to use a *persistent* lifetime to avoid unnecessary restarts. For more information, see Container resource lifetime.

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `mcr.microsoft.com/mssql/server` image, it creates a new SQL Server instance on your local machine. A reference to your SQL Server resource builder (the `sql` variable) is used to add a database. The database is named `database` and then added to the `ExampleProject`. The SQL Server resource includes default credentials with a `username` of `sa` and a random `password` generated using the CreateDefaultPasswordParameter method.

When the app host runs, the password is stored in the app host's secret store. It's added to the `Parameters` section, for example:

JSON

```
{
  "Parameters:sql-password": "<THE_GENERATED_PASSWORD>"
}
```

The name of the parameter is `sql-password`, but really it's just formatting the resource name with a `-password` suffix. For more information, see Safe storage of app secrets in development in ASP.NET Core and Add SQL Server resource with parameters.

The WithReference method configures a connection in the `ExampleProject` named `database`.

> **Tip**
>
> If you'd rather connect to an existing SQL Server, call AddConnectionString instead. For more information, see Reference existing resources.

Add SQL Server resource with data volume

To add a data volume to the SQL Server resource, call the WithDataVolume method on the SQL Server resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var sql = builder.AddSqlServer("sql")
                 .WithDataVolume();

var db = sql.AddDatabase("database");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(db)
       .WaitFor(db);

// After adding all resources, run the app...
```

The data volume is used to persist the SQL Server data outside the lifecycle of its container. The data volume is mounted at the `/var/opt/mssql` path in the SQL Server container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

> ⚠ **Warning**
>
> The password is stored in the data volume. When using a data volume and if the password changes, it will not work until you delete the volume.

Add SQL Server resource with data bind mount

To add a data bind mount to the SQL Server resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var sql = builder.AddSqlServer("sql")
                 .WithDataBindMount(source: @"C:\SqlServer\Data");

var db = sql.AddDatabase("database");

builder.AddProject<Projects.ExampleProject>()
```

```
        .WithReference(db)
        .WaitFor(db);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the SQL Server data across container restarts. The data bind mount is mounted at the `C:\SqlServer\Data` on Windows (or `/SqlServer/Data` on Unix) path on the host machine in the SQL Server container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add SQL Server resource with parameters

When you want to explicitly provide the password used by the container image, you can provide these credentials as parameters. Consider the following alternative example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var password = builder.AddParameter("password", secret: true);

var sql = builder.AddSqlServer("sql", password);
var db = sql.AddDatabase("database");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(db)
       .WaitFor(db);

// After adding all resources, run the app...
```

For more information on providing parameters, see External parameters.

Connect to database resources

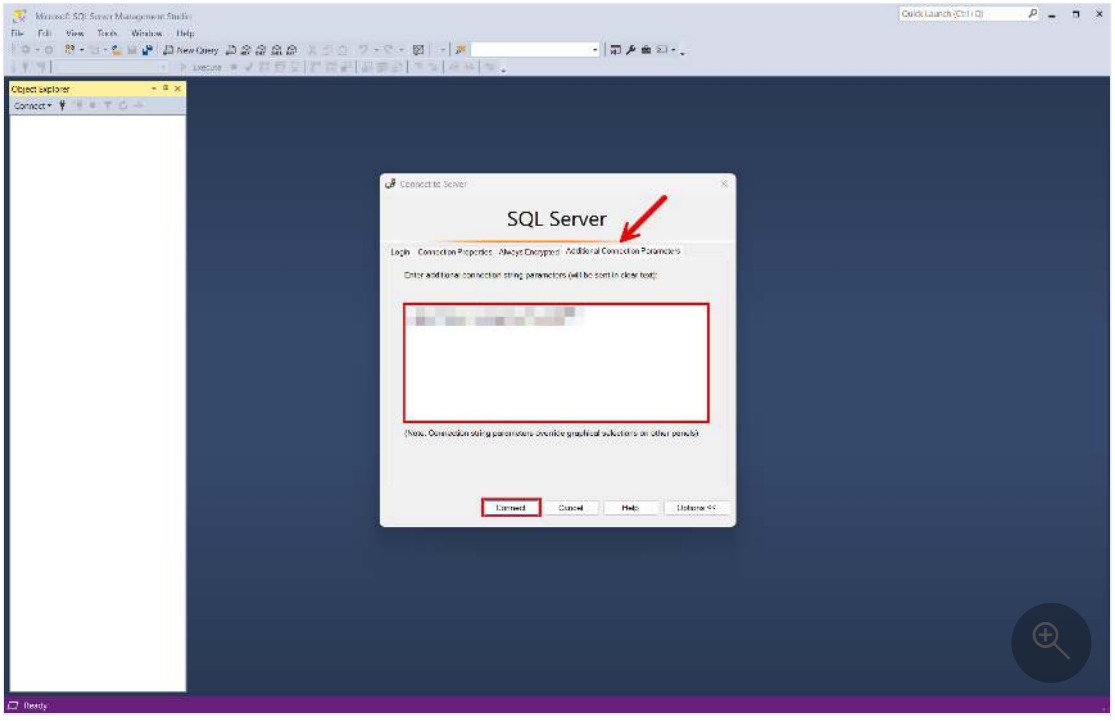
When the .NET Aspire app host runs, the server's database resources can be accessed from external tools, such as SQL Server Management Studio (SSMS) or MSSQL for Visual

Studio Code. The connection string for the database resource is available in the dependent resources environment variables and is accessed using the .NET Aspire dashboard: Resource details pane. The environment variable is named `ConnectionStrings__{name}` where `{name}` is the name of the database resource, in this example it's `database`. Use the connection string to connect to the database resource from external tools. Imagine that you have a database named `todos` with a single `dbo.Todos` table.

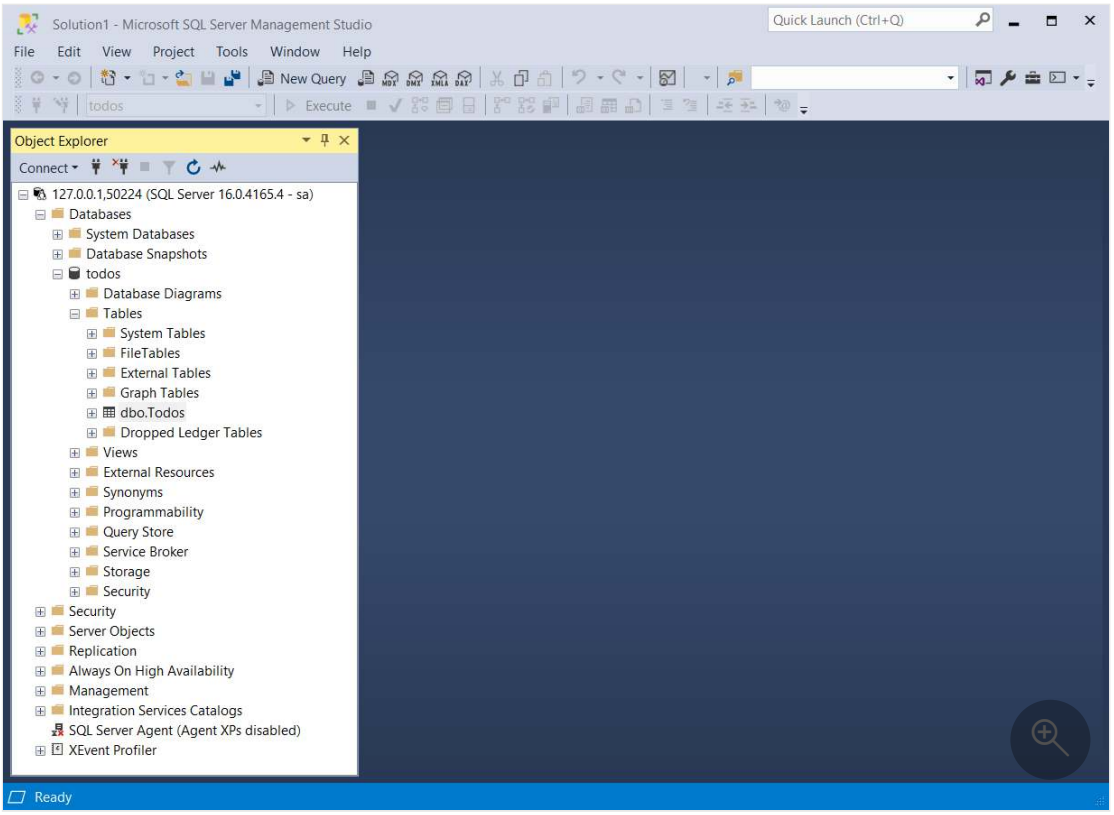
SQL Server Management Studio

To connect to the database resource from SQL Server Management Studio, follow these steps:

1. Open SSMS.
2. In the **Connect to Server** dialog, select the **Additional Connection Parameters** tab.
3. Paste the connection string into the **Additional Connection Parameters** field and select **Connect**.



4. If you're connected, you can see the database resource in the **Object Explorer**:



For more information, see SQL Server Management Studio: Connect to a server.

Hosting integration health checks

The SQL Server hosting integration automatically adds a health check for the SQL Server resource. The health check verifies that the SQL Server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.SqlServer NuGet package.

Client integration

To get started with the .NET Aspire SQL Server Entity Framework Core integration, install the Aspire.Microsoft.EntityFrameworkCore.SqlServer NuGet package in the client-consuming project, that is, the project for the application that uses the SQL Server Entity Framework Core client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Microsoft.EntityFrameworkCore.SqlServer
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add SQL Server database context

In the Program.cs file of your client-consuming project, call the AddSqlServerDbContext extension method on any IHostApplicationBuilder to register a DbContext for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddSqlServerDbContext<ExampleDbContext>(connectionName: "database");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the SQL Server database resource in the app host project. In other words, when you call `AddDatabase` and provide a name of `database` that same name should be used when calling `AddSqlServerDbContext`. For more information, see **Add SQL Server resource and database resource**.

To retrieve `ExampleDbContext` object from a service:

C#

```
public class ExampleService(ExampleDbContext context)
{
    // Use context...
}
```

For more information on dependency injection, see .NET dependency injection.

Enrich a SQL Server database context

You may prefer to use the standard Entity Framework method to obtain a database context and add it to the dependency injection container:

C#

```
builder.Services.AddDbContext<ExampleDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("database")
        ?? throw new InvalidOperationException("Connection string 'database'
not found.")));
```

> **Note**
>
> The connection string name that you pass to the GetConnectionString method must match the name used when adding the SQL server resource in the app host project. For more information, see Add SQL Server resource and database resource.

You have more flexibility when you create the database context in this way, for example:

- You can reuse existing configuration code for the database context without rewriting it for .NET Aspire.
- You can use Entity Framework Core interceptors to modify database operations.
- You can choose not to use Entity Framework Core context pooling, which may perform better in some circumstances.

If you use this method, you can enhance the database context with .NET Aspire-style retries, health checks, logging, and telemetry features by calling the EnrichSqlServerDbContext method:

C#

```
builder.EnrichSqlServerDbContext<ExampleDbContext>(
    configureSettings: settings =>
    {
        settings.DisableRetry = false;
        settings.CommandTimeout = 30; // seconds
    });
```

The `settings` parameter is an instance of the MicrosoftEntityFrameworkCoreSqlServerSettings class.

Configuration

The .NET Aspire SQL Server Entity Framework Core integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use connection string

When using a connection string from the `ConnectionStrings` configuration section, you provide the name of the connection string when calling `builder.AddSqlServerDbContext<TContext>()`:

C#

```
builder.AddSqlServerDbContext<ExampleDbContext>("sql");
```

The connection string is retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "sql": "Data Source=myserver;Initial Catalog=master"
  }
}
```

The `EnrichSqlServerDbContext` won't make use of the `ConnectionStrings` configuration section since it expects a `DbContext` to be registered at the point it's called.

For more information, see the ConnectionString.

Use configuration providers

The .NET Aspire SQL Server Entity Framework Core integration supports Microsoft.Extensions.Configuration. It loads the MicrosoftEntityFrameworkCoreSqlServerSettings from configuration files such as *appsettings.json* by using the `Aspire:Microsoft:EntityFrameworkCore:SqlServer` key. If you have set up your configurations in the `Aspire:Microsoft:EntityFrameworkCore:SqlServer` section you can just call the method without passing any parameter.

The following is an example of an *appsettings.json* file that configures some of the available options:

JSON

```
{
  "Aspire": {
    "Microsoft": {
      "EntityFrameworkCore": {
        "SqlServer": {
```

```
          "ConnectionString": "YOUR_CONNECTIONSTRING",
          "DbContextPooling": true,
          "DisableHealthChecks": true,
          "DisableTracing": true,
          "DisableMetrics": false
        }
      }
    }
  }
}
```

Use inline configurations

You can also pass the `Action<MicrosoftEntityFrameworkCoreSqlServerSettings>` delegate to set up some or all the options inline, for example to turn off the metrics:

C#

```
builder.AddSqlServerDbContext<YourDbContext>(
    "sql",
    static settings =>
        settings.DisableMetrics = true);
```

Configure multiple DbContext connections

If you want to register more than one `DbContext` with different configuration, you can use `$"Aspire.Microsoft.EntityFrameworkCore.SqlServer:{typeof(TContext).Name}"` configuration section name. The json configuration would look like:

JSON

```
{
  "Aspire": {
    "Microsoft": {
      "EntityFrameworkCore": {
          "SqlServer": {
            "ConnectionString": "YOUR_CONNECTIONSTRING",
            "DbContextPooling": true,
            "DisableHealthChecks": true,
            "DisableTracing": true,
            "DisableMetrics": false,
          "AnotherDbContext": {
            "ConnectionString": "AnotherDbContext_CONNECTIONSTRING",
            "DisableTracing": false
          }
        }
      }
    }
```

```
    }
}
```

Then calling the `AddSqlServerDbContext` method with `AnotherDbContext` type parameter would load the settings from `Aspire:Microsoft:EntityFrameworkCore:SqlServer:AnotherDbContext` section.

C#

```
builder.AddSqlServerDbContext<AnotherDbContext>("another-sql");
```

Configuration options

Here are the configurable options with corresponding default values:

Expand table

Name	Description
`ConnectionString`	The connection string of the SQL Server database to connect to.
`DbContextPooling`	A boolean value that indicates whether the db context will be pooled or explicitly created every time it's requested
`MaxRetryCount`	The maximum number of retry attempts. Default value is 6, set it to 0 to disable the retry mechanism.
`DisableHealthChecks`	A boolean value that indicates whether the database health check is disabled or not.
`DisableTracing`	A boolean value that indicates whether the OpenTelemetry tracing is disabled or not.
`DisableMetrics`	A boolean value that indicates whether the OpenTelemetry metrics are disabled or not.
`Timeout`	The time in seconds to wait for the command to execute.

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#

- Health checks in ASP.NET Core

By default, the .NET Aspire Sql Server Entity Framework Core integration handles the following:

- Adds the DbContextHealthCheck, which calls EF Core's CanConnectAsync method. The name of the health check is the name of the `TContext` type.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire SQL Server Entity Framework Core integration uses the following Log categories:

- `Microsoft.EntityFrameworkCore.ChangeTracking`
- `Microsoft.EntityFrameworkCore.Database.Command`
- `Microsoft.EntityFrameworkCore.Database.Connection`
- `Microsoft.EntityFrameworkCore.Database.Transaction`
- `Microsoft.EntityFrameworkCore.Infrastructure`
- `Microsoft.EntityFrameworkCore.Migrations`
- `Microsoft.EntityFrameworkCore.Model`
- `Microsoft.EntityFrameworkCore.Model.Validation`
- `Microsoft.EntityFrameworkCore.Query`
- `Microsoft.EntityFrameworkCore.Update`

Tracing

The .NET Aspire SQL Server Entity Framework Core integration will emit the following Tracing activities using OpenTelemetry:

- "OpenTelemetry.Instrumentation.EntityFrameworkCore"

Metrics

The .NET Aspire SQL Server Entity Framework Core integration will emit the following metrics using OpenTelemetry:

- Microsoft.EntityFrameworkCore:
  - `ec_Microsoft_EntityFrameworkCore_active_db_contexts`
  - `ec_Microsoft_EntityFrameworkCore_total_queries`
  - `ec_Microsoft_EntityFrameworkCore_queries_per_second`
  - `ec_Microsoft_EntityFrameworkCore_total_save_changes`
  - `ec_Microsoft_EntityFrameworkCore_save_changes_per_second`
  - `ec_Microsoft_EntityFrameworkCore_compiled_query_cache_hit_rate`
  - `ec_Microsoft_Entity_total_execution_strategy_operation_failures`
  - `ec_Microsoft_E_execution_strategy_operation_failures_per_second`
  - `ec_Microsoft_EntityFramew_total_optimistic_concurrency_failures`
  - `ec_Microsoft_EntityF_optimistic_concurrency_failures_per_second`

See also

- Azure SQL Database documentation
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire Keycloak integration (Preview)

Article • 03/06/2025

Includes: ✅ Hosting integration and ✅ Client integration

Keycloak is an open-source Identity and Access Management solution aimed at modern applications and services. The .NET Aspire Keycloak integration enables you to connect to existing Keycloak instances or create new instances from .NET with the quay.io/keycloak/keycloak container image.

Hosting integration

The .NET Aspire Keycloak hosting integration models the server as the KeycloakResource type. To access these types and APIs, add the Aspire.Hosting.Keycloak NuGet package in the app host project.

.NET CLI

```
dotnet add package Aspire.Hosting.Keycloak --prerelease
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Keycloak resource

In your app host project, call AddKeycloak to add and return a Keycloak resource builder. Chain a call to the returned resource builder to configure the Keycloak.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var keycloak = builder.AddKeycloak("keycloak", 8080);

var apiService = builder.AddProject<Projects.Keycloak_ApiService>
("apiservice")
                        .WithReference(keycloak)
                        .WaitFor(keycloak);
```

```
builder.AddProject<Projects.Keycloak_Web>("webfrontend")
       .WithExternalHttpEndpoints()
       .WithReference(keycloak)
       .WithReference(apiService)
       .WaitFor(apiService);

// After adding all resources, run the app...
```

> **Tip**
>
> For local development use a stable port for the Keycloak resource (`8080` in the preceding example). It can be any port, but it should be stable to avoid issues with browser cookies that will persist OIDC tokens (which include the authority URL, with port) beyond the lifetime of the *app host*.

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `quay.io/keycloak/keycloak` image, it creates a new Keycloak instance on your local machine. The Keycloak resource includes default credentials:

- `KEYCLOAK_ADMIN`: A value of `admin`.
- `KEYCLOAK_ADMIN_PASSWORD`: Random `password` generated using the CreateDefaultPasswordParameter method.

When the app host runs, the password is stored in the app host's secret store. It's added to the `Parameters` section, for example:

JSON

```
{
  "Parameters:keycloak-password": "<THE_GENERATED_PASSWORD>"
}
```

The name of the parameter is `keycloak-password`, but really it's just formatting the resource name with a `-password` suffix. For more information, see Safe storage of app secrets in development in ASP.NET Core and Add Keycloak resource.

The WithReference method configures a connection in the `ExampleProject` named `keycloak` and the WaitFor instructs the app host to not start the dependant service until the `keycloak` resource is ready.

> **Tip**

If you'd rather connect to an existing Keycloak instance, call AddConnectionString instead. For more information, see Reference existing resources.

Add Keycloak resource with data volume

To add a data volume to the Keycloak resource, call the WithDataVolume method on the Keycloak resource:

C#

```
var keycloak = builder.AddKeycloak("keycloak", 8080)
                      .WithDataVolume();

var apiService = builder.AddProject<Projects.Keycloak_ApiService>
("apiservice")
                        .WithReference(keycloak)
                        .WaitFor(keycloak);

builder.AddProject<Projects.Keycloak_Web>("webfrontend")
       .WithExternalHttpEndpoints()
       .WithReference(keycloak)
       .WithReference(apiService)
       .WaitFor(apiService);

// After adding all resources, run the app...
```

The data volume is used to persist the Keycloak data outside the lifecycle of its container. The data volume is mounted at the `/opt/keycloak/data` path in the Keycloak container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

> ⚠ **Warning**
>
> The admin credentials are stored in the data volume. When using a data volume and if the credentials change, it will not work until you delete the volume.

Add Keycloak resource with data bind mount

To add a data bind mount to the Keycloak resource, call the WithDataBindMount method:

C#

```
var keycloak = builder.AddKeycloak("keycloak", 8080)
                      .WithDataBindMount(@"C:\Keycloak\Data");

var apiService = builder.AddProject<Projects.Keycloak_ApiService>
("apiservice")
                        .WithReference(keycloak)
                        .WaitFor(keycloak);

builder.AddProject<Projects.Keycloak_Web>("webfrontend")
       .WithExternalHttpEndpoints()
       .WithReference(keycloak)
       .WithReference(apiService)
       .WaitFor(apiService);

// After adding all resources, run the app...
```

> **ⓘ Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Keycloak data across container restarts. The data bind mount is mounted at the `C:\Keycloak\Data` on Windows (or `/Keycloak/Data` on Unix) path on the host machine in the Keycloak container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add Keycloak resource with parameters

When you want to explicitly provide the admin username and password used by the container image, you can provide these credentials as parameters. Consider the following alternative example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var username = builder.AddParameter("username");
var password = builder.AddParameter("password", secret: true);

var keycloak = builder.AddKeycloak("keycloak", 8080, username, password);

var apiService = builder.AddProject<Projects.Keycloak_ApiService>
```

```
("apiservice")
                        .WithReference(keycloak)
                        .WaitFor(keycloak);

builder.AddProject<Projects.Keycloak_Web>("webfrontend")
       .WithExternalHttpEndpoints()
       .WithReference(keycloak)
       .WithReference(apiService)
       .WaitFor(apiService);

// After adding all resources, run the app...
```

The `username` and `password` parameters are usually provided as environment variables or secrets. The parameters are used to set the `KEYCLOAK_ADMIN` and `KEYCLOAK_ADMIN_PASSWORD` environment variables in the container. For more information on providing parameters, see External parameters.

Add Keycloak resource with realm import

To import a realm into Keycloak, call the WithRealmImport method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var keycloak = builder.AddKeycloak("keycloak", 8080)
                      .WithDataVolume()
                      .WithRealmImport("./Realms");

var apiService = builder.AddProject<Projects.AspireApp_ApiService>
("apiservice")
                        .WithReference(keycloak)
                        .WaitFor(keycloak);

builder.AddProject<Projects.AspireApp_Web>("webfrontend")
       .WithExternalHttpEndpoints()
       .WithReference(keycloak)
       .WithReference(apiService)
       .WaitFor(apiService);

builder.Build().Run();
```

The realm import files are mounted at `/opt/keycloak/data/import` in the Keycloak container. Realm import files are JSON files that represent the realm configuration. For more information on realm import, see Keycloak docs: Importing a realm.

As an example, the following JSON file could be added to the app host project in a */Realms* folder—to serve as a source realm configuration file:

JSON


```
{
    "id": "86683c73-be28-4380-a014-6316c0404192",
    "realm": "WeatherShop",
    "notBefore": 0,
    "defaultSignatureAlgorithm": "RS256",
    "revokeRefreshToken": false,
    "refreshTokenMaxReuse": 0,
    "accessTokenLifespan": 300,
    "accessTokenLifespanForImplicitFlow": 900,
    "ssoSessionIdleTimeout": 1800,
    "ssoSessionMaxLifespan": 36000,
    "ssoSessionIdleTimeoutRememberMe": 0,
    "ssoSessionMaxLifespanRememberMe": 0,
    "offlineSessionIdleTimeout": 2592000,
    "offlineSessionMaxLifespanEnabled": false,
    "offlineSessionMaxLifespan": 5184000,
    "clientSessionIdleTimeout": 0,
    "clientSessionMaxLifespan": 0,
    "clientOfflineSessionIdleTimeout": 0,
    "clientOfflineSessionMaxLifespan": 0,
    "accessCodeLifespan": 60,
    "accessCodeLifespanUserAction": 300,
    "accessCodeLifespanLogin": 1800,
    "actionTokenGeneratedByAdminLifespan": 43200,
    "actionTokenGeneratedByUserLifespan": 300,
    "oauth2DeviceCodeLifespan": 600,
    "oauth2DevicePollingInterval": 5,
    "enabled": true,
    "sslRequired": "external",
    "registrationAllowed": true,
    "registrationEmailAsUsername": false,
    "rememberMe": false,
    "verifyEmail": false,
    "loginWithEmailAllowed": true,
    "duplicateEmailsAllowed": false,
    "resetPasswordAllowed": false,
    "editUsernameAllowed": false,
    "bruteForceProtected": false,
    "permanentLockout": false,
    "maxTemporaryLockouts": 0,
    "maxFailureWaitSeconds": 900,
    "minimumQuickLoginWaitSeconds": 60,
    "waitIncrementSeconds": 60,
    "quickLoginCheckMilliSeconds": 1000,
    "maxDeltaTimeSeconds": 43200,
    "failureFactor": 30,
    "roles": {
        "realm": [
            {
                "id": "79e15e0c-7084-4595-9066-c852bc5a6aca",
                "name": "uma_authorization",
                "description": "${role_uma_authorization}",
```

```
                "composite": false,
                "clientRole": false,
                "containerId": "86683c73-be28-4380-a014-6316c0404192",
                "attributes": {}
            },
            {
                "id": "f2bd959d-ed9d-4409-af6d-206a4a52cc23",
                "name": "default-roles-weathershop",
                "description": "${role_default-roles}",
                "composite": true,
                "composites": {
                    "realm": [ "offline_access", "uma_authorization" ],
                    "client": {
                        "account": [ "view-profile", "manage-account" ]
                    }
                },
                "clientRole": false,
                "containerId": "86683c73-be28-4380-a014-6316c0404192",
                "attributes": {}
            },
            {
                "id": "5e1d3cf6-c7ac-478d-a70c-4299abf58490",
                "name": "offline_access",
                "description": "${role_offline-access}",
                "composite": false,
                "clientRole": false,
                "containerId": "86683c73-be28-4380-a014-6316c0404192",
                "attributes": {}
            }
        ],
        "client": {
            "realm-management": [
                {
                    "id": "fe6e42fe-8629-40da-9afe-1179fc964988",
                    "name": "manage-users",
                    "description": "${role_manage-users}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
                    "attributes": {}
                },
                {
                    "id": "f82abb6c-239c-4533-afbd-a7aa03937204",
                    "name": "view-users",
                    "description": "${role_view-users}",
                    "composite": true,
                    "composites": {
                        "client": {
                            "realm-management": [ "query-groups", "query-
users" ]
                        }
                    },
                    "clientRole": true,
                    "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
                    "attributes": {}
```

```
                },
                {
                    "id": "1eb57351-1302-45e5-924a-9b0dc337a2bb",
                    "name": "view-events",
                    "description": "${role_view-events}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
                    "attributes": {}
                },
                {
                    "id": "df3a077e-9bd4-4924-8281-cab7c7fd73e3",
                    "name": "manage-authorization",
                    "description": "${role_manage-authorization}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
                    "attributes": {}
                },
                {
                    "id": "d9fcb43a-3bad-492c-9af9-f199a6382064",
                    "name": "query-groups",
                    "description": "${role_query-groups}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
                    "attributes": {}
                },
                {
                    "id": "1dbdaf2b-a29c-4c54-86b7-d7c338e7672f",
                    "name": "realm-admin",
                    "description": "${role_realm-admin}",
                    "composite": true,
                    "composites": {
                        "client": {
                            "realm-management": [ "view-users", "manage-
users", "view-events", "query-groups", "manage-authorization", "query-
users", "manage-realm", "view-identity-providers", "create-client", "view-
authorization", "query-clients", "view-clients", "query-realms",
"impersonation", "view-realm", "manage-events", "manage-identity-providers",
"manage-clients" ]
                        }
                    },
                    "clientRole": true,
                    "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
                    "attributes": {}
                },
                {
                    "id": "4c1ff7e3-cc1d-4b1d-a88f-fb71416c742a",
                    "name": "query-users",
                    "description": "${role_query-users}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
                    "attributes": {}
```

```
                },
                {
                    "id": "284b35f8-5bc2-4482-8769-81d3594df5a3",
                    "name": "manage-realm",
                    "description": "${role_manage-realm}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
                    "attributes": {}
                },
                {
                    "id": "7e872c38-8a22-469f-92ca-ec67e95d3c33",
                    "name": "view-identity-providers",
                    "description": "${role_view-identity-providers}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
                    "attributes": {}
                },
                {
                    "id": "9f4c2563-7575-461e-b2c8-b2b87f314cb9",
                    "name": "create-client",
                    "description": "${role_create-client}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
                    "attributes": {}
                },
                {
                    "id": "8f92f45c-bfa0-4a66-9812-334fe223c8be",
                    "name": "query-clients",
                    "description": "${role_query-clients}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
                    "attributes": {}
                },
                {
                    "id": "2a0143cf-ad90-4f68-bcb2-a50aa358b070",
                    "name": "view-authorization",
                    "description": "${role_view-authorization}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
                    "attributes": {}
                },
                {
                    "id": "95cc13bf-1342-445a-99fd-141522a7e777",
                    "name": "view-clients",
                    "description": "${role_view-clients}",
                    "composite": true,
                    "composites": {
                        "client": {
                            "realm-management": [ "query-clients" ]
                        }
```

```
            },
            "clientRole": true,
            "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
            "attributes": {}
        },
        {
            "id": "109f4b83-ba7d-4036-91e7-7e169cd4c30c",
            "name": "query-realms",
            "description": "${role_query-realms}",
            "composite": false,
            "clientRole": true,
            "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
            "attributes": {}
        },
        {
            "id": "17bcb2b7-3a35-4089-85ea-1d034303b5d6",
            "name": "impersonation",
            "description": "${role_impersonation}",
            "composite": false,
            "clientRole": true,
            "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
            "attributes": {}
        },
        {
            "id": "21c51846-5f22-4318-82b7-9e64e2d256f4",
            "name": "view-realm",
            "description": "${role_view-realm}",
            "composite": false,
            "clientRole": true,
            "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
            "attributes": {}
        },
        {
            "id": "a0599e32-b53b-43bf-a7f6-ac0507ed277d",
            "name": "manage-events",
            "description": "${role_manage-events}",
            "composite": false,
            "clientRole": true,
            "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
            "attributes": {}
        },
        {
            "id": "e732e665-efb7-4df0-8843-b22bf2fe4717",
            "name": "manage-identity-providers",
            "description": "${role_manage-identity-providers}",
            "composite": false,
            "clientRole": true,
            "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
            "attributes": {}
        },
        {
            "id": "6d1b10f2-4c51-4279-8418-d4b82c17f203",
            "name": "manage-clients",
            "description": "${role_manage-clients}",
            "composite": false,
```

```
                    "clientRole": true,
                    "containerId": "0aa2db92-8cc4-490f-a084-55f5b889613a",
                    "attributes": {}
                }
            ],
            "WeatherWeb": [],
            "security-admin-console": [],
            "admin-cli": [],
            "account-console": [],
            "broker": [
                {
                    "id": "7184260f-55c4-454a-bf67-dade5b74df7e",
                    "name": "read-token",
                    "description": "${role_read-token}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "db2ab30c-b83b-499e-9545-decdc906a372",
                    "attributes": {}
                }
            ],
            "Postman": [],
            "weather.api": [],
            "account": [
                {
                    "id": "b4a01a53-3ed0-4e96-8fd1-efb0c143a45d",
                    "name": "view-groups",
                    "description": "${role_view-groups}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "65816a45-48d3-4856-b052-c65cb03881d3",
                    "attributes": {}
                },
                {
                    "id": "e9af1e5f-c0a5-4515-a77a-38fec79135d0",
                    "name": "delete-account",
                    "description": "${role_delete-account}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "65816a45-48d3-4856-b052-c65cb03881d3",
                    "attributes": {}
                },
                {
                    "id": "526cc4f7-6cf8-4f2b-8241-de0e60d2fd47",
                    "name": "manage-consent",
                    "description": "${role_manage-consent}",
                    "composite": true,
                    "composites": {
                        "client": {
                            "account": [ "view-consent" ]
                        }
                    },
                    "clientRole": true,
                    "containerId": "65816a45-48d3-4856-b052-c65cb03881d3",
                    "attributes": {}
                },
```

```
                {
                    "id": "cf93d42f-ffd9-4b3f-bf8d-55aa934f2fe3",
                    "name": "view-applications",
                    "description": "${role_view-applications}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "65816a45-48d3-4856-b052-c65cb03881d3",
                    "attributes": {}
                },
                {
                    "id": "f3b44155-fe06-4fea-8b8f-6954f54d48bb",
                    "name": "view-profile",
                    "description": "${role_view-profile}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "65816a45-48d3-4856-b052-c65cb03881d3",
                    "attributes": {}
                },
                {
                    "id": "65a74b6a-a00f-46b6-8ead-6c051e78c37e",
                    "name": "manage-account",
                    "description": "${role_manage-account}",
                    "composite": true,
                    "composites": {
                        "client": {
                            "account": [ "manage-account-links" ]
                        }
                    },
                    "clientRole": true,
                    "containerId": "65816a45-48d3-4856-b052-c65cb03881d3",
                    "attributes": {}
                },
                {
                    "id": "c914cc47-8a49-4f30-9851-6f639c4e7adf",
                    "name": "manage-account-links",
                    "description": "${role_manage-account-links}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "65816a45-48d3-4856-b052-c65cb03881d3",
                    "attributes": {}
                },
                {
                    "id": "261d0db4-28c7-4900-a156-01ab4e2483e5",
                    "name": "view-consent",
                    "description": "${role_view-consent}",
                    "composite": false,
                    "clientRole": true,
                    "containerId": "65816a45-48d3-4856-b052-c65cb03881d3",
                    "attributes": {}
                }
            ]
        }
    },
    "groups": [],
    "defaultRole": {
```

```
        "id": "f2bd959d-ed9d-4409-af6d-206a4a52cc23",
        "name": "default-roles-weathershop",
        "description": "${role_default-roles}",
        "composite": true,
        "clientRole": false,
        "containerId": "86683c73-be28-4380-a014-6316c0404192"
    },
    "requiredCredentials": [ "password" ],
    "otpPolicyType": "totp",
    "otpPolicyAlgorithm": "HmacSHA1",
    "otpPolicyInitialCounter": 0,
    "otpPolicyDigits": 6,
    "otpPolicyLookAheadWindow": 1,
    "otpPolicyPeriod": 30,
    "otpPolicyCodeReusable": false,
    "otpSupportedApplications": [ "totpAppFreeOTPName", "totpAppGoogleName",
"totpAppMicrosoftAuthenticatorName" ],
    "localizationTexts": {},
    "webAuthnPolicyRpEntityName": "keycloak",
    "webAuthnPolicySignatureAlgorithms": [ "ES256" ],
    "webAuthnPolicyRpId": "",
    "webAuthnPolicyAttestationConveyancePreference": "not specified",
    "webAuthnPolicyAuthenticatorAttachment": "not specified",
    "webAuthnPolicyRequireResidentKey": "not specified",
    "webAuthnPolicyUserVerificationRequirement": "not specified",
    "webAuthnPolicyCreateTimeout": 0,
    "webAuthnPolicyAvoidSameAuthenticatorRegister": false,
    "webAuthnPolicyAcceptableAaguids": [],
    "webAuthnPolicyExtraOrigins": [],
    "webAuthnPolicyPasswordlessRpEntityName": "keycloak",
    "webAuthnPolicyPasswordlessSignatureAlgorithms": [ "ES256" ],
    "webAuthnPolicyPasswordlessRpId": "",
    "webAuthnPolicyPasswordlessAttestationConveyancePreference": "not
specified",
    "webAuthnPolicyPasswordlessAuthenticatorAttachment": "not specified",
    "webAuthnPolicyPasswordlessRequireResidentKey": "not specified",
    "webAuthnPolicyPasswordlessUserVerificationRequirement": "not
specified",
    "webAuthnPolicyPasswordlessCreateTimeout": 0,
    "webAuthnPolicyPasswordlessAvoidSameAuthenticatorRegister": false,
    "webAuthnPolicyPasswordlessAcceptableAaguids": [],
    "webAuthnPolicyPasswordlessExtraOrigins": [],
    "scopeMappings": [
        {
            "clientScope": "offline_access",
            "roles": [ "offline_access" ]
        }
    ],
    "clientScopeMappings": {
        "account": [
            {
                "client": "account-console",
                "roles": [ "manage-account", "view-groups" ]
            }
        ]
```

```
    },
    "clients": [
        {
            "id": "bd03dd61-71bf-4f50-acfa-bfc2444ee1d2",
            "clientId": "Postman",
            "name": "",
            "description": "",
            "rootUrl": "",
            "adminUrl": "",
            "baseUrl": "",
            "surrogateAuthRequired": false,
            "enabled": true,
            "alwaysDisplayInConsole": false,
            "clientAuthenticatorType": "client-secret",
            "redirectUris": [ "https://oauth.pstmn.io/v1/callback" ],
            "webOrigins": [ "https://oauth.pstmn.io" ],
            "notBefore": 0,
            "bearerOnly": false,
            "consentRequired": false,
            "standardFlowEnabled": true,
            "implicitFlowEnabled": false,
            "directAccessGrantsEnabled": false,
            "serviceAccountsEnabled": false,
            "publicClient": true,
            "frontchannelLogout": true,
            "protocol": "openid-connect",
            "attributes": {
                "oidc.ciba.grant.enabled": "false",
                "client.secret.creation.time": "1718111570",
                "backchannel.logout.session.required": "true",
                "post.logout.redirect.uris": "+",
                "oauth2.device.authorization.grant.enabled": "false",
                "display.on.consent.screen": "false",
                "backchannel.logout.revoke.offline.tokens": "false"
            },
            "authenticationFlowBindingOverrides": {},
            "fullScopeAllowed": true,
            "nodeReRegistrationTimeout": -1,
            "defaultClientScopes": [ "web-origins", "acr", "profile",
"roles", "email" ],
            "optionalClientScopes": [ "address", "phone", "offline_access",
"weather:all", "microprofile-jwt" ]
        },
        {
            "id": "016c17d1-8e0f-4a67-9116-86b4691ba99c",
            "clientId": "WeatherWeb",
            "name": "",
            "description": "",
            "rootUrl": "",
            "adminUrl": "",
            "baseUrl": "",
            "surrogateAuthRequired": false,
            "enabled": true,
            "alwaysDisplayInConsole": false,
            "clientAuthenticatorType": "client-secret",
```

```
            "redirectUris": [ "https://localhost:7085/signin-oidc" ],
            "webOrigins": [ "https://localhost:7085" ],
            "notBefore": 0,
            "bearerOnly": false,
            "consentRequired": false,
            "standardFlowEnabled": true,
            "implicitFlowEnabled": false,
            "directAccessGrantsEnabled": false,
            "serviceAccountsEnabled": false,
            "publicClient": true,
            "frontchannelLogout": true,
            "protocol": "openid-connect",
            "attributes": {
                "oidc.ciba.grant.enabled": "false",
                "post.logout.redirect.uris":
"https://localhost:7085/signout-callback-oidc",
                "oauth2.device.authorization.grant.enabled": "false",
                "backchannel.logout.session.required": "true",
                "backchannel.logout.revoke.offline.tokens": "false"
            },
            "authenticationFlowBindingOverrides": {},
            "fullScopeAllowed": true,
            "nodeReRegistrationTimeout": -1,
            "defaultClientScopes": [ "web-origins", "acr", "profile",
"roles", "email" ],
            "optionalClientScopes": [ "address", "phone", "offline_access",
"weather:all", "microprofile-jwt" ]
        },
        {
            "id": "65816a45-48d3-4856-b052-c65cb03881d3",
            "clientId": "account",
            "name": "${client_account}",
            "rootUrl": "${authBaseUrl}",
            "baseUrl": "/realms/WeatherShop/account/",
            "surrogateAuthRequired": false,
            "enabled": true,
            "alwaysDisplayInConsole": false,
            "clientAuthenticatorType": "client-secret",
            "redirectUris": [ "/realms/WeatherShop/account/*" ],
            "webOrigins": [],
            "notBefore": 0,
            "bearerOnly": false,
            "consentRequired": false,
            "standardFlowEnabled": true,
            "implicitFlowEnabled": false,
            "directAccessGrantsEnabled": false,
            "serviceAccountsEnabled": false,
            "publicClient": true,
            "frontchannelLogout": false,
            "protocol": "openid-connect",
            "attributes": {
                "post.logout.redirect.uris": "+"
            },
            "authenticationFlowBindingOverrides": {},
            "fullScopeAllowed": false,
```

```
            "nodeReRegistrationTimeout": 0,
            "defaultClientScopes": [ "web-origins", "acr", "profile",
"roles", "email" ],
            "optionalClientScopes": [ "address", "phone", "offline_access",
"microprofile-jwt" ]
        },
        {
            "id": "437fda77-3ba1-4d7b-b192-808e4e62833b",
            "clientId": "account-console",
            "name": "${client_account-console}",
            "rootUrl": "${authBaseUrl}",
            "baseUrl": "/realms/WeatherShop/account/",
            "surrogateAuthRequired": false,
            "enabled": true,
            "alwaysDisplayInConsole": false,
            "clientAuthenticatorType": "client-secret",
            "redirectUris": [ "/realms/WeatherShop/account/*" ],
            "webOrigins": [],
            "notBefore": 0,
            "bearerOnly": false,
            "consentRequired": false,
            "standardFlowEnabled": true,
            "implicitFlowEnabled": false,
            "directAccessGrantsEnabled": false,
            "serviceAccountsEnabled": false,
            "publicClient": true,
            "frontchannelLogout": false,
            "protocol": "openid-connect",
            "attributes": {
                "post.logout.redirect.uris": "+",
                "pkce.code.challenge.method": "S256"
            },
            "authenticationFlowBindingOverrides": {},
            "fullScopeAllowed": false,
            "nodeReRegistrationTimeout": 0,
            "protocolMappers": [
                {
                    "id": "e4606d8a-a581-402c-9290-4e3b988f2090",
                    "name": "audience resolve",
                    "protocol": "openid-connect",
                    "protocolMapper": "oidc-audience-resolve-mapper",
                    "consentRequired": false,
                    "config": {}
                }
            ],
            "defaultClientScopes": [ "web-origins", "acr", "profile",
"roles", "email" ],
            "optionalClientScopes": [ "address", "phone", "offline_access",
"microprofile-jwt" ]
        },
        {
            "id": "f13fd042-6931-4032-a0ba-f63b364f8d56",
            "clientId": "admin-cli",
            "name": "${client_admin-cli}",
            "surrogateAuthRequired": false,
```

```
            "enabled": true,
            "alwaysDisplayInConsole": false,
            "clientAuthenticatorType": "client-secret",
            "redirectUris": [],
            "webOrigins": [],
            "notBefore": 0,
            "bearerOnly": false,
            "consentRequired": false,
            "standardFlowEnabled": false,
            "implicitFlowEnabled": false,
            "directAccessGrantsEnabled": true,
            "serviceAccountsEnabled": false,
            "publicClient": true,
            "frontchannelLogout": false,
            "protocol": "openid-connect",
            "attributes": {
                "post.logout.redirect.uris": "+"
            },
            "authenticationFlowBindingOverrides": {},
            "fullScopeAllowed": false,
            "nodeReRegistrationTimeout": 0,
            "defaultClientScopes": [ "web-origins", "acr", "profile",
"roles", "email" ],
            "optionalClientScopes": [ "address", "phone", "offline_access",
"microprofile-jwt" ]
        },
        {
            "id": "db2ab30c-b83b-499e-9545-decdc906a372",
            "clientId": "broker",
            "name": "${client_broker}",
            "surrogateAuthRequired": false,
            "enabled": true,
            "alwaysDisplayInConsole": false,
            "clientAuthenticatorType": "client-secret",
            "redirectUris": [],
            "webOrigins": [],
            "notBefore": 0,
            "bearerOnly": true,
            "consentRequired": false,
            "standardFlowEnabled": true,
            "implicitFlowEnabled": false,
            "directAccessGrantsEnabled": false,
            "serviceAccountsEnabled": false,
            "publicClient": false,
            "frontchannelLogout": false,
            "protocol": "openid-connect",
            "attributes": {
                "post.logout.redirect.uris": "+"
            },
            "authenticationFlowBindingOverrides": {},
            "fullScopeAllowed": false,
            "nodeReRegistrationTimeout": 0,
            "defaultClientScopes": [ "web-origins", "acr", "profile",
"roles", "email" ],
            "optionalClientScopes": [ "address", "phone", "offline_access",
```

```
"microprofile-jwt" ]
        },
        {
            "id": "0aa2db92-8cc4-490f-a084-55f5b889613a",
            "clientId": "realm-management",
            "name": "${client_realm-management}",
            "surrogateAuthRequired": false,
            "enabled": true,
            "alwaysDisplayInConsole": false,
            "clientAuthenticatorType": "client-secret",
            "redirectUris": [],
            "webOrigins": [],
            "notBefore": 0,
            "bearerOnly": true,
            "consentRequired": false,
            "standardFlowEnabled": true,
            "implicitFlowEnabled": false,
            "directAccessGrantsEnabled": false,
            "serviceAccountsEnabled": false,
            "publicClient": false,
            "frontchannelLogout": false,
            "protocol": "openid-connect",
            "attributes": {
                "post.logout.redirect.uris": "+"
            },
            "authenticationFlowBindingOverrides": {},
            "fullScopeAllowed": false,
            "nodeReRegistrationTimeout": 0,
            "defaultClientScopes": [ "web-origins", "acr", "profile",
"roles", "email" ],
            "optionalClientScopes": [ "address", "phone", "offline_access",
"microprofile-jwt" ]
        },
        {
            "id": "e0cc9cef-924e-4799-a921-4811f3bb5d65",
            "clientId": "security-admin-console",
            "name": "${client_security-admin-console}",
            "rootUrl": "${authAdminUrl}",
            "baseUrl": "/admin/WeatherShop/console/",
            "surrogateAuthRequired": false,
            "enabled": true,
            "alwaysDisplayInConsole": false,
            "clientAuthenticatorType": "client-secret",
            "redirectUris": [ "/admin/WeatherShop/console/*" ],
            "webOrigins": [ "+" ],
            "notBefore": 0,
            "bearerOnly": false,
            "consentRequired": false,
            "standardFlowEnabled": true,
            "implicitFlowEnabled": false,
            "directAccessGrantsEnabled": false,
            "serviceAccountsEnabled": false,
            "publicClient": true,
            "frontchannelLogout": false,
            "protocol": "openid-connect",
```

```
            "attributes": {
                "post.logout.redirect.uris": "+",
                "pkce.code.challenge.method": "S256"
            },
            "authenticationFlowBindingOverrides": {},
            "fullScopeAllowed": false,
            "nodeReRegistrationTimeout": 0,
            "protocolMappers": [
                {
                    "id": "254ac20c-6701-4095-82c6-6abd6669b9de",
                    "name": "locale",
                    "protocol": "openid-connect",
                    "protocolMapper": "oidc-usermodel-attribute-mapper",
                    "consentRequired": false,
                    "config": {
                        "introspection.token.claim": "true",
                        "userinfo.token.claim": "true",
                        "user.attribute": "locale",
                        "id.token.claim": "true",
                        "access.token.claim": "true",
                        "claim.name": "locale",
                        "jsonType.label": "String"
                    }
                }
            ],
            "defaultClientScopes": [ "web-origins", "acr", "profile",
"roles", "email" ],
            "optionalClientScopes": [ "address", "phone", "offline_access",
"microprofile-jwt" ]
        },
        {
            "id": "4b5953fd-b218-41be-b061-58f37c1c7d26",
            "clientId": "weather.api",
            "name": "",
            "description": "",
            "rootUrl": "",
            "adminUrl": "",
            "baseUrl": "",
            "surrogateAuthRequired": false,
            "enabled": true,
            "alwaysDisplayInConsole": false,
            "clientAuthenticatorType": "client-secret",
            "secret": "**********",
            "redirectUris": [ "/*" ],
            "webOrigins": [ "/*" ],
            "notBefore": 0,
            "bearerOnly": false,
            "consentRequired": false,
            "standardFlowEnabled": true,
            "implicitFlowEnabled": false,
            "directAccessGrantsEnabled": false,
            "serviceAccountsEnabled": false,
            "publicClient": false,
            "frontchannelLogout": true,
            "protocol": "openid-connect",
```

```
            "attributes": {
                "oidc.ciba.grant.enabled": "false",
                "client.secret.creation.time": "1718111354",
                "backchannel.logout.session.required": "true",
                "post.logout.redirect.uris": "+",
                "oauth2.device.authorization.grant.enabled": "false",
                "backchannel.logout.revoke.offline.tokens": "false"
            },
            "authenticationFlowBindingOverrides": {},
            "fullScopeAllowed": true,
            "nodeReRegistrationTimeout": -1,
            "defaultClientScopes": [ "web-origins", "acr", "profile",
"roles", "email" ],
            "optionalClientScopes": [ "address", "phone", "offline_access",
"microprofile-jwt" ]
        }
    ],
    "clientScopes": [
        {
            "id": "2a6322a2-2f6a-469f-b3c7-d0922db4ad46",
            "name": "phone",
            "description": "OpenID Connect built-in scope: phone",
            "protocol": "openid-connect",
            "attributes": {
                "include.in.token.scope": "true",
                "display.on.consent.screen": "true",
                "consent.screen.text": "${phoneScopeConsentText}"
            },
            "protocolMappers": [
                {
                    "id": "ab515c55-d65b-42d3-9d3c-18921a8df065",
                    "name": "phone number",
                    "protocol": "openid-connect",
                    "protocolMapper": "oidc-usermodel-attribute-mapper",
                    "consentRequired": false,
                    "config": {
                        "introspection.token.claim": "true",
                        "userinfo.token.claim": "true",
                        "user.attribute": "phoneNumber",
                        "id.token.claim": "true",
                        "access.token.claim": "true",
                        "claim.name": "phone_number",
                        "jsonType.label": "String"
                    }
                },
                {
                    "id": "bfaa5db4-137c-4824-bfae-ed77762872c2",
                    "name": "phone number verified",
                    "protocol": "openid-connect",
                    "protocolMapper": "oidc-usermodel-attribute-mapper",
                    "consentRequired": false,
                    "config": {
                        "introspection.token.claim": "true",
                        "userinfo.token.claim": "true",
                        "user.attribute": "phoneNumberVerified",
```

```
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "phone_number_verified",
                    "jsonType.label": "boolean"
                }
            }
        ]
    },
    {
        "id": "52fc55cb-995e-4aa2-95ae-3b3d6601dc41",
        "name": "weather:all",
        "description": "",
        "protocol": "openid-connect",
        "attributes": {
            "include.in.token.scope": "true",
            "display.on.consent.screen": "true",
            "gui.order": "",
            "consent.screen.text": ""
        },
        "protocolMappers": [
            {
                "id": "06d03e02-1e56-4bde-911d-bcf28aeba90f",
                "name": "weather api audience",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-audience-mapper",
                "consentRequired": false,
                "config": {
                    "included.client.audience": "weather.api",
                    "introspection.token.claim": "true",
                    "userinfo.token.claim": "false",
                    "id.token.claim": "false",
                    "lightweight.claim": "false",
                    "access.token.claim": "true"
                }
            }
        ]
    },
    {
        "id": "292ded65-c85e-4c56-ad4d-8e886b9bb261",
        "name": "email",
        "description": "OpenID Connect built-in scope: email",
        "protocol": "openid-connect",
        "attributes": {
            "include.in.token.scope": "true",
            "display.on.consent.screen": "true",
            "consent.screen.text": "${emailScopeConsentText}"
        },
        "protocolMappers": [
            {
                "id": "42743882-7e0a-455e-b6a3-794ec8bf0f22",
                "name": "email verified",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-usermodel-property-mapper",
                "consentRequired": false,
                "config": {
```

```
                    "introspection.token.claim": "true",
                    "userinfo.token.claim": "true",
                    "user.attribute": "emailVerified",
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "email_verified",
                    "jsonType.label": "boolean"
                }
            },
            {
                "id": "b6dd2af9-e583-4d01-95fa-3f0db3ab0129",
                "name": "email",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-usermodel-attribute-mapper",
                "consentRequired": false,
                "config": {
                    "introspection.token.claim": "true",
                    "userinfo.token.claim": "true",
                    "user.attribute": "email",
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "email",
                    "jsonType.label": "String"
                }
            }
        ]
    },
    {
        "id": "09a76939-4997-49f5-b88e-dfe54a2819f5",
        "name": "offline_access",
        "description": "OpenID Connect built-in scope: offline_access",
        "protocol": "openid-connect",
        "attributes": {
            "consent.screen.text": "${offlineAccessScopeConsentText}",
            "display.on.consent.screen": "true"
        }
    },
    {
        "id": "95ff8627-716e-49f0-b960-52185409d628",
        "name": "profile",
        "description": "OpenID Connect built-in scope: profile",
        "protocol": "openid-connect",
        "attributes": {
            "include.in.token.scope": "true",
            "display.on.consent.screen": "true",
            "consent.screen.text": "${profileScopeConsentText}"
        },
        "protocolMappers": [
            {
                "id": "b1ae43d1-9d52-40cd-9c6a-a8557ea63f9a",
                "name": "picture",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-usermodel-attribute-mapper",
                "consentRequired": false,
                "config": {
```

```
                    "introspection.token.claim": "true",
                    "userinfo.token.claim": "true",
                    "user.attribute": "picture",
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "picture",
                    "jsonType.label": "String"
                }
            },
            {
                "id": "b9e09b82-3e67-4175-b34a-419b24a13a7f",
                "name": "zoneinfo",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-usermodel-attribute-mapper",
                "consentRequired": false,
                "config": {
                    "introspection.token.claim": "true",
                    "userinfo.token.claim": "true",
                    "user.attribute": "zoneinfo",
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "zoneinfo",
                    "jsonType.label": "String"
                }
            },
            {
                "id": "06e88533-d2cc-4ae3-a25a-a17e93f69dee",
                "name": "nickname",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-usermodel-attribute-mapper",
                "consentRequired": false,
                "config": {
                    "introspection.token.claim": "true",
                    "userinfo.token.claim": "true",
                    "user.attribute": "nickname",
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "nickname",
                    "jsonType.label": "String"
                }
            },
            {
                "id": "b697c055-2fb9-4985-8919-33d9f524eaa9",
                "name": "full name",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-full-name-mapper",
                "consentRequired": false,
                "config": {
                    "id.token.claim": "true",
                    "introspection.token.claim": "true",
                    "access.token.claim": "true",
                    "userinfo.token.claim": "true"
                }
            },
            {
```

```
                "id": "70663048-2110-4271-a8f4-105e77fe2905",
                "name": "profile",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-usermodel-attribute-mapper",
                "consentRequired": false,
                "config": {
                    "introspection.token.claim": "true",
                    "userinfo.token.claim": "true",
                    "user.attribute": "profile",
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "profile",
                    "jsonType.label": "String"
                }
            },
            {
                "id": "85c992ed-4971-41f1-a4b8-c6263b29dff8",
                "name": "website",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-usermodel-attribute-mapper",
                "consentRequired": false,
                "config": {
                    "introspection.token.claim": "true",
                    "userinfo.token.claim": "true",
                    "user.attribute": "website",
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "website",
                    "jsonType.label": "String"
                }
            },
            {
                "id": "49e1d494-a72e-49de-a5de-8c2ad752205c",
                "name": "birthdate",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-usermodel-attribute-mapper",
                "consentRequired": false,
                "config": {
                    "introspection.token.claim": "true",
                    "userinfo.token.claim": "true",
                    "user.attribute": "birthdate",
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "birthdate",
                    "jsonType.label": "String"
                }
            },
            {
                "id": "a746724f-7622-4ad4-91ef-811da6c735ad",
                "name": "updated at",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-usermodel-attribute-mapper",
                "consentRequired": false,
                "config": {
                    "introspection.token.claim": "true",
```

```
                    "userinfo.token.claim": "true",
                    "user.attribute": "updatedAt",
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "updated_at",
                    "jsonType.label": "long"
                }
            },
            {
                "id": "d647d13f-9f96-49f3-b32a-62ad63c37d0e",
                "name": "gender",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-usermodel-attribute-mapper",
                "consentRequired": false,
                "config": {
                    "introspection.token.claim": "true",
                    "userinfo.token.claim": "true",
                    "user.attribute": "gender",
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "gender",
                    "jsonType.label": "String"
                }
            },
            {
                "id": "86204f6f-16ce-4c9f-9fca-f66b3f292554",
                "name": "given name",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-usermodel-attribute-mapper",
                "consentRequired": false,
                "config": {
                    "introspection.token.claim": "true",
                    "userinfo.token.claim": "true",
                    "user.attribute": "firstName",
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "given_name",
                    "jsonType.label": "String"
                }
            },
            {
                "id": "f4005bd9-18d2-4456-9e7c-98c5a637f063",
                "name": "locale",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-usermodel-attribute-mapper",
                "consentRequired": false,
                "config": {
                    "introspection.token.claim": "true",
                    "userinfo.token.claim": "true",
                    "user.attribute": "locale",
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "locale",
                    "jsonType.label": "String"
                }
```

```
            },
            {
                "id": "04b20a5a-1588-476c-a465-26a691320510",
                "name": "family name",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-usermodel-attribute-mapper",
                "consentRequired": false,
                "config": {
                    "introspection.token.claim": "true",
                    "userinfo.token.claim": "true",
                    "user.attribute": "lastName",
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "family_name",
                    "jsonType.label": "String"
                }
            },
            {
                "id": "f812580a-7863-44c3-bcf0-c3f441f0194e",
                "name": "middle name",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-usermodel-attribute-mapper",
                "consentRequired": false,
                "config": {
                    "introspection.token.claim": "true",
                    "userinfo.token.claim": "true",
                    "user.attribute": "middleName",
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "middle_name",
                    "jsonType.label": "String"
                }
            },
            {
                "id": "c2ae83c9-62c2-4a65-adef-c1d6fd126ec4",
                "name": "username",
                "protocol": "openid-connect",
                "protocolMapper": "oidc-usermodel-attribute-mapper",
                "consentRequired": false,
                "config": {
                    "introspection.token.claim": "true",
                    "userinfo.token.claim": "true",
                    "user.attribute": "username",
                    "id.token.claim": "true",
                    "access.token.claim": "true",
                    "claim.name": "preferred_username",
                    "jsonType.label": "String"
                }
            }
        ]
    },
    {
        "id": "3ab2520f-feb4-43fd-9256-fa9f3e521aa7",
        "name": "address",
        "description": "OpenID Connect built-in scope: address",
```

```
            "protocol": "openid-connect",
            "attributes": {
                "include.in.token.scope": "true",
                "display.on.consent.screen": "true",
                "consent.screen.text": "${addressScopeConsentText}"
            },
            "protocolMappers": [
                {
                    "id": "b083371e-07ae-4b01-9e8c-6a54b396359f",
                    "name": "address",
                    "protocol": "openid-connect",
                    "protocolMapper": "oidc-address-mapper",
                    "consentRequired": false,
                    "config": {
                        "user.attribute.formatted": "formatted",
                        "user.attribute.country": "country",
                        "introspection.token.claim": "true",
                        "user.attribute.postal_code": "postal_code",
                        "userinfo.token.claim": "true",
                        "user.attribute.street": "street",
                        "id.token.claim": "true",
                        "user.attribute.region": "region",
                        "access.token.claim": "true",
                        "user.attribute.locality": "locality"
                    }
                }
            ]
        },
        {
            "id": "fda66a99-e8b6-49e6-9186-0a7026ec0275",
            "name": "web-origins",
            "description": "OpenID Connect scope for add allowed web origins
to the access token",
            "protocol": "openid-connect",
            "attributes": {
                "include.in.token.scope": "false",
                "display.on.consent.screen": "false",
                "consent.screen.text": ""
            },
            "protocolMappers": [
                {
                    "id": "4b12d4de-8c05-4251-9c8f-f801cfa3bf2a",
                    "name": "allowed web origins",
                    "protocol": "openid-connect",
                    "protocolMapper": "oidc-allowed-origins-mapper",
                    "consentRequired": false,
                    "config": {
                        "introspection.token.claim": "true",
                        "access.token.claim": "true"
                    }
                }
            ]
        },
        {
            "id": "581928cd-35d7-4c79-a9d1-4e1c9c8ade7e",
```

```
            "name": "acr",
            "description": "OpenID Connect scope for add acr (authentication
context class reference) to the token",
            "protocol": "openid-connect",
            "attributes": {
                "include.in.token.scope": "false",
                "display.on.consent.screen": "false"
            },
            "protocolMappers": [
                {
                    "id": "d664d3db-be9f-4fee-a72a-be6a295c47f5",
                    "name": "acr loa level",
                    "protocol": "openid-connect",
                    "protocolMapper": "oidc-acr-mapper",
                    "consentRequired": false,
                    "config": {
                        "id.token.claim": "true",
                        "introspection.token.claim": "true",
                        "access.token.claim": "true",
                        "userinfo.token.claim": "true"
                    }
                }
            ]
        },
        {
            "id": "9ac2f136-a665-4550-ac77-cc61a1cd1e95",
            "name": "role_list",
            "description": "SAML role list",
            "protocol": "saml",
            "attributes": {
                "consent.screen.text": "${samlRoleListScopeConsentText}",
                "display.on.consent.screen": "true"
            },
            "protocolMappers": [
                {
                    "id": "040520c7-9dfb-4f9d-a93e-17e3267b1517",
                    "name": "role list",
                    "protocol": "saml",
                    "protocolMapper": "saml-role-list-mapper",
                    "consentRequired": false,
                    "config": {
                        "single": "false",
                        "attribute.nameformat": "Basic",
                        "attribute.name": "Role"
                    }
                }
            ]
        },
        {
            "id": "f0ff8363-c507-4113-adc0-47f6b346de26",
            "name": "roles",
            "description": "OpenID Connect scope for add user roles to the
access token",
            "protocol": "openid-connect",
            "attributes": {
```

```
                "include.in.token.scope": "false",
                "display.on.consent.screen": "true",
                "consent.screen.text": "${rolesScopeConsentText}"
            },
            "protocolMappers": [
                {
                    "id": "13aa2234-81bf-4018-a67c-d657045eac1f",
                    "name": "client roles",
                    "protocol": "openid-connect",
                    "protocolMapper": "oidc-usermodel-client-role-mapper",
                    "consentRequired": false,
                    "config": {
                        "introspection.token.claim": "true",
                        "multivalued": "true",
                        "user.attribute": "foo",
                        "access.token.claim": "true",
                        "claim.name": "resource_access.${client_id}.roles",
                        "jsonType.label": "String"
                    }
                },
                {
                    "id": "d0b63e0d-5543-41dc-baf6-1c6987d6a18d",
                    "name": "audience resolve",
                    "protocol": "openid-connect",
                    "protocolMapper": "oidc-audience-resolve-mapper",
                    "consentRequired": false,
                    "config": {
                        "introspection.token.claim": "true",
                        "access.token.claim": "true"
                    }
                },
                {
                    "id": "38881c98-4009-412b-b924-d36f55273f3e",
                    "name": "realm roles",
                    "protocol": "openid-connect",
                    "protocolMapper": "oidc-usermodel-realm-role-mapper",
                    "consentRequired": false,
                    "config": {
                        "introspection.token.claim": "true",
                        "multivalued": "true",
                        "user.attribute": "foo",
                        "access.token.claim": "true",
                        "claim.name": "realm_access.roles",
                        "jsonType.label": "String"
                    }
                }
            ]
        },
        {
            "id": "b8a70a2a-a24d-4862-ad4b-dd737b60f7ce",
            "name": "microprofile-jwt",
            "description": "Microprofile - JWT built-in scope",
            "protocol": "openid-connect",
            "attributes": {
                "include.in.token.scope": "true",
```

```
                "display.on.consent.screen": "false"
            },
            "protocolMappers": [
                {
                    "id": "4eb888ff-b503-4506-b35c-ea0ac0ff3cd3",
                    "name": "groups",
                    "protocol": "openid-connect",
                    "protocolMapper": "oidc-usermodel-realm-role-mapper",
                    "consentRequired": false,
                    "config": {
                        "introspection.token.claim": "true",
                        "multivalued": "true",
                        "userinfo.token.claim": "true",
                        "user.attribute": "foo",
                        "id.token.claim": "true",
                        "access.token.claim": "true",
                        "claim.name": "groups",
                        "jsonType.label": "String"
                    }
                },
                {
                    "id": "d39896cb-3f45-4a62-a254-f7d0eb10e60a",
                    "name": "upn",
                    "protocol": "openid-connect",
                    "protocolMapper": "oidc-usermodel-attribute-mapper",
                    "consentRequired": false,
                    "config": {
                        "introspection.token.claim": "true",
                        "userinfo.token.claim": "true",
                        "user.attribute": "username",
                        "id.token.claim": "true",
                        "access.token.claim": "true",
                        "claim.name": "upn",
                        "jsonType.label": "String"
                    }
                }
            ]
        }
    ],
    "defaultDefaultClientScopes": [ "role_list", "profile", "email",
"roles", "web-origins", "acr" ],
    "defaultOptionalClientScopes": [ "offline_access", "address", "phone",
"microprofile-jwt", "weather:all" ],
    "browserSecurityHeaders": {
        "contentSecurityPolicyReportOnly": "",
        "xContentTypeOptions": "nosniff",
        "referrerPolicy": "no-referrer",
        "xRobotsTag": "none",
        "xFrameOptions": "SAMEORIGIN",
        "contentSecurityPolicy": "frame-src 'self'; frame-ancestors 'self';
object-src 'none';",
        "xXSSProtection": "1; mode=block",
        "strictTransportSecurity": "max-age=31536000; includeSubDomains"
    },
    "smtpServer": {},
```

```
    "eventsEnabled": false,
    "eventsListeners": [ "jboss-logging" ],
    "enabledEventTypes": [],
    "adminEventsEnabled": false,
    "adminEventsDetailsEnabled": false,
    "identityProviders": [],
    "identityProviderMappers": [],
    "components": {

"org.keycloak.services.clientregistration.policy.ClientRegistrationPolicy":
[
            {
                "id": "887848c6-60f6-47ac-ae7f-62f8a5608a4b",
                "name": "Trusted Hosts",
                "providerId": "trusted-hosts",
                "subType": "anonymous",
                "subComponents": {},
                "config": {
                    "host-sending-registration-request-must-match": [ "true"
],
                    "client-uris-must-match": [ "true" ]
                }
            },
            {
                "id": "4333143f-bf59-419a-99e2-2cce8a5d414a",
                "name": "Consent Required",
                "providerId": "consent-required",
                "subType": "anonymous",
                "subComponents": {},
                "config": {}
            },
            {
                "id": "1ac2db3a-57a1-4567-bc2b-80a2b0c96b71",
                "name": "Max Clients Limit",
                "providerId": "max-clients",
                "subType": "anonymous",
                "subComponents": {},
                "config": {
                    "max-clients": [ "200" ]
                }
            },
            {
                "id": "adb4b546-6386-46e0-8ce9-80bacbba2afe",
                "name": "Allowed Protocol Mapper Types",
                "providerId": "allowed-protocol-mappers",
                "subType": "authenticated",
                "subComponents": {},
                "config": {
                    "allowed-protocol-mapper-types": [ "oidc-usermodel-
attribute-mapper", "oidc-address-mapper", "saml-role-list-mapper", "saml-
user-property-mapper", "saml-user-attribute-mapper", "oidc-sha256-pairwise-
sub-mapper", "oidc-full-name-mapper", "oidc-usermodel-property-mapper" ]
                }
            },
            {
```

```
                "id": "241c5dea-68b9-4684-a816-80b08ef86bff",
                "name": "Full Scope Disabled",
                "providerId": "scope",
                "subType": "anonymous",
                "subComponents": {},
                "config": {}
            },
            {
                "id": "6b0e159c-33dc-492b-9d88-421973015466",
                "name": "Allowed Protocol Mapper Types",
                "providerId": "allowed-protocol-mappers",
                "subType": "anonymous",
                "subComponents": {},
                "config": {
                    "allowed-protocol-mapper-types": [ "saml-user-property-
mapper", "oidc-usermodel-attribute-mapper", "saml-user-attribute-mapper",
"oidc-address-mapper", "oidc-sha256-pairwise-sub-mapper", "oidc-usermodel-
property-mapper", "saml-role-list-mapper", "oidc-full-name-mapper" ]
                }
            },
            {
                "id": "730409bc-ce7e-4b64-a870-946aeba9f65b",
                "name": "Allowed Client Scopes",
                "providerId": "allowed-client-templates",
                "subType": "anonymous",
                "subComponents": {},
                "config": {
                    "allow-default-scopes": [ "true" ]
                }
            },
            {
                "id": "0b4b8c74-20b6-4902-b971-9b97001da41e",
                "name": "Allowed Client Scopes",
                "providerId": "allowed-client-templates",
                "subType": "authenticated",
                "subComponents": {},
                "config": {
                    "allow-default-scopes": [ "true" ]
                }
            }
        ],
        "org.keycloak.keys.KeyProvider": [
            {
                "id": "c08db9a9-0d9d-4b56-96d5-7f2b1d4528df",
                "name": "rsa-generated",
                "providerId": "rsa-generated",
                "subComponents": {},
                "config": {
                    "privateKey": [
"MIIEowIBAAKCAQEAyJtAKWr1DdQmh9Nxp2LUGOrc5OA+rdXkV6+kOT21wVsIP/1bg6HekqfMySZ
hIxlALfegc90j0mrqkolb5s7axotTwwABwIvgxW5hHIQ4huntiZUYPUuf4m51dwyLs/GM1gSbzs9
ciBKC5i4S21CQzuGp0QHpyOOn1kQZd0vYSGjpG3ewMYphJEfd60TQP74RcqASNoOaS3lU7+5SCQu
iff1fSZYqYvIFmK3rcNrauTSryx6rh935ODSdYzQN0XA6g1WJK2hbBlBJJzeAj/CXXcBaw7aB1Ao
C7kjJ7XaYmHdC+7zIYhvNKcGtFhrMjoOVnJM9PiRMrk7ous7XAmKc6QIDAQABAoIBAC4qjm5Js1o
qnfBpwJDrPVD7sfjJQ5t5azqjzQkwUrUMGF6zlZ06QhDhpY8QMlAjxkGd6JLpjE4nNVMiYeBA8Be
```

```
7pjvs8zpG5qRBBf/MTP79dGFSitjGX+X6EjXi0P7JIuZ4+otybMLS8cV7ynKm/KBjzhMv28fT3oM
AupSaA40MESKVD8BDR0bNQ6H3h1xrq2+81xViFsle9qcrZjJzCz/rttNi3DUct5IeJwc9Wai937L
6H/BU5eBd9vQHM5raLL0iDSF5CQF2tE/j3hq8kdpRm9XRUm/0WaXOz8Il96GfSEUXLVvKnLdHu7q
YbqKKBYI7B3xz8VYiHgDvBMqiBzUCgYEA9Sd/EijWpXAAaFOJk75BOu0sSjZHjoGbaLE1XWsQyM1
fkFT9kxQ+8/kDiHSJal+utoEIrMWK/KdrrIjT0TXbWKCarV+ebgCt9dkTd6Or2SUku5k0TMSMoVE
I2KxI6ZHvCz2Vxe8Ahbao28u8xOGRQYSs5ynvH0oBTex+RP3sv6UCgYEA0Xs55XdSn1PGazAuZYk
9ZKJhCovdalYdjnJYxuSPfBgHV4CIHdhHvd/hohumMakQsnRDmJzbk/uFlqJecZtcQ/DukGylC/6
dvfp6prUrVhghK+7+Bnry7zfQ9rMWslrlhf9ZTrQ8F9pssCtgAfs+Pfj0zF5DxQqFqF3wz12sJPU
CgYATR5HkubV3uUEu8zLknZe/rJtJEs+501OHfjg2Ko9dW1linmx6vqLcyP6QIqoT5YZ179vgyoB
NslTzcqdF0rh3VdoUPGrXN9J2fSXcyNBg+VzULA5C40oz/Y12jMYHKGTmO2el80/VNDI/Ztxnl12
3C1oVq+SUT1ue5zRe9KFDyQKBgBihnqsmnqZxWVFdNvdlbbyZg0OUMpLAUXVgaKPqWBzFToexa0/
nEHh5DLTc/2uzb20sUo5tUzxRROHzcZt2IyEyATsmKzn/1Fh0TVuwzcmvyKa70U69wjbynzWC1VZ
fbcGVxtCETNSZMFJ+pylUe3sZ/N7S7rEKjbDAawJXB1jJAoGBAOZF4M9C1GLCZqt4RrF1MBI0Q+7
Im45Hvr0FSck0y9xLpl9yolQREWoiCo5+WM54YMeSWu0rzBsLAjCoFpIX2cWcICsdPAnyyjIMPmu
Gt/uUaIWmq/ZRAsqkOW0sqxUMggjytNCtvdCyfQOcKsEHpBKrA6z+8l/3Z1RQFwan+VPs" ],
                    "certificate": [
"MIICpTCCAY0CBgGQcJfMUjANBgkqhkiG9w0BAQsFADAWMRQwEgYDVQQDDAtXZWF0aGVyU2hvcDA
eFw0yNDA3MDEyMzE2NTRaFw0zNDA3MDEyMzE4MzRaMBYxFDASBgNVBAMMC1dlYXRoZXJTaG9wMII
BIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAyJtAKWr1DdQmh9Nxp2LUGOrc5OA+rdXkV6+
kOT21wVsIP/1bg6HekqfMySZhIxlALfegc90j0mrqkolb5s7axotTwwABwIvgxW5hHIQ4huntiZU
YPUuf4m51dwyLs/GM1gSbzs9ciBKC5i4S21CQzuGp0QHpyOOn1kQZd0vYSGjpG3ewMYphJEfd60T
QP74RcqASNoOaS3lU7+5SCQuiff1fSZYqYvIFmK3rcNrauTSryx6rh935ODSdYzQN0XA6g1WJK2h
bBlBJJzeAj/CXXcBaw7aB1AoC7kjJ7XaYmHdC+7zIYhvNKcGtFhrMjoOVnJM9PiRMrk7ous7XAmK
c6QIDAQABMA0GCSqGSIb3DQEBCwUAA4IBAQAeDhADaUkVzu1o8CvCfU9oa6hGsJa+2qFeUPQnG7H
ZAQ07MjQ/RZgMMOmdhspZIuaf+Yrx/My8VDphKzNtb2eNHgFVVQDy1F1jVy9z+t7xCwX3UfjtVNM
p3tMmzWUEi0pckW0mq4Bz3w0+dKPw8z0K2c19dVN1NHDViqFghB+77tO8JguwTDE8fkmXjLixDCc
enBCPXjxNWmhXgOMF1wKhlq1h0+SaKt/F2P/WzyoYu6tz8qVysQvv4knB/HEGjSji+DN+uJFE4RJ
G+B5X+vp5LHvlYKIROQ7/aSzvCjx7zrslmcTsSTL3F1k2Ox2Hz+rGIcA8sGqbj+W5+nKxRbad"
],
                    "priority": [ "100" ]
                }
            },
            {
                "id": "9adc7ed1-f51b-48d4-9d52-96b46abefa18",
                "name": "aes-generated",
                "providerId": "aes-generated",
                "subComponents": {},
                "config": {
                    "kid": [ "eb1853e7-ac8c-4e8f-8b4f-9ea28a71da76" ],
                    "secret": [ "Jges9iPdpfx0aivf3RUUpw" ],
                    "priority": [ "100" ]
                }
            },
            {
                "id": "f4b81876-91be-46f3-9050-c351cd1531af",
                "name": "rsa-enc-generated",
                "providerId": "rsa-enc-generated",
                "subComponents": {},
                "config": {
                    "privateKey": [
"MIIEowIBAAKCAQEAlP+NrT2KVZpdrbPoTsMO7MqeXYDeJNl7IXjY0hCb7p1iB6YLOr/lA5ryk/C
IHI05HRt+AEYFac87mb51SEvAa9cHjQ00v7t6hoYV1esyRmB0Nnf8AAEq9GoZxX9nUsIMcExQ3gH
kF56kidYtjQSgl5SlwgdvlsjRiDP9ZJlsWTBb+8v0OCCbQLZFl93IlTZ7QlaxXoB0dCuuLNyBpEL
Fbc0+JeB+1P/Dw5azUKGdp3ng2K9IrtDBiMh+KicpLZeBpUlqdKqyI2WvwruL1SlqH/ymWCxseRH
0Z9VZ30MfW8C5fHq3qnLQp0OWDa7Re/pRbCZPabivDkZCtuWUeAVbRwIDAQABAoIBAAoioBaKuyA
```

```
7kefA9yp0Zk2BMuiVXYcQLCoIuGcBrjm7BvISP21NpFxsa9fYYsneaWYrepS2LqQV7q30oLG8RWi
Qhfj4TwBD1n/UGyQkDZVv9jfGTaQKcEuT9BDVK8gbXxE8f7u6UTOyHOsrYInZKLtm5yedrd+J5Yb
oUnJHZXFjmCpuyap8zJQdczUpZeLkj4bRqEHYlM1a5vfMFpr2+k4/Nqo36CCaGzIcwtYxnC002rL
7ra9MaR1fy5KHtcoYQeePuPvXu1UxBU01+W1QmIsw+KmXftdtWYAMcVFkHDs+22h3mCxRhQqcxmL
1LUP9FfGzrWiX4eTduw4YOkzO7+ECgYEAx732U2G7Qdp1uXsVGE7ceFcrpl/LOTaL8eEtULBiFRd
rn5LOTAptpoDPB8vYCwIQSNrBWXNqTiDubOBcL7sumBK2i4Omtzk/USuMcJmSiMcHaANa/ZNK9nq
twMACqXpIPEkpofzfgBSaRYY1KxaWLbepyLjqviADV98zOI4qY28CgYEAvvbNd+C0pi5u/EkUxhR
dxwCxICfkDYjArcjPU3ZBjIyAzmJ08wU+3CDm2sgts1TX+D7MPsRLqsaM8vMB+BR1xo3FfP/nqNv
jYkChACtHhvkWGVE/qf3/53NNa4lKeVz8h1iV18dvLICj+V+lvUyQPuER4cA8Kzs7dObl52V6Oak
CgYB5bSkvPW2aNhV1QbbsRRzQZ6XYicnAqUFgNRTYRbIKwmch5hxVq81G+G1jfu+CmamOsLX0DC7
m+iwXsjk4pyFHP7ELlWgnYLz2OnQxC5tCXURKXifVmdJrjt7MG65Cm10IkS2nFVRFx8CVXWY7IIs
BlfK4XHoQROPjaoP38K0iLwKBgQCho6RdkR04ANu+vllQJNMP7B0Be+KENjnpn60mF1X6kr9AcoR
NZCZGC698hq5wOiOoo/ccNelafz+1MU58X00lqMD+QlojSySX+N6OlxOvQs2a1nQN/sqKbcWdfZN
FURkLs0b6Y3xN7gFdxsEyj0kVgEszjBUh/rwgAoWdrP6dKQKBgD2b7GG1/TPTBfOvvE7q7xa5IlT
MUbBU2yY8H6XvDgfG5KUXXFpqINTOE4OjuyVwJRwE2/GQGayfZvfa+LGkW8Hy6Iz/vaa1Brjodza
/2PblM5v1t96xA0WQDime8okJN0q7ocbLfQnT0+3TVMbvCjeBtOqmMOJT3EijpAqVbR6N" ],
                    "certificate": [
"MIICpTCCAY0CBgGQcJfM/jANBgkqhkiG9w0BAQsFADAWMRQwEgYDVQQDDAtXZWF0aGVyU2hvcDA
eFw0yNDA3MDEyMzE2NTVaFw0zNDA3MDEyMzE4MzVaMBYxFDASBgNVBAMMC1dlYXRoZXJTaG9wMII
BIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAlP+NrT2KVZpdrbPoTsMO7MqeXYDeJNl7IXj
Y0hCb7p1iB6YLOr/lA5ryk/CIHI05HRt+AEYFac87mb51SEvAa9cHjQ00v7t6hoYV1esyRmB0Nnf
8AAEq9GoZxX9nUsIMcExQ3gHkF56kidYtjQSgl5SlwgdvlsjRiDP9ZJlsWTBb+8v0OCCbQLZFl93
IlTZ7QlaxXoB0dCuuLNyBpELFbc0+JeB+1P/Dw5azUKGdp3ng2K9IrtDBiMh+KicpLZeBpUlqdKq
yI2WvwruL1SlqH/ymWCxseRH0Z9VZ30MfW8C5fHq3qnLQp0OWDa7Re/pRbCZPabivDkZCtuWUeAV
bRwIDAQABMA0GCSqGSIb3DQEBCwUAA4IBAQAUIS+ce3NPCSk7iiA4vzm1hGrEq7Q+1CwE9hq/p8o
KowOEVvg68tE+yzNsYw6qM+KKdzQfmiVeT8skDhNwL+5+Oxsg9dw7KW1me+g/pjiFx1eXt/rHN5a
VDz7/F3QAP0G/CUF6dVNh0ggoGhwAH74iH91apmJgDUEBVzwaYCrHDJ81nWZOGZm4MF6FFvc8Kwf
/+KEefL7psH5I4BqS+gRaPFWjBnABS7WkJ879gv0Q3tHE4KXF1b3eudGFrW4rG048pqNJgxAXdoD
qFR5qIi9pfuE+HCmuhPv2Xq+I7S4PpUYnUM7o0Ng+1hJsRLhiG0Kmcepy7thiJJI619miVXdF"
],
                    "priority": [ "100" ],
                    "algorithm": [ "RSA-OAEP" ]
                }
            },
            {
                "id": "c962c2d9-ac19-4a91-88b3-959c6fcfc4c4",
                "name": "hmac-generated-hs512",
                "providerId": "hmac-generated",
                "subComponents": {},
                "config": {
                    "kid": [ "f43f7c3d-e27c-4a86-bda1-e1f9fa0b2c0b" ],
                    "secret": [
"FAyCBV9_zIF2oQO0XqgwCJz09iJDMKPHORhWI1ZV4OA9cLFVJCA-
z4tEXq2QNU48xDMwv_z_UYIEm73nnJEypuaVwacu6N7jexaKjhqROYidQyPzXAr7QwD6Du1LaLdC
AHaBo7rRP3Pl5fDmcX6K3C1Qe1OK86fkZVuD_2TX6No" ],
                    "priority": [ "100" ],
                    "algorithm": [ "HS512" ]
                }
            }
        ]
    },
    "internationalizationEnabled": false,
    "supportedLocales": [],
    "authenticationFlows": [
        {
```

```
            "id": "57c6972a-8262-4fdd-9a3d-6454f7e4804d",
            "alias": "Account verification options",
            "description": "Method with which to verity the existing
account",
            "providerId": "basic-flow",
            "topLevel": false,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticator": "idp-email-verification",
                    "authenticatorFlow": false,
                    "requirement": "ALTERNATIVE",
                    "priority": 10,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticatorFlow": true,
                    "requirement": "ALTERNATIVE",
                    "priority": 20,
                    "autheticatorFlow": true,
                    "flowAlias": "Verify Existing Account by Re-
authentication",
                    "userSetupAllowed": false
                }
            ]
        },
        {
            "id": "12219f2a-a63e-4e35-87b0-0c1fc82e9e00",
            "alias": "Browser - Conditional OTP",
            "description": "Flow to determine if the OTP is required for the
authentication",
            "providerId": "basic-flow",
            "topLevel": false,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticator": "conditional-user-configured",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 10,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticator": "auth-otp-form",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 20,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                }
            ]
        },
        {
```

```
            "id": "16a216e0-2305-4a26-8b3c-a1ad95ee0551",
            "alias": "Direct Grant - Conditional OTP",
            "description": "Flow to determine if the OTP is required for the
authentication",
            "providerId": "basic-flow",
            "topLevel": false,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticator": "conditional-user-configured",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 10,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticator": "direct-grant-validate-otp",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 20,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                }
            ]
        },
        {
            "id": "973a2638-3205-40a5-9ae1-171e862f98c2",
            "alias": "First broker login - Conditional OTP",
            "description": "Flow to determine if the OTP is required for the
authentication",
            "providerId": "basic-flow",
            "topLevel": false,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticator": "conditional-user-configured",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 10,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticator": "auth-otp-form",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 20,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                }
            ]
        },
        {
            "id": "d024eba2-74d6-4cd2-a149-eda663682a7b",
```

```
            "alias": "Handle Existing Account",
            "description": "Handle what to do if there is existing account
with same email/username like authenticated identity provider",
            "providerId": "basic-flow",
            "topLevel": false,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticator": "idp-confirm-link",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 10,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticatorFlow": true,
                    "requirement": "REQUIRED",
                    "priority": 20,
                    "autheticatorFlow": true,
                    "flowAlias": "Account verification options",
                    "userSetupAllowed": false
                }
            ]
        },
        {
            "id": "37b5496c-9e09-402f-b079-9c588322f91d",
            "alias": "Reset - Conditional OTP",
            "description": "Flow to determine if the OTP should be reset or
not. Set to REQUIRED to force.",
            "providerId": "basic-flow",
            "topLevel": false,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticator": "conditional-user-configured",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 10,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticator": "reset-otp",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 20,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                }
            ]
        },
        {
            "id": "62d51e78-668a-4d31-9369-0611a7507ed5"
```

```
            "description": "Flow for the existing/non-existing user
alternatives",
            "providerId": "basic-flow",
            "topLevel": false,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticatorConfig": "create unique user config",
                    "authenticator": "idp-create-user-if-unique",
                    "authenticatorFlow": false,
                    "requirement": "ALTERNATIVE",
                    "priority": 10,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticatorFlow": true,
                    "requirement": "ALTERNATIVE",
                    "priority": 20,
                    "autheticatorFlow": true,
                    "flowAlias": "Handle Existing Account",
                    "userSetupAllowed": false
                }
            ]
        },
        {
            "id": "ab6b9698-fd26-40a7-8edf-fa456a395394",
            "alias": "Verify Existing Account by Re-authentication",
            "description": "Reauthentication of existing account",
            "providerId": "basic-flow",
            "topLevel": false,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticator": "idp-username-password-form",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 10,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticatorFlow": true,
                    "requirement": "CONDITIONAL",
                    "priority": 20,
                    "autheticatorFlow": true,
                    "flowAlias": "First broker login - Conditional OTP",
                    "userSetupAllowed": false
                }
            ]
        },
        {
            "id": "74b79e04-bf44-4d84-92e7-04b753801622",
            "alias": "browser",
            "description": "browser based authentication",
```

```
            "providerId": "basic-flow",
            "topLevel": true,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticator": "auth-cookie",
                    "authenticatorFlow": false,
                    "requirement": "ALTERNATIVE",
                    "priority": 10,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticator": "auth-spnego",
                    "authenticatorFlow": false,
                    "requirement": "DISABLED",
                    "priority": 20,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticator": "identity-provider-redirector",
                    "authenticatorFlow": false,
                    "requirement": "ALTERNATIVE",
                    "priority": 25,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticatorFlow": true,
                    "requirement": "ALTERNATIVE",
                    "priority": 30,
                    "autheticatorFlow": true,
                    "flowAlias": "forms",
                    "userSetupAllowed": false
                }
            ]
        },
        {
            "id": "bb15232e-9f1c-4dfd-8d10-e1d35cd1bfde",
            "alias": "clients",
            "description": "Base authentication for clients",
            "providerId": "client-flow",
            "topLevel": true,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticator": "client-secret",
                    "authenticatorFlow": false,
                    "requirement": "ALTERNATIVE",
                    "priority": 10,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
```

```
                "authenticator": "client-jwt",
                "authenticatorFlow": false,
                "requirement": "ALTERNATIVE",
                "priority": 20,
                "autheticatorFlow": false,
                "userSetupAllowed": false
            },
            {
                "authenticator": "client-secret-jwt",
                "authenticatorFlow": false,
                "requirement": "ALTERNATIVE",
                "priority": 30,
                "autheticatorFlow": false,
                "userSetupAllowed": false
            },
            {
                "authenticator": "client-x509",
                "authenticatorFlow": false,
                "requirement": "ALTERNATIVE",
                "priority": 40,
                "autheticatorFlow": false,
                "userSetupAllowed": false
            }
        ]
    },
    {
        "id": "dd3fe894-1594-40fb-949d-9986f36bd725",
        "alias": "direct grant",
        "description": "OpenID Connect Resource Owner Grant",
        "providerId": "basic-flow",
        "topLevel": true,
        "builtIn": true,
        "authenticationExecutions": [
            {
                "authenticator": "direct-grant-validate-username",
                "authenticatorFlow": false,
                "requirement": "REQUIRED",
                "priority": 10,
                "autheticatorFlow": false,
                "userSetupAllowed": false
            },
            {
                "authenticator": "direct-grant-validate-password",
                "authenticatorFlow": false,
                "requirement": "REQUIRED",
                "priority": 20,
                "autheticatorFlow": false,
                "userSetupAllowed": false
            },
            {
                "authenticatorFlow": true,
                "requirement": "CONDITIONAL",
                "priority": 30,
                "autheticatorFlow": true,
                "flowAlias": "Direct Grant - Conditional OTP",
```

```
                    "userSetupAllowed": false
                }
            ]
        },
        {
            "id": "5c277901-3d67-470e-baf1-e47e9ab92dbd",
            "alias": "docker auth",
            "description": "Used by Docker clients to authenticate against
the IDP",
            "providerId": "basic-flow",
            "topLevel": true,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticator": "docker-http-basic-authenticator",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 10,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                }
            ]
        },
        {
            "id": "e14e1d45-149f-4090-ad79-7c2d0c2f185c",
            "alias": "first broker login",
            "description": "Actions taken after first broker login with
identity provider account, which is not yet linked to any Keycloak account",
            "providerId": "basic-flow",
            "topLevel": true,
            "builtIn": true
```

```
            "providerId": "basic-flow",
            "topLevel": false,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticator": "auth-username-password-form",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 10,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticatorFlow": true,
                    "requirement": "CONDITIONAL",
                    "priority": 20,
                    "autheticatorFlow": true,
                    "flowAlias": "Browser - Conditional OTP",
                    "userSetupAllowed": false
                }
            ]
        },
        {
            "id": "d88b5042-8af8-4797-9c6a-ae44a9a0fff2",
            "alias": "registration",
            "description": "registration flow",
            "providerId": "basic-flow",
            "topLevel": true,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticator": "registration-page-form",
                    "authenticatorFlow": true,
                    "requirement": "REQUIRED",
                    "priority": 10,
                    "autheticatorFlow": true,
                    "flowAlias": "registration form",
                    "userSetupAllowed": false
                }
            ]
        },
        {
            "id": "2a9320cb-970e-4b4d-b585-60d2299a043f",
            "alias": "registration form",
            "description": "registration form",
            "providerId": "form-flow",
            "topLevel": false,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticator": "registration-user-creation",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 20,
                    "autheticatorFlow": false,
```

```
                    "userSetupAllowed": false
                },
                {
                    "authenticator": "registration-password-action",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 50,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticator": "registration-recaptcha-action",
                    "authenticatorFlow": false,
                    "requirement": "DISABLED",
                    "priority": 60,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticator": "registration-terms-and-conditions",
                    "authenticatorFlow": false,
                    "requirement": "DISABLED",
                    "priority": 70,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                }
            ]
        },
        {
            "id": "528ba840-6b22-4c32-ba17-40c99783883e",
            "alias": "reset credentials",
            "description": "Reset credentials for a user if they forgot
their password or something",
            "providerId": "basic-flow",
            "topLevel": true,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticator": "reset-credentials-choose-user",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 10,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticator": "reset-credential-email",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 20,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticator": "reset-password",
```

```
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 30,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                },
                {
                    "authenticatorFlow": true,
                    "requirement": "CONDITIONAL",
                    "priority": 40,
                    "autheticatorFlow": true,
                    "flowAlias": "Reset - Conditional OTP",
                    "userSetupAllowed": false
                }
            ]
        },
        {
            "id": "bf172b2d-052a-4ce4-9084-c59e9b82bc10",
            "alias": "saml ecp",
            "description": "SAML ECP Profile Authentication Flow",
            "providerId": "basic-flow",
            "topLevel": true,
            "builtIn": true,
            "authenticationExecutions": [
                {
                    "authenticator": "http-basic-authenticator",
                    "authenticatorFlow": false,
                    "requirement": "REQUIRED",
                    "priority": 10,
                    "autheticatorFlow": false,
                    "userSetupAllowed": false
                }
            ]
        }
    ],
    "authenticatorConfig": [
        {
            "id": "796c70f7-6391-45ed-aafa-4ed82c84d14e",
            "alias": "create unique user config",
            "config": {
                "require.password.update.after.registration": "false"
            }
        },
        {
            "id": "e007b422-2050-43af-b132-10ea16d92f5c",
            "alias": "review profile config",
            "config": {
                "update.profile.on.first.login": "missing"
            }
        }
    ],
    "requiredActions": [
        {
            "alias": "CONFIGURE_TOTP",
            "name": "Configure OTP",
```

```
            "providerId": "CONFIGURE_TOTP",
            "enabled": true,
            "defaultAction": false,
            "priority": 10,
            "config": {}
        },
        {
            "alias": "TERMS_AND_CONDITIONS",
            "name": "Terms and Conditions",
            "providerId": "TERMS_AND_CONDITIONS",
            "enabled": false,
            "defaultAction": false,
            "priority": 20,
            "config": {}
        },
        {
            "alias": "UPDATE_PASSWORD",
            "name": "Update Password",
            "providerId": "UPDATE_PASSWORD",
            "enabled": true,
            "defaultAction": false,
            "priority": 30,
            "config": {}
        },
        {
            "alias": "UPDATE_PROFILE",
            "name": "Update Profile",
            "providerId": "UPDATE_PROFILE",
            "enabled": true,
            "defaultAction": false,
            "priority": 40,
            "config": {}
        },
        {
            "alias": "VERIFY_EMAIL",
            "name": "Verify Email",
            "providerId": "VERIFY_EMAIL",
            "enabled": true,
            "defaultAction": false,
            "priority": 50,
            "config": {}
        },
        {
            "alias": "delete_account",
            "name": "Delete Account",
            "providerId": "delete_account",
            "enabled": false,
            "defaultAction": false,
            "priority": 60,
            "config": {}
        },
        {
            "alias": "webauthn-register",
            "name": "Webauthn Register",
            "providerId": "webauthn-register",
```

```
            "enabled": true,
            "defaultAction": false,
            "priority": 70,
            "config": {}
        },
        {
            "alias": "webauthn-register-passwordless",
            "name": "Webauthn Register Passwordless",
            "providerId": "webauthn-register-passwordless",
            "enabled": true,
            "defaultAction": false,
            "priority": 80,
            "config": {}
        },
        {
            "alias": "VERIFY_PROFILE",
            "name": "Verify Profile",
            "providerId": "VERIFY_PROFILE",
            "enabled": true,
            "defaultAction": false,
            "priority": 90,
            "config": {}
        },
        {
            "alias": "delete_credential",
            "name": "Delete Credential",
            "providerId": "delete_credential",
            "enabled": true,
            "defaultAction": false,
            "priority": 100,
            "config": {}
        },
        {
            "alias": "update_user_locale",
            "name": "Update User Locale",
            "providerId": "update_user_locale",
            "enabled": true,
            "defaultAction": false,
            "priority": 1000,
            "config": {}
        }
    ],
    "browserFlow": "browser",
    "registrationFlow": "registration",
    "directGrantFlow": "direct grant",
    "resetCredentialsFlow": "reset credentials",
    "clientAuthenticationFlow": "clients",
    "dockerAuthenticationFlow": "docker auth",
    "firstBrokerLoginFlow": "first broker login",
    "attributes": {
        "cibaBackchannelTokenDeliveryMode": "poll",
        "cibaExpiresIn": "120",
        "cibaAuthRequestedUserHint": "login_hint",
        "oauth2DeviceCodeLifespan": "600",
        "clientOfflineSessionMaxLifespan": "0",
```

```
            "oauth2DevicePollingInterval": "5",
            "clientSessionIdleTimeout": "0",
            "parRequestUriLifespan": "60",
            "clientSessionMaxLifespan": "0",
            "clientOfflineSessionIdleTimeout": "0",
            "cibaInterval": "5",
            "realmReusableOtpCode": "false"
        },
        "keycloakVersion": "24.0.5",
        "userManagedAccessAllowed": false,
        "clientProfiles": {
            "profiles": []
        },
        "clientPolicies": {
            "policies": []
        }
    }
```

Hosting integration health checks

The Keycloak hosting integration doesn't currently support a health checks, nor does it automatically add them.

Client integration

To get started with the .NET Aspire Keycloak client integration, install the 📦 Aspire.Keycloak.Authentication NuGet package in the client-consuming project, that is, the project for the application that uses the Keycloak client. The Keycloak client integration registers JwtBearer and OpenId Connect authentication handlers in the DI container for connecting to a Keycloak.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Keycloak.Authentication --prerelease
```

Add JWT bearer authentication

In the *Program.cs* file of your ASP.NET Core API project, call the AddKeycloakJwtBearer extension method to add JwtBearer authentication, using a connection name, realm and any required JWT Bearer options:

C#

```
builder.Services.AddAuthentication()
                .AddKeycloakJwtBearer(
                    serviceName: "keycloak",
                    realm: "api",
                    options =>
                    {
                        options.Audience = "store.api";
                    });
```

You can set many other options via the `Action<JwtBearerOptions> configureOptions` delegate.

JWT bearer authentication example

To further exemplify the JWT bearer authentication, consider the following example:

C#

```
var builder = WebApplication.CreateBuilder(args);

// Add service defaults & Aspire client integrations.
builder.AddServiceDefaults();

// Add services to the container.
builder.Services.AddProblemDetails();

// Learn more about configuring OpenAPI at https://aka.ms/aspnet/openapi
builder.Services.AddOpenApi();

builder.Services.AddAuthentication()
                .AddKeycloakJwtBearer(
                    serviceName: "keycloak",
                    realm: "WeatherShop",
                    configureOptions: options =>
                {
                    options.RequireHttpsMetadata = false;
                    options.Audience = "weather.api";
                });

builder.Services.AddAuthorizationBuilder();

var app = builder.Build();

// Configure the HTTP request pipeline.
app.UseExceptionHandler();

if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
```

```
}

string[] summaries = ["Freezing", "Bracing", "Chilly", "Cool", "Mild",
"Warm", "Balmy", "Hot", "Sweltering", "Scorching"];

app.MapGet("/weatherforecast", () =>
    {
        var forecast = Enumerable.Range(1, 5).Select(index =>
            new WeatherForecast
            (
                DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
                Random.Shared.Next(-20, 55),
                summaries[Random.Shared.Next(summaries.Length)]
            ))
            .ToArray();
        return forecast;
    })
    .WithName("GetWeatherForecast")
    .RequireAuthorization();

app.MapDefaultEndpoints();

app.Run();

record WeatherForecast(DateOnly Date, int TemperatureC, string? Summary)
{
    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
}
```

The preceding ASP.NET Core Minimal API `Program` class demonstrates:

- Adding authentication services to the DI container with the AddAuthentication API.
- Adding JWT bearer authentication with the AddKeycloakJwtBearer API and configuring:
  - The `serviceName` as `keycloak`.
  - The `realm` as `WeatherShop`.
  - The `options` with the `Audience` set to `weather.api` and sets `RequireHttpsMetadata` to `false`.
- Adds authorization services to the DI container with the AddAuthorizationBuilder API.
- Calls the RequireAuthorization API to require authorization on the `/weatherforecast` endpoint.

For a complete working sample, see .NET Aspire playground: Keycloak integration.

Add OpenId Connect authentication

In the *Program.cs* file of your API-consuming project (for example, Blazor), call the AddKeycloakOpenIdConnect extension method to add OpenId Connect authentication, using a connection name, realm and any required OpenId Connect options:

C#

```
builder.Services.AddAuthentication(OpenIdConnectDefaults.AuthenticationSchem
e)
                .AddKeycloakOpenIdConnect(
                    serviceName: "keycloak",
                    realm: "api",
                    options =>
                    {
                        options.ClientId = "StoreWeb";
                        options.ResponseType =
OpenIdConnectResponseType.Code;
                        options.Scope.Add("store:all");
                    });
```

You can set many other options via the `Action<OpenIdConnectOptions>? configureOptions` delegate.

OpenId Connect authentication example

To further exemplify the OpenId Connect authentication, consider the following example:

C#

```
using System.IdentityModel.Tokens.Jwt;

using AspireApp.Web;
using AspireApp.Web.Components;

using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.OpenIdConnect;
using Microsoft.IdentityModel.Protocols.OpenIdConnect;

var builder = WebApplication.CreateBuilder(args);

// Add service defaults & Aspire client integrations.
builder.AddServiceDefaults();

// Add services to the container.
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();

builder.Services.AddOutputCache();
```

```
builder.Services.AddHttpContextAccessor()
                .AddTransient<AuthorizationHandler>();

builder.Services.AddHttpClient<WeatherApiClient>(client =>
    {
        // This URL uses "https+http://" to indicate HTTPS is preferred over
HTTP.
        // Learn more about service discovery scheme resolution at
https://aka.ms/dotnet/sdschemes.
        client.BaseAddress = new("https+http://apiservice");
    })
    .AddHttpMessageHandler<AuthorizationHandler>();

var oidcScheme = OpenIdConnectDefaults.AuthenticationScheme;

builder.Services.AddAuthentication(oidcScheme)
                .AddKeycloakOpenIdConnect("keycloak", realm: "WeatherShop",
oidcScheme, options =>
                {
                    options.ClientId = "WeatherWeb";
                    options.ResponseType = OpenIdConnectResponseType.Code;
                    options.Scope.Add("weather:all");
                    options.RequireHttpsMetadata = false;
                    options.TokenValidationParameters.NameClaimType =
JwtRegisteredClaimNames.Name;
                    options.SaveTokens = true;
                    options.SignInScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
                })

.AddCookie(CookieAuthenticationDefaults.AuthenticationScheme);

builder.Services.AddCascadingAuthenticationState();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseAntiforgery();

app.UseOutputCache();

app.MapStaticAssets();

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();
```

```
app.MapDefaultEndpoints();
app.MapLoginAndLogout();

app.Run();
```

The preceding ASP.NET Core Blazor `Program` class:

- Adds the `HttpContextAccessor` to the DI container with the AddHttpContextAccessor API.
- Adds a custom `AuthorizationHandler` as a transient service to the DI container with the AddTransient<TService>(IServiceCollection) API.
- Adds an HttpClient to the `WeatherApiClient` service with the AddHttpClient<TClient>(IServiceCollection) API and configuring it's base address with service discovery semantics that resolves to the `apiservice`.
  - Chains a call to the AddHttpMessageHandler API to add a `AuthorizationHandler` to the `HttpClient` pipeline.
- Adds authentication services to the DI container with the AddAuthentication API passing int the OpenId Connect default authentication scheme.
- Calls AddKeycloakOpenIdConnect and configures the `serviceName` as `keycloak`, the `realm` as `WeatherShop`, and the `options` object with various settings.
- Adds cascading authentication state to the Blazor app with the AddCascadingAuthenticationState API.

The final callout is the `MapLoginAndLogout` extension method that adds login and logout routes to the Blazor app. This is defined as follows:

C#

```
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authentication.OpenIdConnect;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Http.HttpResults;

namespace AspireApp.Web;

internal static class LoginLogoutEndpointRouteBuilderExtensions
{
    internal static IEndpointConventionBuilder MapLoginAndLogout(
        this IEndpointRouteBuilder endpoints)
    {
        var group = endpoints.MapGroup("authentication");

        group.MapGet(pattern: "/login", OnLogin).AllowAnonymous();
        group.MapPost(pattern: "/logout", OnLogout);

        return group;
    }
```

```
    static ChallengeHttpResult OnLogin() =>
        TypedResults.Challenge(properties: new AuthenticationProperties
        {
            RedirectUri = "/"
        });

    static SignOutHttpResult OnLogout() =>
        TypedResults.SignOut(properties: new AuthenticationProperties
        {
            RedirectUri = "/"
        },
        [
            CookieAuthenticationDefaults.AuthenticationScheme,
            OpenIdConnectDefaults.AuthenticationScheme
        ]);
}
```

The preceding code:

- Maps a group for the `authentication` route and maps two endpoints for the `login` and `logout` routes:
  - Maps a `GET` request to the `/login` route that's handler is the `OnLogin` method—this is an anonymous endpoint.
  - Maps a `GET` request to the `/logout` route that's handler is the `OnLogout` method.

The `AuthorizationHandler` is a custom handler that adds the `Bearer` token to the `HttpClient` request. The handler is defined as follows:

C#

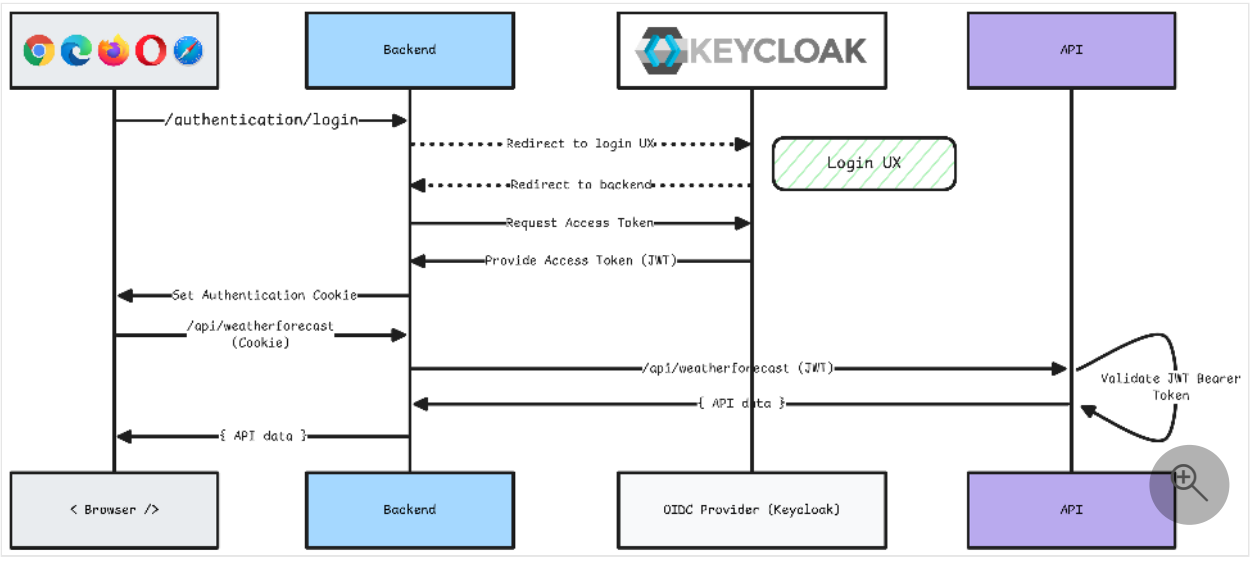
```
using Microsoft.AspNetCore.Authentication;
using System.Net.Http.Headers;

namespace AspireApp.Web;

public class AuthorizationHandler(IHttpContextAccessor httpContextAccessor)
    : DelegatingHandler
{
    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        var httpContext = httpContextAccessor.HttpContext ??
            throw new InvalidOperationException("""
                No HttpContext available from the IHttpContextAccessor.
                """);

        var accessToken = await httpContext.GetTokenAsync("access_token");
```

```
        if (!string.IsNullOrWhiteSpace(accessToken))
        {
            request.Headers.Authorization = new
AuthenticationHeaderValue("Bearer", accessToken);
        }

        return await base.SendAsync(request, cancellationToken);
    }
}
```

The preceding code:

- Is a subclass of the DelegatingHandler class.
- Injects the `IHttpContextAccessor` service in the primary constructor.
- Overrides the `SendAsync` method to add the `Bearer` token to the `HttpClient` request:
  - The `access_token` is retrieved from the `HttpContext` and added to the `Authorization` header.

To help visualize the auth flow, consider the following sequence diagram:



For a complete working sample, see .NET Aspire playground: Keycloak integration.

See also

- Keycloak
- .NET Aspire playground: Keycloak integration
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire Milvus database integration

Article • 02/14/2025

Includes: ✅ Hosting integration and ✅ Client integration

Milvus is an open-source vector database system that efficiently stores, indexes, and searches large-scale vector data. It's commonly used in machine learning, artificial intelligence, and data science applications.

Vector data encodes information as mathematical vectors, which are arrays of numbers or coordinates. Machine learning and AI systems often use vectors to represent unstructured objects like images, text, audio, or video. Each dimension in the vector describes a specific characteristic of the object. By comparing them, systems can classify, search, and identify clusters of objects.

In this article, you learn how to use the .NET Aspire Milvus database integration. The .NET Aspire Milvus database integration enables you to connect to existing Milvus databases or create new instances with the milvusdb/milvus container image.

Hosting integration

The Milvus database hosting integration models the server as the MilvusServerResource type and the database as the MilvusDatabaseResource type. To access these types and APIs, add the Aspire.Hosting.Milvus NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Milvus
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Milvus server and database resources

In your app host project, call AddMilvus to add and return a Milvus resource builder. Chain a call to the returned resource builder to AddDatabase, to add a Milvus database resource.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var milvus = builder.AddMilvus("milvus")
                    .WithLifetime(ContainerLifetime.Persistent);

var milvusdb = milvus.AddDatabase("milvusdb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(milvusdb)
       .WaitFor(milvusdb);

// After adding all resources, run the app...
```

> **Note**
>
> The Milvus container can be slow to start, so it's best to use a *persistent* lifetime to avoid unnecessary restarts. For more information, see Container resource lifetime.

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `milvusdb/milvus` image, it creates a new Milvus instance on your local machine. A reference to your Milvus resource builder (the `milvus` variable) is used to add a database. The database is named `milvusdb` and then added to the `ExampleProject`.

The WithReference method configures a connection in the `ExampleProject` named `milvusdb`.

> **Tip**
>
> If you'd rather connect to an existing Milvus server, call AddConnectionString instead. For more information, see Reference existing resources.

Handling credentials and passing other parameters for the Milvus resource

The Milvus resource includes default credentials with a `username` of `root` and the password `Milvus`. Milvus supports configuration-based default passwords by using the environment variable `COMMON_SECURITY_DEFAULTROOTPASSWORD`. To change the default password in the container, pass an `apiKey` parameter when calling the `AddMilvus` hosting API:

C#

```
var apiKey = builder.AddParameter("apiKey", secret: true);

var milvus = builder.AddMilvus("milvus", apiKey);

var myService = builder.AddProject<Projects.ExampleProject>()
                       .WithReference(milvus);
```

The preceding code gets a parameter to pass to the `AddMilvus` API, and internally assigns the parameter to the `COMMON_SECURITY_DEFAULTROOTPASSWORD` environment variable of the Milvus container. The `apiKey` parameter is usually specified as a *user secret*:

JSON

```
{
  "Parameters": {
    "apiKey": "Non-default-P@ssw0rd"
  }
}
```

For more information, see External parameters.

Add a Milvus resource with a data volume

To add a data volume to the Milvus service resource, call the WithDataVolume method on the Milvus resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var milvus = builder.AddMilvus("milvus")
                    .WithDataVolume();

var milvusdb = milvus.AddDatabase("milvusdb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(milvusdb)
       .WaitFor(milvusdb);

// After adding all resources, run the app...
```

The data volume is used to persist the Milvus data outside the lifecycle of its container. The data volume is mounted at the `/var/lib/milvus` path in the SQL Server container

and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add a Milvus resource with a data bind mount

To add a data bind mount to the Milvus resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var milvus = builder.AddMilvus("milvus")
                    .WithDataBindMount(source: @"C:\Milvus\Data");

var milvusdb = milvus.AddDatabase("milvusdb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(milvusdb)
       .WaitFor(milvusdb);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Milvus data across container restarts. The data bind mount is mounted at the `C:\Milvus\Data` on Windows (or `/Milvus/Data` on Unix) path on the host machine in the Milvus container. For more information on data bind mounts, see Docker docs: Bind mounts.

Create an Attu resource

Attu is a graphical user interface (GUI) and management tool designed to interact with Milvus and its databases. It includes rich visualization features that can help you investigate and understand your vector data.

If you want to use Attu to manage Milvus in your .NET Aspire solution, call the WithAttu extension method on your Milvus resource. The method creates a container from the zilliz/attu image:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var milvus = builder.AddMilvus("milvus")
                    .WithAttu()
                    .WithLifetime(ContainerLifetime.Persistent);

var milvusdb = milvus.AddDatabase("milvusdb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(milvusdb)
       .WaitFor(milvusdb);

// After adding all resources, run the app...
```

When you debug the .NET Aspire solution, you'll see an Attu container listed in the solution's resources. Select the resource's endpoint to open the GUI and start managing databases.

Client integration (Preview)

To get started with the .NET Aspire Milvus client integration, install the Aspire.Milvus.Client NuGet package in the client-consuming project, that is, the project for the application that uses the Milvus database client. The Milvus client integration registers a Milvus.Client.MilvusClient instance that you can use to interact with Milvus databases.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Milvus.Client
```

Add a Milvus client

In the *Program.cs* file of your client-consuming project, call the AddMilvusClient extension method on any IHostApplicationBuilder to register a `MilvusClient` for use

through the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddMilvusClient("milvusdb");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Milvus database resource in the app host project. In other words, when you call `AddDatabase` and provide a name of `milvusdb` that same name should be used when calling `AddMilvusClient`. For more information, see **Add a Milvus server resource and database resource**.

You can then retrieve the `MilvusClient` instance using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(MilvusClient client)
{
    // Use the Milvus Client...
}
```

For more information on dependency injection, see .NET dependency injection.

Add a keyed Milvus client

There might be situations where you want to register multiple `MilvusClient` instances with different connection names. To register keyed Milvus clients, call the AddKeyedMilvusClient method:

C#

```
builder.AddKeyedMilvusClient(name: "mainDb");
builder.AddKeyedMilvusClient(name: "loggingDb");
```

> **Important**

When using keyed services, it's expected that your Milvus resource configured two named databases, one for the `mainDb` and one for the `loggingDb`.

Then you can retrieve the `MilvusClient` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("mainDb")] MilvusClient mainDbClient,
    [FromKeyedServices("loggingDb")] MilvusClient loggingDbClient)
{
    // Use clients...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire Milvus client integration provides multiple options to configure the connection to Milvus based on the requirements and conventions of your project.

> **Tip**
>
> The default use is `root` and the default password is `Milvus`. To configure a different password in the Milvus container, see Handling credentials and passing other parameters for the Milvus resource. Use the following techniques to configure consuming client apps in your .NET Aspire solution with the same password or other settings.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddMilvusClient()`:

C#

```
builder.AddMilvusClient("milvus");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "milvus": "Endpoint=http://localhost:19530/;Key=root:Non-default-
P@ssw0rd"
  }
}
```

By default the `MilvusClient` uses the gRPC API endpoint.

Use configuration providers

The .NET Aspire Milvus client integration supports Microsoft.Extensions.Configuration. It loads the MilvusClientSettings from configuration by using the `Aspire:Milvus:Client` key. The following snippet is an example of a *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "Milvus": {
      "Client": {
        "Endpoint": "http://localhost:19530/",
        "Database": "milvusdb",
        "Key": "root:Non-default-P@ssw0rd",
        "DisableHealthChecks": false
      }
    }
  }
}
```

For the complete Milvus client integration JSON schema, see Aspire.Milvus.Client/ConfigurationSchema.json.

Use inline delegates

Also you can pass the `Action<MilvusSettings> configureSettings` delegate to set up some or all the options inline, for example to set the API key from code:

C#

```
builder.AddMilvusClient(
    "milvus",
```

```
    static settings => settings.Key = "root:Non-default-P@ssw0rd");
```

Client integration health checks

By default, .NET Aspire integrations enable health checks for all services. For more information, see .NET Aspire integrations overview.

The .NET Aspire Milvus database integration:

- Adds the health check when MilvusClientSettings.DisableHealthChecks is `false`, which attempts to connect to the Milvus server.
- Uses the configured client to perform a `HealthAsync`. If the result *is healthy*, the health check is considered healthy, otherwise it's unhealthy. Likewise, if there's an exception, the health check is considered unhealthy with the error propagating through the health check failure.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Milvus database integration uses standard .NET logging, and you'll see log entries from the following category:

- `Milvus.Client`

Tracing

The .NET Aspire Milvus database integration doesn't currently emit tracing activities because they are not supported by the `Milvus.Client` library.

Metrics

The .NET Aspire Milvus database integration doesn't currently emit metrics because they are not supported by the `Milvus.Client` library.

See also

- Milvus
- Milvus GitHub repo
- Milvus .NET SDK
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire MongoDB database integration

Article • 02/25/2025

Includes: ✅ Hosting integration and ✅ Client integration

MongoDB is a NoSQL database that provides high performance, high availability, and easy scalability. The .NET Aspire MongoDB integration enables you to connect to existing MongoDB instances (including MongoDB Atlas) or create new instances from .NET with the docker.io/library/mongo container image

Hosting integration

The MongoDB server hosting integration models the server as the MongoDBServerResource type and the database as the MongoDBDatabaseResource type. To access these types and APIs, add the Aspire.Hosting.MongoDB NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.MongoDB
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add MongoDB server resource and database resource

In your app host project, call AddMongoDB to add and return a MongoDB server resource builder. Chain a call to the returned resource builder to AddDatabase, to add a MongoDB database resource.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var mongo = builder.AddMongoDB("mongo")
                   .WithLifetime(ContainerLifetime.Persistent);
```

```
var mongodb = mongo.AddDatabase("mongodb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(mongodb)
       .WaitFor(mongodb);

// After adding all resources, run the app...
```

> **Note**
>
> The MongoDB container can be slow to start, so it's best to use a *persistent* lifetime to avoid unnecessary restarts. For more information, see Container resource lifetime.

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/library/mongo` image, it creates a new MongoDB instance on your local machine. A reference to your MongoDB server resource builder (the `mongo` variable) is used to add a database. The database is named `mongodb` and then added to the `ExampleProject`. The MongoDB server resource includes default credentials:

- `MONGO_INITDB_ROOT_USERNAME`: A value of `admin`.
- `MONGO_INITDB_ROOT_PASSWORD`: Random `password` generated using the CreateDefaultPasswordParameter method.

When the app host runs, the password is stored in the app host's secret store. It's added to the `Parameters` section, for example:

JSON

```
{
  "Parameters:mongo-password": "<THE_GENERATED_PASSWORD>"
}
```

The name of the parameter is `mongo-password`, but really it's just formatting the resource name with a `-password` suffix. For more information, see Safe storage of app secrets in development in ASP.NET Core and Add MongoDB server resource with parameters.

The WithReference method configures a connection in the `ExampleProject` named `mongodb` and the WaitFor instructs the app host to not start the dependant service until the `mongodb` resource is ready.

> **Tip**

If you'd rather connect to an existing MongoDB server, call **AddConnectionString** instead. For more information, see

To add a data volume to the MongoDB server resource, call the WithDataVolume method on the MongoDB server resource:

C#

The data volume is used to persist the MongoDB server data outside the lifecycle of its container. The data volume is mounted at the `/data/db` path in the MongoDB server container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes .

To add a data bind mount to the MongoDB server resource, call the WithDataBindMount method:

C#

```
var mongo = builder.AddMongoDB("mongo")
                   .WithDataBindMount(@"C:\MongoDB\Data");

var mongodb = mongo.AddDatabase("mongodb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(mongodb)
       .WaitFor(mongodb);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the MongoDB server data across container restarts. The data bind mount is mounted at the `C:\MongoDB\Data` on Windows (or `/MongoDB/Data` on Unix) path on the host machine in the MongoDB server container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add MongoDB server resource with initialization data bind mount

To add an initialization folder data bind mount to the MongoDB server resource, call the WithInitBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var mongo = builder.AddMongoDB("mongo")
                   .WithInitBindMount(@"C:\MongoDB\Init");

var mongodb = mongo.AddDatabase("mongodb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(mongodb)
       .WaitFor(mongodb);

// After adding all resources, run the app...
```

The initialization data bind mount is used to initialize the MongoDB server with data. The initialization data bind mount is mounted at the `C:\MongoDB\Init` on Windows (or `/MongoDB/Init` on Unix) path on the host machine in the MongoDB server container and maps to the `/docker-entrypoint-initdb.d` path in the MongoDB server container. MongoDB executes the scripts found in this folder, which is useful for loading data into the database.

Add MongoDB server resource with parameters

When you want to explicitly provide the password used by the container image, you can provide these credentials as parameters. Consider the following alternative example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var username = builder.AddParameter("username");
var password = builder.AddParameter("password", secret: true);

var mongo = builder.AddMongoDB("mongo", username, password);
var mongodb = mongo.AddDatabase("mongodb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(mongodb)
       .WaitFor(mongodb);

// After adding all resources, run the app...
```

For more information on providing parameters, see External parameters.

Add MongoDB Express resource

MongoDB Express is a web-based MongoDB admin user interface. To add a MongoDB Express resource that corresponds to the docker.io/library/mongo-express container image, call the WithMongoExpress method on the MongoDB server resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var mongo = builder.AddMongoDB("mongo")
                   .WithMongoExpress();

var mongodb = mongo.AddDatabase("mongodb");

builder.AddProject<Projects.ExampleProject>()
```

```
        .WithReference(mongodb)
        .WaitFor(mongodb);

// After adding all resources, run the app...
```

> **Tip**
>
> To configure the host port for the MongoExpressContainerResource chain a call to the WithHostPort API and provide the desired port number.

The preceding code adds a MongoDB Express resource that is configured to connect to the MongoDB server resource. The default credentials are:

- `ME_CONFIG_MONGODB_SERVER`: The name assigned to the parent `MongoDBServerResource`, in this case it would be `mongo`.
- `ME_CONFIG_BASICAUTH`: A value of `false`.
- `ME_CONFIG_MONGODB_PORT`: Assigned from the primary endpoint's target port of the parent `MongoDBServerResource`.
- `ME_CONFIG_MONGODB_ADMINUSERNAME`: The same username as configured in the parent `MongoDBServerResource`.
- `ME_CONFIG_MONGODB_ADMINPASSWORD`: The same password as configured in the parent `MongoDBServerResource`.

Additionally, the `WithMongoExpress` API exposes an optional `configureContainer` parameter of type `Action<IResourceBuilder<MongoExpressContainerResource>>` that you use to configure the MongoDB Express container resource.

Hosting integration health checks

The MongoDB hosting integration automatically adds a health check for the MongoDB server resource. The health check verifies that the MongoDB server resource is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.MongoDb NuGet package.

Client integration

To get started with the .NET Aspire MongoDB client integration, install the Aspire.MongoDB.Driver NuGet package in the client-consuming project, that is, the

project for the application that uses the MongoDB client. The MongoDB client integration registers a IMongoClient instance that you can use to interact with the MongoDB server resource. If your app host adds MongoDB database resources, the IMongoDatabase instance is also registered.

.NET CLI

.NET CLI

```
dotnet add package Aspire.MongoDB.Driver
```

> **Important**
>
> The `Aspire.MongoDB.Driver` NuGet package depends on the `MongoDB.Driver` NuGet package. With the release of version 3.0.0 of `MongoDB.Driver`, a binary breaking change was introduced. To address this, a new client integration package, `Aspire.MongoDB.Driver.v3`, was created. The original `Aspire.MongoDB.Driver` package continues to reference `MongoDB.Driver` version 2.30.0, ensuring compatibility with previous versions of the RabbitMQ client integration. The new `Aspire.MongoDB.Driver.v3` package references `MongoDB.Driver` version 3.0.0. In a future version of .NET Aspire, the `Aspire.MongoDB.Driver` will be updated to version `3.x` and the `Aspire.MongoDB.Driver.v3` package will be deprecated. For more information, see Upgrade to version 3.0.

Add MongoDB client

In the *Program.cs* file of your client-consuming project, call the AddMongoDBClient extension method on any IHostApplicationBuilder to register a `IMongoClient` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddMongoDBClient(connectionName: "mongodb");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the MongoDB server resource (or the database resource when provided) in the app

host project. In other words, when you call `AddDatabase` and provide a name of `mongodb` that same name should be used when calling `AddMongoDBClient`. For more information, see **Add MongoDB server resource and database resource**.

You can then retrieve the `IMongoClient` instance using dependency injection. For example, to retrieve the client from an example service:

C#

```
public class ExampleService(IMongoClient client)
{
    // Use client...
}
```

The `IMongoClient` is used to interact with the MongoDB server resource. It can be used to create databases that aren't already known to the app host project. When you define a MongoDB database resource in your app host, you could instead require that the dependency injection container provides an `IMongoDatabase` instance. For more information on dependency injection, see .NET dependency injection.

Add keyed MongoDB client

There might be situations where you want to register multiple `IMongoDatabase` instances with different connection names. To register keyed MongoDB clients, call the AddKeyedMongoDBClient method:

C#

```
builder.AddKeyedMongoDBClient(name: "mainDb");
builder.AddKeyedMongoDBClient(name: "loggingDb");
```

> **Important**
>
> When using keyed services, it's expected that your MongoDB resource configured two named databases, one for the `mainDb` and one for the `loggingDb`.

Then you can retrieve the `IMongoDatabase` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("mainDb")] IMongoDatabase mainDatabase,
    [FromKeyedServices("loggingDb")] IMongoDatabase loggingDatabase)
{
    // Use databases...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire MongoDB database integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddMongoDBClient()`:

C#

```
builder.AddMongoDBClient("mongo");
```

The connection string is retrieved from the `ConnectionStrings` configuration section. Consider the following MongoDB example JSON configuration:

JSON

```
{
  "ConnectionStrings": {
    "mongo": "mongodb://server:port/test",
  }
}
```

Alternatively, consider the following MongoDB Atlas example JSON configuration:

JSON

```
{
  "ConnectionStrings": {
    "mongo": "mongodb+srv://username:password@server.mongodb.net/",
  }
}
```

For more information on how to format this connection string, see MongoDB: ConnectionString documentation.

Use configuration providers

The .NET Aspire MongoDB integration supports Microsoft.Extensions.Configuration. It loads the MongoDBSettings from configuration by using the `Aspire:MongoDB:Driver` key. The following snippet is an example of a *appsettings.json* file that configures some of the options:

JSON

```
{
  "Aspire": {
    "MongoDB": {
      "Driver": {
        "ConnectionString": "mongodb://server:port/test",
        "DisableHealthChecks": false,
        "HealthCheckTimeout": 10000,
        "DisableTracing": false
      },
    }
  }
```

Use inline configurations

You can also pass the `Action<MongoDBSettings>` delegate to set up some or all the options inline:

C#

```
builder.AddMongoDBClient("mongodb",
    static settings => settings.ConnectionString =
"mongodb://server:port/test");
```

Configuration options

Here are the configurable options with corresponding default values:

Expand table

Name	Description
`ConnectionString`	The connection string of the MongoDB database database to connect to.

Name	Description
`DisableHealthChecks`	A boolean value that indicates whether the database health check is disabled or not.
`HealthCheckTimeout`	An `int?` value that indicates the MongoDB health check timeout in milliseconds.
`DisableTracing`	A boolean value that indicates whether the OpenTelemetry tracing is disabled or not.

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

By default, the .NET Aspire MongoDB client integration handles the following scenarios:

- Adds a health check when enabled that verifies that a connection can be made commands can be run against the MongoDB database within a certain amount of time.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire MongoDB database integration uses standard .NET logging, and you see log entries from the following categories:

- `MongoDB[.*]`: Any log entries from the MongoDB namespace.

Tracing

The .NET Aspire MongoDB database integration emits the following Tracing activities using OpenTelemetry:

- `MongoDB.Driver.Core.Extensions.DiagnosticSources`

Metrics

The .NET Aspire MongoDB database integration doesn't currently expose any OpenTelemetry metrics.

See also

- MongoDB database
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire MySQL integration

Article • 02/07/2025

Includes: ✅ Hosting integration and ✅ Client integration

MySQL is an open-source Relational Database Management System (RDBMS) that uses Structured Query Language (SQL) to manage and manipulate data. It's employed in a many different environments, from small projects to large-scale enterprise systems and it's a popular choice to host data that underpins microservices in a cloud-native application. The .NET Aspire MySQL database integration enables you to connect to existing MySQL databases or create new instances from .NET with the mysql container image.

Hosting integration

The MySQL hosting integration models the server as the MySqlServerResource type and the database as the MySqlDatabaseResource type. To access these types and APIs, add the Aspire.Hosting.MySql NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.MySql
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add MySQL server resource and database resource

In your app host project, call AddMySql to add and return a MySQL resource builder. Chain a call to the returned resource builder to AddDatabase, to add a MySQL database resource.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var mysql = builder.AddMySql("mysql")
                   .WithLifetime(ContainerLifetime.Persistent);
```

```
var mysqldb = mysql.AddDatabase("mysqldb");

var myService = builder.AddProject<Projects.ExampleProject>()
                       .WithReference(mysqldb)
                       .WaitFor(mysqldb);

// After adding all resources, run the app...
```

> **Note**
>
> The SQL Server container is slow to start, so it's best to use a *persistent* lifetime to avoid unnecessary restarts. For more information, see Container resource lifetime.

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `mysql` image, it creates a new MySQL instance on your local machine. A reference to your MySQL resource builder (the `mysql` variable) is used to add a database. The database is named `mysqldb` and then added to the `ExampleProject`. The MySQL resource includes default credentials with a `username` of `root` and a random `password` generated using the CreateDefaultPasswordParameter method.

When the app host runs, the password is stored in the app host's secret store. It's added to the `Parameters` section, for example:

JSON

```
{
  "Parameters:mysql-password": "<THE_GENERATED_PASSWORD>"
}
```

The name of the parameter is `mysql-password`, but really it's just formatting the resource name with a `-password` suffix. For more information, see Safe storage of app secrets in development in ASP.NET Core and Add MySQL resource with parameters.

The WithReference method configures a connection in the `ExampleProject` named `mysqldb`.

> **Tip**
>
> If you'd rather connect to an existing MySQL server, call AddConnectionString instead. For more information, see Reference existing resources.

Add a MySQL resource with a data volume

To add a data volume to the SQL Server resource, call the WithDataVolume method on the SQL Server resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var mysql = builder.AddMySql("mysql")
                   .WithDataVolume();

var mysqldb = mysql.AddDatabase("mysqldb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(mysqldb)
       .WaitFor(mysqldb);

// After adding all resources, run the app...
```

The data volume is used to persist the MySQL server data outside the lifecycle of its container. The data volume is mounted at the `/var/lib/mysql` path in the SQL Server container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

> ⚠ **Warning**
>
> The password is stored in the data volume. When using a data volume and if the password changes, it will not work until you delete the volume.

Add a MySQL resource with a data bind mount

To add a data bind mount to the MySQL resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var mysql = builder.AddMySql("mysql")
                   .WithDataBindMount(source: @"C:\MySql\Data");

var db = sql.AddDatabase("mysqldb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(mysqldb)
```

```
        .WaitFor(mysqldb);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the MySQL data across container restarts. The data bind mount is mounted at the `C:\MySql\Data` on Windows (or `/MySql/Data` on Unix) path on the host machine in the MySQL container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add MySQL resource with parameters

When you want to provide a root MySQL password explicitly, you can pass it as a parameter. Consider the following alternative example:

C#

```
var password = builder.AddParameter("password", secret: true);

var mysql = builder.AddMySql("mysql", password)
                   .WithLifetime(ContainerLifetime.Persistent);

var mysqldb = mysql.AddDatabase("mysqldb");

var myService = builder.AddProject<Projects.ExampleProject>()
                       .WithReference(mysqldb)
                       .WaitFor(mysqldb);
```

For more information, see External parameters.

Add a PhpMyAdmin resource

phpMyAdmin is a popular web-based administration tool for MySQL. You can use it to browse and modify MySQL objects such as databases, tables, views, and indexes. To use phpMyAdmin within your .NET Aspire solution, call the WithPhpMyAdmin method.

This method adds a new container resource to the solution that hosts phpMyAdmin and connects it to the MySQL container:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var mysql = builder.AddMySql("mysql")
                   .WithPhpMyAdmin();

var db = sql.AddDatabase("mysqldb");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(mysqldb)
       .WaitFor(mysqldb);

// After adding all resources, run the app...
```

When you run the solution, the .NET Aspire dashboard displays the phpMyAdmin resources with an endpoint. Select the link to the endpoint to view phpMyAdmin in a new browser tab.

Hosting integration health checks

The MySQL hosting integration automatically adds a health check for the MySQL resource. The health check verifies that the MySQL server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.MySql NuGet package.

Client integration

To get started with the .NET Aspire MySQL database integration, install the Aspire.MySqlConnector NuGet package in the client-consuming project, that is, the project for the application that uses the MySQL client. The MySQL client integration registers a `MySqlConnector.MySqlDataSource` instance that you can use to interact with the MySQL server.

.NET CLI

.NET CLI

```
dotnet add package Aspire.MySqlConnector
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add a MySQL data source

In the *Program.cs* file of your client-consuming project, call the AddMySqlDataSource extension method to register a `MySqlDataSource` for use via the dependency injection container. The method takes a `connectionName` parameter.

C#

```
builder.AddMySqlDataSource(connectionName: "mysqldb");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the MySQL database resource in the app host project. In other words, when you call `AddDatabase` and provide a name of `mysqldb` that same name should be used when calling `AddMySqlDataSource`. For more information, see **Add MySQL server resource and database resource**.

You can then retrieve the `MySqlConnector.MySqlDataSource` instance using dependency injection. For example, to retrieve the data source from an example service:

C#

```
public class ExampleService(MySqlDataSource dataSource)
{
    // Use dataSource...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed MySQL data source

There might be situations where you want to register multiple `MySqlDataSource` instances with different connection names. To register keyed MySQL data sources, call

the AddKeyedMySqlDataSource method:

```C#
builder.AddKeyedMySqlDataSource(name: "mainDb");
builder.AddKeyedMySqlDataSource(name: "loggingDb");
```

> **ⓘ Important**
>
> When using keyed services, it's expected that your MySQL resource configured two named databases, one for the `mainDb` and one for the `loggingDb`.

Then you can retrieve the `MySqlDatSource` instances using dependency injection. For example, to retrieve the connection from an example service:

```C#
public class ExampleService(
    [FromKeyedServices("mainDb")] MySqlDataSource mainDbConnection,
    [FromKeyedServices("loggingDb")] MySqlDataSource loggingDbConnection)
{
    // Use connections...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire MySQL database integration provides multiple options to configure the connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling AddMySqlDataSource method:

```C#
builder.AddMySqlDataSource(connectionName: "mysql");
```

Then the connection string is retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "mysql": "Server=mysql;Database=mysqldb"
  }
}
```

For more information on how to format this connection string, see MySqlConnector: ConnectionString documentation.

Use configuration providers

The .NET Aspire MySQL database integration supports Microsoft.Extensions.Configuration. It loads the MySqlConnectorSettings from configuration by using the `Aspire:MySqlConnector` key. The following snippet is an example of a *appsettings.json* file that configures some of the options:

JSON

```
{
  "Aspire": {
    "MySqlConnector": {
      "ConnectionString": "YOUR_CONNECTIONSTRING",
      "DisableHealthChecks": true,
      "DisableTracing": true
    }
  }
}
```

For the complete MySQL integration JSON schema, see Aspire.MySqlConnector/ConfigurationSchema.json.

Use inline delegates

Also you can pass the `Action<MySqlConnectorSettings>` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddMySqlDataSource(
    "mysql",
```

```
    static settings => settings.DisableHealthChecks  = true);
```

Client integration health checks

By default, .NET Aspire integrations enable health checks for all services. For more information, see .NET Aspire integrations overview.

The .NET Aspire MySQL database integration:

- Adds the health check when MySqlConnectorSettings.DisableHealthChecks is `false`, which verifies that a connection can be made and commands can be run against the MySQL database.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire MySQL integration uses the following log categories:

- `MySqlConnector.ConnectionPool`
- `MySqlConnector.MySqlBulkCopy`
- `MySqlConnector.MySqlCommand`
- `MySqlConnector.MySqlConnection`
- `MySqlConnector.MySqlDataSource`

Tracing

The .NET Aspire MySQL integration emits the following tracing activities using OpenTelemetry:

- `MySqlConnector`

Metrics

The .NET Aspire MySQL integration will emit the following metrics using OpenTelemetry:

- MySqlConnector
  - `db.client.connections.create_time`
  - `db.client.connections.use_time`
  - `db.client.connections.wait_time`
  - `db.client.connections.idle.max`
  - `db.client.connections.idle.min`
  - `db.client.connections.max`
  - `db.client.connections.pending_requests`
  - `db.client.connections.timeouts`
  - `db.client.connections.usage`

See also

- MySQL database
- .NET Aspire database containers sample
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire NATS integration

Article • 02/25/2025

Includes: ✅ Hosting integration and ✅ Client integration

NATS is a high-performance, secure, distributed messaging system. The .NET Aspire NATS integration enables you to connect to existing NATS instances, or create new instances from .NET with the docker.io/library/nats container image.

Hosting integration

NATS hosting integration for .NET Aspire models a NATS server as the NatsServerResource type. To access this type, install the Aspire.Hosting.Nats NuGet package in the app host project, then add it with the builder.

.NET CLI

```
.NET CLI
dotnet add package Aspire.Hosting.Nats
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add NATS server resource

In your app host project, call AddNats on the `builder` instance to add a NATS server resource:

```
C#
var builder = DistributedApplication.CreateBuilder(args);

var nats = builder.AddNats("nats");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(nats);

// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/library/nats` image, it creates a new NATS server instance on your local machine. A reference to your NATS server (the `nats` variable) is added to the `ExampleProject`.

The WithReference method configures a connection in the `ExampleProject` named `"nats"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing NATS server, call **AddConnectionString** instead. For more information, see **Reference existing resources**.

Add NATS server resource with JetStream

To add the NATS JetStream to the NATS server resource, call the WithJetStream method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var nats = builder.AddNats("nats");
                  .WithJetStream();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(nats);

// After adding all resources, run the app...
```

The NATS JetStream functionality provides a built-in persistence engine called JetStream which enables messages to be stored and replayed at a later time.

Add NATS server resource with authentication parameters

When you want to explicitly provide the username and password, you can provide those as parameters. Consider the following alternative example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);
```

```
var username = builder.AddParameter("username");
var password = builder.AddParameter("password", secret: true);

var nats = builder.AddNats(
    name: "nats",
    userName: username,
    password: password);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(nats);


// After adding all resources, run the app...
```

For more information, see External parameters.

Add NATS server resource with data volume

To add a data volume to the NATS server resource, call the WithDataVolume method on the NATS server resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var nats = builder.AddNats("nats");
                  .WithDataVolume(isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(nats);

// After adding all resources, run the app...
```

The data volume is used to persist the NATS server data outside the lifecycle of its container. The data volume is mounted at the `/var/lib/nats` path in the NATS server container. A name is generated at random unless you provide a set the `name` parameter. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add NATS server resource with data bind mount

To add a data bind mount to the NATS server resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var nats = builder.AddNats("nats");
                  .WithDataBindMount(
                      source: @"C:\NATS\Data",
                      isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(nats);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the NATS server data across container restarts. The data bind mount is mounted at the `C:\NATS\Data` on Windows (or `/NATS/Data` on Unix) path on the host machine in the NATS server container. For more information on data bind mounts, see Docker docs: Bind mounts.

Hosting integration health checks

The NATS hosting integration automatically adds a health check for the NATS server resource. The health check verifies that the NATS server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Nats NuGet package.

Client integration

To get started with the .NET Aspire NATS client integration, install the Aspire.NATS.Net NuGet package in the client-consuming project, that is, the project for the application that uses the NATS client. The NATS client integration registers an INatsConnection instance that you can use to interact with NATS.

.NET CLI

.NET CLI

```
dotnet add package Aspire.NATS.Net
```

Add NATS client

In the *Program.cs* file of your client-consuming project, call the AddNatsClient extension method on any IHostApplicationBuilder to register an `INatsConnection` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddNatsClient(connectionName: "nats");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the NATS server resource in the app host project. For more information, see **Add NATS server resource**.

You can then retrieve the `INatsConnection` instance using dependency injection. For example, to retrieve the client from a service:

C#

```
public class ExampleService(INatsConnection connection)
{
    // Use connection...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed NATS client

There might be situations where you want to register multiple `INatsConnection` instances with different connection names. To register keyed NATS clients, call the AddKeyedNatsClient method:

C#

```
builder.AddKeyedNatsClient(name: "chat");
builder.AddKeyedNatsClient(name: "queue");
```

Then you can retrieve the `IConnection` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("chat")] INatsConnection chatConnection,
    [FromKeyedServices("queue")] INatsConnection queueConnection)
{
    // Use connections...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire NATS integration provides multiple options to configure the NATS connection based on the requirements and conventions of your project.

Use a connection string

Provide the name of the connection string when you call `builder.AddNatsClient`:

C#

```
builder.AddNatsClient(connectionName: "nats");
```

The connection string is retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "nats": "nats://nats:4222"
  }
}
```

See the ConnectionString documentation for more information on how to format this connection string.

Use configuration providers

The .NET Aspire NATS integration supports Microsoft.Extensions.Configuration. It loads the NatsClientSettings from configuration by using the `Aspire:Nats:Client` key. The following snippet is an example of a *appsettings.json* file that configures some of the options:

JSON

```
{
  "Aspire": {
    "Nats": {
      "Client": {
        "ConnectionString": "nats://nats:4222",
        "DisableHealthChecks": true,
        "DisableTracing": true
      }
    }
  }
}
```

For the complete NATS client integration JSON schema, see Aspire.NATS.Net/ConfigurationSchema.json.

Use inline delegates

Pass the `Action<NatsClientSettings> configureSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddNatsClient(
    "nats",
    static settings => settings.DisableHealthChecks  = true);
```

NATS in the .NET Aspire manifest

NATS isn't part of the .NET Aspire deployment manifest. It's recommended you set up a secure production NATS server outside of .NET Aspire.

Client integration health checks

By default, .NET Aspire integrations enable health checks for all services. For more information, see .NET Aspire integrations overview.

The .NET Aspire NATS integration handles the following:

- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire NATS integration uses the following log categories:

- `NATS`

Tracing

The .NET Aspire NATS integration emits the following tracing activities:

- `NATS.Net`

See also

- NATS.Net quickstart
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire Orleans integration

Article • 01/09/2025

Orleans has built-in support for .NET Aspire. .NET Aspire's application model lets you describe the services, databases, and other resources and infrastructure in your app and how they relate to each other. Orleans provides a straightforward way to build distributed applications that are elastically scalable and fault-tolerant. You can use .NET Aspire to configure and orchestrate Orleans and its dependencies, such as by providing Orleans with cluster membership and storage.

Orleans is represented as a resource in .NET Aspire. Unlike other integrations, the Orleans integration doesn't create a container and doesn't require a separate client integration package. Instead you complete the Orleans configuration in the .NET Aspire app host project.

> **Note**
>
> This integration requires Orleans version 8.1.0 or later.

Hosting integration

The Orleans hosting integration models an Orleans service as the OrleansService type. To access this type and APIs, add the Aspire.Hosting.Orleans NuGet package in the app host project.

.NET CLI

```
dotnet add package Aspire.Hosting.Orleans
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add an Orleans resource

In your app host project, call AddOrleans to add and return an Orleans service resource builder. The name provided to the Orleans resource is for diagnostic purposes. For most

applications, a value of `"default"` suffices.

C#

```
var orleans = builder.AddOrleans("default")
```

Use Azure storage for clustering tables and grain storage

In an Orleans app, the fundamental building block is a **grain**. Grains can have durable states. You must store the durable state for a grain somewhere. In a .NET Aspire application, **Azure Blob Storage** is one possible location.

Orleans hosts register themselves in a database and use that database to find each other and form a cluster. They store which servers are members of which silos in a database table. You can use either relational or NoSQL databases to store this information. In a .NET Aspire application, a popular choice to store this table is **Azure Table Storage**.

To configure Orleans with clustering and grain storage in Azure, install the Aspire.Hosting.Azure.Storage NuGet package in the app host project:

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Storage
```

In your app host project, after you call AddOrleans, configure the Orleans resource with clustering and grain storage using the WithClustering and WithGrainStorage methods respectively:

C#

```
// Add the resources which you will use for Orleans clustering and
// grain state storage.
var storage = builder.AddAzureStorage("storage").RunAsEmulator();
var clusteringTable = storage.AddTables("clustering");
var grainStorage = storage.AddBlobs("grain-state");

// Add the Orleans resource to the Aspire DistributedApplication
// builder, then configure it with Azure Table Storage for clustering
// and Azure Blob Storage for grain storage.
var orleans = builder.AddOrleans("default")
```

```
                        .WithClustering(clusteringTable)
                        .WithGrainStorage("Default", grainStorage);
```

The preceding code tells Orleans that any service referencing it must also reference the `clusteringTable` resource.

Add an Orleans server project in the app host

Now you can add a new project, enrolled in .NET Aspire orchestration, to your solution as an Orleans server. It will take part in the Orleans cluster as a silo with constituent grains. Reference the Orleans resource from your server project using `WithReference(orleans)`. When you reference the Orleans resource from your service, those resources are also referenced:

C#

```
// Add your server project and reference your 'orleans' resource from it.
// It can join the Orleans cluster as a silo.
// This implicitly adds references to the required resources.
// In this case, that is the 'clusteringTable' resource declared earlier.
builder.AddProject<Projects.OrleansServer>("silo")
       .WithReference(orleans)
       .WithReplicas(3);
```

Add an Orleans client project in the app host

Orleans clients communicate with grains hosted on Orleans servers. In a .NET Aspire app, for example, you might have a front-end Web site that calls grains in an Orleans cluster. Reference the Orleans resource from your Orleans client using `WithReference(orleans.AsClient())`.

C#

```
// Reference the Orleans resource as a client from the 'frontend'
// project so that it can connect to the Orleans cluster.
builder.AddProject<Projects.OrleansClient>("frontend")
       .WithReference(orleans.AsClient())
       .WithExternalHttpEndpoints()
       .WithReplicas(3);
```

Create the Orleans server project

Now that the app host project is completed, you can implement the Orleans server project. Let's start by adding the necessary NuGet packages:

.NET CLI

In the folder for the Orleans server project, run these commands:

.NET CLI

```
dotnet add package Aspire.Azure.Data.Tables
dotnet add package Aspire.Azure.Storage.Blobs
dotnet add package Microsoft.Orleans.Server
dotnet add package Microsoft.Orleans.Persistence.AzureStorage
dotnet add package Microsoft.Orleans.Clustering.AzureStorage
```

Next, in the *Program.cs* file of your Orleans server project, add the Azure Storage blob and tables clients and then call UseOrleans.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.AddServiceDefaults();
builder.AddKeyedAzureTableClient("clustering");
builder.AddKeyedAzureBlobClient("grain-state");
builder.UseOrleans();
```

The following code is a complete example of an Orleans server project, including a grain named `CounterGrain`:

C#

```
using Orleans.Runtime;
using OrleansContracts;

var builder = WebApplication.CreateBuilder(args);

builder.AddServiceDefaults();
builder.AddKeyedAzureTableClient("clustering");
builder.AddKeyedAzureBlobClient("grain-state");
builder.UseOrleans();

var app = builder.Build();

app.MapGet("/", () => "OK");

await app.RunAsync();
```

```
public sealed class CounterGrain(
    [PersistentState("count")] IPersistentState<int> count) : ICounterGrain
{
    public ValueTask<int> Get()
    {
        return ValueTask.FromResult(count.State);
    }

    public async ValueTask<int> Increment()
    {
        var result = ++count.State;
        await count.WriteStateAsync();
        return result;
    }
}
```

Create an Orleans client project

In the Orleans client project, add the same NuGet packages:

.NET CLI

.NET CLI

```
dotnet add package Aspire.Azure.Data.Tables
dotnet add package Aspire.Azure.Storage.Blobs
dotnet add package Microsoft.Orleans.Client
dotnet add package Microsoft.Orleans.Persistence.AzureStorage
dotnet add package Microsoft.Orleans.Clustering.AzureStorage
```

Next, in the *Program.cs* file of your Orleans client project, add the Azure table storage client and then call UseOrleansClient.

C#

```
builder.AddKeyedAzureTableClient("clustering");
builder.UseOrleansClient();
```

The following code is a complete example of an Orleans client project. It calls the `CounterGrain` grain defined in the Orleans server example above:

C#

```
using OrleansContracts;

var builder = WebApplication.CreateBuilder(args);
```

```
builder.AddServiceDefaults();
builder.AddKeyedAzureTableClient("clustering");
builder.UseOrleansClient();

var app = builder.Build();

app.MapGet("/counter/{grainId}", async (IClusterClient client, string
grainId) =>
{
    var grain = client.GetGrain<ICounterGrain>(grainId);
    return await grain.Get();
});

app.MapPost("/counter/{grainId}", async (IClusterClient client, string
grainId) =>
{
    var grain = client.GetGrain<ICounterGrain>(grainId);
    return await grain.Increment();
});

app.UseFileServer();

await app.RunAsync();
```

Enabling OpenTelemetry

By convention, .NET Aspire solutions include a project for defining default configuration and behavior for your service. This project is called the *service defaults* project and templates create it with a name ending in *ServiceDefaults*. To configure Orleans for OpenTelemetry in .NET Aspire, apply configuration to your service defaults project following the Orleans observability guide.

Modify the `ConfigureOpenTelemetry` method to add the Orleans *meters* and *tracing* instruments. The following code snippet shows the modified *Extensions.cs* file from a service defaults project that includes metrics and traces from Orleans.

C#

```
public static IHostApplicationBuilder ConfigureOpenTelemetry(this
IHostApplicationBuilder builder)
{
    builder.Logging.AddOpenTelemetry(logging =>
    {
        logging.IncludeFormattedMessage = true;
        logging.IncludeScopes = true;
    });

    builder.Services.AddOpenTelemetry()
```

```
            .WithMetrics(metrics =>
            {
                metrics.AddAspNetCoreInstrumentation()
                    .AddHttpClientInstrumentation()
                    .AddRuntimeInstrumentation()
                    .AddMeter("Microsoft.Orleans");
            })
            .WithTracing(tracing =>
            {
                tracing.AddSource("Microsoft.Orleans.Runtime");
                tracing.AddSource("Microsoft.Orleans.Application");

                tracing.AddAspNetCoreInstrumentation()
                    .AddHttpClientInstrumentation();
            });

        builder.AddOpenTelemetryExporters();

        return builder;
    }
```

Supported providers

The Orleans Aspire integration supports a limited subset of Orleans providers today:

- Clustering:
  - Redis
  - Azure Storage Tables
- Persistence:
  - Redis
  - Azure Storage Tables
  - Azure Storage Blobs
- Reminders:
  - Redis
  - Azure Storage Tables
- Grain directory:
  - Redis
  - Azure Storage Tables

Streaming providers aren't supported as of Orleans version 8.1.0.

Next steps

Microsoft Orleans documentation | Explore the Orleans voting sample app

.NET Aspire PostgreSQL integration

Article • 02/07/2025

Includes: ✅ Hosting integration and ✅ Client integration

PostgreSQL is a powerful, open source object-relational database system with many years of active development that has earned it a strong reputation for reliability, feature robustness, and performance. The .NET Aspire PostgreSQL integration provides a way to connect to existing PostgreSQL databases, or create new instances from .NET with the docker.io/library/postgres container image.

Hosting integration

The PostgreSQL hosting integration models various PostgreSQL resources as the following types.

- PostgresServerResource
- PostgresDatabaseResource
- PgAdminContainerResource
- PgWebContainerResource

To access these types and APIs for expressing them as resources in your app host project, install the Aspire.Hosting.PostgreSQL NuGet package:

.NET CLI

```
.NET CLI
dotnet add package Aspire.Hosting.PostgreSQL
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add PostgreSQL server resource

In your app host project, call AddPostgres on the `builder` instance to add a PostgreSQL server resource then call AddDatabase on the `postgres` instance to add a database resource as shown in the following example:

```
C#
```

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres");
var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/library/postgres` image, it creates a new PostgreSQL server instance on your local machine. A reference to your PostgreSQL server and your PostgreSQL database instance (the `postgresdb` variable) are used to add a dependency to the `ExampleProject`. The PostgreSQL server resource includes default credentials with a `username` of `"postgres"` and randomly generated `password` using the CreateDefaultPasswordParameter method.

The WithReference method configures a connection in the `ExampleProject` named `"messaging"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing PostgreSQL server, call **AddConnectionString** instead. For more information, see **Reference existing resources**.

Add PostgreSQL pgAdmin resource

When adding PostgreSQL resources to the `builder` with the `AddPostgres` method, you can chain calls to WithPgAdmin to add the **dpage/pgadmin4** container. This container is a cross-platform client for PostgreSQL databases, that serves a web-based admin dashboard. Consider the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres")
                      .WithPgAdmin();

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);
```

```
// After adding all resources, run the app...
```

The preceding code adds a container based on the `docker.io/dpage/pgadmin4` image. The container is used to manage the PostgreSQL server and database resources. The `WithPgAdmin` method adds a container that serves a web-based admin dashboard for PostgreSQL databases.

Configure the pgAdmin host port

To configure the host port for the pgAdmin container, call the WithHostPort method on the PostgreSQL server resource. The following example shows how to configure the host port for the pgAdmin container:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres")
                      .WithPgAdmin(pgAdmin => pgAdmin.WithHostPort(5050));

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

The preceding code adds and configures the host port for the pgAdmin container. The host port is otherwise randomly assigned.

Add PostgreSQL pgWeb resource

When adding PostgreSQL resources to the `builder` with the `AddPostgres` method, you can chain calls to WithPgWeb to add the **sosedoff/pgweb** container. This container is a cross-platform client for PostgreSQL databases, that serves a web-based admin dashboard. Consider the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres")
                      .WithPgWeb();
```

```
var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

The preceding code adds a container based on the `docker.io/sosedoff/pgweb` image. All registered PostgresDatabaseResource instances are used to create a configuration file per instance, and each config is bound to the **pgweb** container bookmark directory. For more information, see PgWeb docs: Server connection bookmarks.

Configure the pgWeb host port

To configure the host port for the pgWeb container, call the WithHostPort method on the PostgreSQL server resource. The following example shows how to configure the host port for the pgAdmin container:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres")
                      .WithPgWeb(pgWeb => pgWeb.WithHostPort(5050));

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

The preceding code adds and configures the host port for the pgWeb container. The host port is otherwise randomly assigned.

Add PostgreSQL server resource with data volume

To add a data volume to the PostgreSQL server resource, call the WithDataVolume method on the PostgreSQL server resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres")
                      .WithDataVolume(isReadOnly: false);
```

```
var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

The data volume is used to persist the PostgreSQL server data outside the lifecycle of its container. The data volume is mounted at the `/var/lib/postgresql/data` path in the PostgreSQL server container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add PostgreSQL server resource with data bind mount

To add a data bind mount to the PostgreSQL server resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres")
                      .WithDataBindMount(
                          source: @"C:\PostgreSQL\Data",
                          isReadOnly: false);

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the PostgreSQL server data across container restarts. The data bind mount is mounted at the

`C:\PostgreSQL\Data` on Windows (or `/PostgreSQL/Data` on Unix) path on the host machine in the PostgreSQL server container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add PostgreSQL server resource with init bind mount

To add an init bind mount to the PostgreSQL server resource, call the WithInitBindMount method:

```
C#

var builder = DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres")
                      .WithInitBindMount(@"C:\PostgreSQL\Init");

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

The init bind mount relies on the host machine's filesystem to initialize the PostgreSQL server database with the containers *init* folder. This folder is used for initialization, running any executable shell scripts or *.sql* command files after the *postgres-data* folder is created. The init bind mount is mounted at the `C:\PostgreSQL\Init` on Windows (or `/PostgreSQL/Init` on Unix) path on the host machine in the PostgreSQL server container.

Add PostgreSQL server resource with parameters

When you want to explicitly provide the username and password used by the container image, you can provide these credentials as parameters. Consider the following alternative example:

```
C#

var builder = DistributedApplication.CreateBuilder(args);

var username = builder.AddParameter("username", secret: true);
var password = builder.AddParameter("password", secret: true);

var postgres = builder.AddPostgres("postgres", username, password);
var postgresdb = postgres.AddDatabase("postgresdb");
```

```
var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(postgresdb);

// After adding all resources, run the app...
```

For more information on providing parameters, see External parameters.

Hosting integration health checks

The PostgreSQL hosting integration automatically adds a health check for the PostgreSQL server resource. The health check verifies that the PostgreSQL server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Npgsql NuGet package.

Client integration

To get started with the .NET Aspire PostgreSQL client integration, install the Aspire.Npgsql NuGet package in the client-consuming project, that is, the project for the application that uses the PostgreSQL client. The PostgreSQL client integration registers an NpgsqlDataSource instance that you can use to interact with PostgreSQL.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Npgsql
```

Add Npgsql client

In the *Program.cs* file of your client-consuming project, call the AddNpgsqlDataSource extension method on any IHostApplicationBuilder to register an `NpgsqlDataSource` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddNpgsqlDataSource(connectionName: "postgresdb");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the PostgreSQL server resource in the app host project. For more information, see Add PostgreSQL server resource.

After adding `NpgsqlDataSource` to the builder, you can get the `NpgsqlDataSource` instance using dependency injection. For example, to retrieve your data source object from an example service define it as a constructor parameter and ensure the `ExampleService` class is registered with the dependency injection container:

```
C#

public class ExampleService(NpgsqlDataSource dataSource)
{
    // Use dataSource...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed Npgsql client

There might be situations where you want to register multiple `NpgsqlDataSource` instances with different connection names. To register keyed Npgsql clients, call the AddKeyedNpgsqlDataSource method:

```
C#

builder.AddKeyedNpgsqlDataSource(name: "chat");
builder.AddKeyedNpgsqlDataSource(name: "queue");
```

Then you can retrieve the `NpgsqlDataSource` instances using dependency injection. For example, to retrieve the connection from an example service:

```
C#

public class ExampleService(
    [FromKeyedServices("chat")] NpgsqlDataSource chatDataSource,
    [FromKeyedServices("queue")] NpgsqlDataSource queueDataSource)
{
    // Use data sources...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire PostgreSQL integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling the AddNpgsqlDataSource method:

C#

```
builder.AddNpgsqlDataSource("postgresdb");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "postgresdb": "Host=myserver;Database=postgresdb"
  }
}
```

For more information, see the ConnectionString.

Use configuration providers

The .NET Aspire PostgreSQL integration supports Microsoft.Extensions.Configuration. It loads the NpgsqlSettings from *appsettings.json* or other configuration files by using the `Aspire:Npgsql` key. Example *appsettings.json* that configures some of the options:

The following example shows an *appsettings.json* file that configures some of the available options:

JSON

```
{
  "Aspire": {
```

```
    "Npgsql": {
      "ConnectionString": "Host=myserver;Database=postgresdb",
      "DisableHealthChecks": false,
      "DisableTracing": true,
      "DisableMetrics": false
    }
  }
}
```

For the complete PostgreSQL client integration JSON schema, see Aspire.Npgsql/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<NpgsqlSettings> configureSettings` delegate to set up some or all the options inline, for example to disable health checks:

C#

```
builder.AddNpgsqlDataSource(
    "postgresdb",
     static settings => settings.DisableHealthChecks = true);
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

- Adds the NpgSqlHealthCheck, which verifies that commands can be successfully executed against the underlying Postgres database.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some

of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire PostgreSQL integration uses the following log categories:

- `Npgsql.Connection`
- `Npgsql.Command`
- `Npgsql.Transaction`
- `Npgsql.Copy`
- `Npgsql.Replication`
- `Npgsql.Exception`

Tracing

The .NET Aspire PostgreSQL integration will emit the following tracing activities using OpenTelemetry:

- `Npgsql`

Metrics

The .NET Aspire PostgreSQL integration will emit the following metrics using OpenTelemetry:

- Npgsql:
  - `ec_Npgsql_bytes_written_per_second`
  - `ec_Npgsql_bytes_read_per_second`
  - `ec_Npgsql_commands_per_second`
  - `ec_Npgsql_total_commands`
  - `ec_Npgsql_current_commands`
  - `ec_Npgsql_failed_commands`
  - `ec_Npgsql_prepared_commands_ratio`
  - `ec_Npgsql_connection_pools`
  - `ec_Npgsql_multiplexing_average_commands_per_batch`
  - `ec_Npgsql_multiplexing_average_write_time_per_batch`

See also

- PostgreSQL docs
- .NET Aspire Azure PostgreSQL integration
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire Qdrant integration

Article • 01/18/2025

Includes: ✅ Hosting integration and ✅ Client integration

Qdrant is an open-source vector similarity search engine that efficiently stores, indexes, and searches large-scale vector data. It's commonly used in machine learning, artificial intelligence, and data science applications.

Vector data encodes information as mathematical vectors, which are arrays of numbers or coordinates. Machine learning and AI systems often use vectors to represent unstructured objects like images, text, audio, or video. Each dimension in the vector describes a specific characteristic of the object. By comparing them, systems can classify, search, and identify clusters of objects.

In this article, you learn how to use the .NET Aspire Qdrant integration. The .NET Aspire Qdrant integration enables you to connect to existing Qdrant databases or create new instances with the qdrant/qdrant container image.

Hosting integration

The Qdrant hosting integration models the server as the QdrantServerResource type. To access this type and APIs, add the Aspire.Hosting.Qdrant NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Qdrant
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Qdrant resource

In your app host project, call AddQdrant to add and return a Qdrant resource builder.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var qdrant = builder.AddQdrant("qdrant")
                    .WithLifetime(ContainerLifetime.Persistent);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(qdrant)
       .WaitFor(qdrant);

// After adding all resources, run the app...
```

> **Note**
>
> The Qdrant container can be slow to start, so it's best to use a *persistent* lifetime to avoid unnecessary restarts. For more information, see Container resource lifetime.

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `qdrant/qdrant` image, it creates a new Qdrant instance on your local machine. The resource is named `qdrant` and then added to the `ExampleProject`.

The WithReference method configures a connection in the `ExampleProject` named `qdrant`.

> **Tip**
>
> If you'd rather connect to an existing Qdrant server, call AddConnectionString instead. For more information, see Reference existing resources.

> **Tip**
>
> The `qdrant/qdrant` container image includes a web UI that you can use to explore your vectors and administer the database. To access this tool, start your .NET Aspire solution and then, in the .NET Aspire dashboard, select the endpoint for the Qdrant resource. In your browser's address bar, append **/dashboard** and press `Enter`.

Handling API keys and passing other parameters for the Qdrant resource

To connect to Qdrant a client must pass the right API key. In the above code, when .NET Aspire adds a Qdrant resource to your solution, it sets the API key to a random string. If

you want to use a specific API key instead, you can pass it as an `apiKey` parameter:

C#

```
var apiKey = builder.AddParameter("apiKey", secret: true);

var qdrant = builder.AddQdrant("qdrant", apiKey);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(qdrant);
```

Qdrant supports configuration-based default API keys by using the environment variable `QDRANT__SERVICE__API_KEY`.

The preceding code gets a parameter to pass to the `AddQdrant` API, and internally assigns the parameter to the `QDRANT__SERVICE__API_KEY` environment variable of the Qdrant container. The `apiKey` parameter is usually specified as a *user secret*:

JSON

```
{
  "Parameters": {
    "apiKey": "Non-default-P@ssw0rd"
  }
}
```

For more information, see External parameters.

Add Qdrant resource with data volume

To add a data volume to the Qdrant resource, call the WithDataVolume extension method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var qdrant = builder.AddQdrant("qdrant")
                    .WithLifetime(ContainerLifetime.Persistent)
                    .WithDataVolume();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(qdrant)
       .WaitFor(qdrant);

// After adding all resources, run the app...
```

The data volume is used to persist the Qdrant data outside the lifecycle of its container. The data volume is mounted at the `/qdrant/storage` path in the Qdrant container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add Qdrant resource with data bind mount

To add a data bind mount to the Qdrant resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var qdrant = builder.AddQdrant("qdrant")
                    .WithLifetime(ContainerLifetime.Persistent)
                    .WithDataBindMount(source: @"C:\Qdrant\Data");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(qdrant)
       .WaitFor(qdrant);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Qdrant data across container restarts. The data bind mount is mounted at the `C:\Qdrant\Data` folder on Windows (or `/Qdrant/Data` on Unix) on the host machine in the Qdrant container. For more information on data bind mounts, see Docker docs: Bind mounts.

Hosting integration health checks

The Qdrant hosting integration automatically adds a health check for the Qdrant resource. The health check verifies that Qdrant is running and that a connection can be established to it.

Client integration

To get started with the .NET Aspire Qdrant client integration, install the Aspire.Qdrant.Client NuGet package in the client-consuming project, that is, the project for the application that uses the Qdrant client. The Qdrant client integration registers a Qdrant.Client.QdrantClient instance that you can use to interact with Qdrant vector data.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Qdrant.Client
```

Add a Qdrant client

In the *Program.cs* file of your client-consuming project, call the AddQdrantClient extension method on any IHostApplicationBuilder to register a `QdrantClient` for use through the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddQdrantClient("qdrant");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Qdrant resource in the app host project. In other words, when you call `AddQdrant` and provide a name of `qdrant` that same name should be used when calling `AddQdrantClient`. For more information, see Add Qdrant resource.

You can then retrieve the `QdrantClient` instance using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(QdrantClient client)
{
```

```
    // Use client...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed Qdrant client

There might be situations where you want to register multiple `QdrantClient` instances with different connection names. To register keyed Qdrant clients, call the AddKeyedQdrantClient method:

C#

```
builder.AddKeyedQdrantClient(name: "mainQdrant");
builder.AddKeyedQdrantClient(name: "loggingQdrant");
```

Then you can retrieve the `QdrantClient` instances using dependency injection. For example, to retrieve the connections from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("mainQdrant")] QdrantClient mainQdrantClient,
    [FromKeyedServices("loggingQdrant")] QdrantClient loggingQdrantClient)
{
    // Use clients...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire Qdrant client integration provides multiple options to configure the connection to Qdrant based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddQdrantClient()`:

C#

```
builder.AddQdrantClient("qdrant");
```

Then .NET Aspire retrieves the connection string from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "qdrant": "Endpoint=http://localhost:6334;Key=123456!@#$%"
  }
}
```

By default the `QdrantClient` uses the gRPC API endpoint.

Use configuration providers

The .NET Aspire Qdrant client integration supports Microsoft.Extensions.Configuration. It loads the QdrantClientSettings from configuration by using the `Aspire:Qdrant:Client` key. The following is an example of an *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "Qdrant": {
      "Client": {
        "Endpoint": "http://localhost:6334/",
        "Key": "123456!@#$%"
      }
    }
  }
}
```

For the complete Qdrant client integration JSON schema, see Aspire.Qdrant.Client/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<QdrantClientSettings> configureSettings` delegate to set up some or all the options inline, for example to set the API key from code:

C#

```
builder.AddQdrantClient(
    "qdrant",
    settings => settings.Key = "12345!@#$%");
```

Client integration health checks

By default, .NET Aspire integrations enable health checks for all services. For more information, see .NET Aspire integrations overview.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Qdrant integration uses standard .NET logging, and you'll see log entries from the following category:

- `Qdrant.Client`

Tracing

The .NET Aspire Qdrant integration doesn't currently emit tracing activities because they are not supported by the `Qdrant.Client` library.

Metrics

The .NET Aspire Qdrant integration doesn't currently emit metrics because they are not supported by the `Qdrant.Client` library.

See also

- Qdrant

- Qdrant documentation
- Qdrant GitHub repo
- Qdrant .NET SDK
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire RabbitMQ integration

Article • 02/25/2025

Includes: ✅ Hosting integration and ✅ Client integration

RabbitMQ is a reliable messaging and streaming broker, which is easy to deploy on cloud environments, on-premises, and on your local machine. The .NET Aspire RabbitMQ integration enables you to connect to existing RabbitMQ instances, or create new instances from .NET with the docker.io/library/rabbitmq container image.

Hosting integration

The RabbitMQ hosting integration models a RabbitMQ server as the RabbitMQServerResource type. To access this type and its APIs add the Aspire.Hosting.RabbitMQ NuGet package in the app host project.

.NET CLI

```
.NET CLI
dotnet add package Aspire.Hosting.RabbitMQ
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add RabbitMQ server resource

In your app host project, call AddRabbitMQ on the `builder` instance to add a RabbitMQ server resource:

```
C#
var builder = DistributedApplication.CreateBuilder(args);

var rabbitmq = builder.AddRabbitMQ("messaging");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(rabbitmq);

// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/library/rabbitmq` image, it creates a new RabbitMQ server instance on your local machine. A reference to your RabbitMQ server (the `rabbitmq` variable) is added to the `ExampleProject`. The RabbitMQ server resource includes default credentials with a `username` of `"guest"` and randomly generated `password` using the CreateDefaultPasswordParameter method.

The WithReference method configures a connection in the `ExampleProject` named `"messaging"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing RabbitMQ server, call **AddConnectionString** instead. For more information, see **Reference existing resources**.

Add RabbitMQ server resource with management plugin

To add the RabbitMQ management plugin to the RabbitMQ server resource, call the WithManagementPlugin method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var rabbitmq = builder.AddRabbitMQ("messaging")
                      .WithManagementPlugin();

builder.AddProject<Projects.ExampleProject>()
        .WithReference(rabbitmq);

// After adding all resources, run the app...
```

The RabbitMQ management plugin provides an HTTP-based API for management and monitoring of your RabbitMQ server. .NET Aspire adds another container image docker.io/library/rabbitmq-management to the app host that runs the management plugin.

Add RabbitMQ server resource with data volume

To add a data volume to the RabbitMQ server resource, call the WithDataVolume method on the RabbitMQ server resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var rabbitmq = builder.AddRabbitMQ("messaging")
                      .WithDataVolume(isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
        .WithReference(rabbitmq);

// After adding all resources, run the app...
```

The data volume is used to persist the RabbitMQ server data outside the lifecycle of its container. The data volume is mounted at the `/var/lib/rabbitmq` path in the RabbitMQ server container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add RabbitMQ server resource with data bind mount

To add a data bind mount to the RabbitMQ server resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var rabbitmq = builder.AddRabbitMQ("messaging")
                      .WithDataBindMount(
                          source: @"C:\RabbitMQ\Data",
                          isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
        .WithReference(rabbitmq);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the RabbitMQ server data across container restarts. The data bind mount is mounted at the `C:\RabbitMQ\Data` on Windows (or `/RabbitMQ/Data` on Unix) path on the host machine in the RabbitMQ server container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add RabbitMQ server resource with parameters

When you want to explicitly provide the username and password used by the container image, you can provide these credentials as parameters. Consider the following alternative example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var username = builder.AddParameter("username", secret: true);
var password = builder.AddParameter("password", secret: true);

var rabbitmq = builder.AddRabbitMQ("messaging", username, password);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(rabbitmq);

// After adding all resources, run the app...
```

For more information on providing parameters, see External parameters.

Hosting integration health checks

The RabbitMQ hosting integration automatically adds a health check for the RabbitMQ server resource. The health check verifies that the RabbitMQ server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Rabbitmq NuGet package.

Client integration

To get started with the .NET Aspire RabbitMQ client integration, install the Aspire.RabbitMQ.Client NuGet package in the client-consuming project, that is, the project for the application that uses the RabbitMQ client. The RabbitMQ client

integration registers an IConnection instance that you can use to interact with RabbitMQ.

.NET CLI

.NET CLI

```
dotnet add package Aspire.RabbitMQ.Client
```

> **Important**
>
> The `Aspire.RabbitMQ.Client` NuGet package depends on the `RabbitMQ.Client` NuGet package. With the release of version 7.0.0 of `RabbitMQ.Client`, a binary breaking change was introduced. To address this, a new client integration package, `Aspire.RabbitMQ.Client.v7`, was created. The original `Aspire.RabbitMQ.Client` package continues to reference `RabbitMQ.Client` version 6.8.1, ensuring compatibility with previous versions of the RabbitMQ client integration. The new `Aspire.RabbitMQ.Client.v7` package references `RabbitMQ.Client` version 7.0.0. In a future version of .NET Aspire, the `Aspire.RabbitMQ.Client` will be updated to version `7.x` and the `Aspire.RabbitMQ.Client.v7` package will be deprecated. For more information, see Migrating to RabbitMQ .NET Client 7.x.

Add RabbitMQ client

In the *Program.cs* file of your client-consuming project, call the AddRabbitMQClient extension method on any IHostApplicationBuilder to register an `IConnection` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddRabbitMQClient(connectionName: "messaging");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the RabbitMQ server resource in the app host project. For more information, see Add RabbitMQ server resource.

You can then retrieve the `IConnection` instance using dependency injection. For example, to retrieve the connection from an example service:

```C#
public class ExampleService(IConnection connection)
{
    // Use connection...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed RabbitMQ client

There might be situations where you want to register multiple `IConnection` instances with different connection names. To register keyed RabbitMQ clients, call the AddKeyedRabbitMQClient method:

```C#
builder.AddKeyedRabbitMQClient(name: "chat");
builder.AddKeyedRabbitMQClient(name: "queue");
```

Then you can retrieve the `IConnection` instances using dependency injection. For example, to retrieve the connection from an example service:

```C#
public class ExampleService(
    [FromKeyedServices("chat")] IConnection chatConnection,
    [FromKeyedServices("queue")] IConnection queueConnection)
{
    // Use connections...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire RabbitMQ integration provides multiple options to configure the connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the

For the complete RabbitMQ client integration JSON schema, see Aspire.RabbitMQ.Client/ConfigurationSchema.json.

Use inline delegates

Also you can pass the `Action<RabbitMQClientSettings> configureSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

```
C#

builder.AddRabbitMQClient(
    "messaging",
    static settings => settings.DisableHealthChecks = true);
```

You can also set up the IConnectionFactory using the `Action<IConnectionFactory> configureConnectionFactory` delegate parameter of the `AddRabbitMQClient` method. For example to set the client provided name for connections:

```
C#

builder.AddRabbitMQClient(
    "messaging",
    configureConnectionFactory:
        static factory => factory.ClientProvidedName = "MyApp");
```

Client integration health checks

By default, .NET Aspire integrations enable health checks for all services. For more information, see .NET Aspire integrations overview.

The .NET Aspire RabbitMQ integration:

- Adds the health check when RabbitMQClientSettings.DisableHealthChecks is `false`, which attempts to connect to and create a channel on the RabbitMQ server.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations

overview. Depending on the backing service, some integrations might only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire RabbitMQ integration uses the following log categories:

- `RabbitMQ.Client`

Tracing

The .NET Aspire RabbitMQ integration emits the following tracing activities using OpenTelemetry:

- `Aspire.RabbitMQ.Client`

Metrics

The .NET Aspire RabbitMQ integration currently doesn't support metrics by default.

See also

- Send messages with RabbitMQ in .NET Aspire
- RabbitMQ .NET Client docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

Stack Exchange Redis®* caching overview

Article • 02/11/2025

With .NET Aspire, there are several ways to use caching in your applications. One popular option is to use Stack Exchange Redis, which is a high-performance data store that can be used to store frequently accessed data. This article provides an overview of Stack Exchange Redis caching and links to resources that help you use it in your applications.

To use multiple Redis caching integrations in your application, see Tutorial: Implement caching with .NET Aspire integrations. If you're interested in using the Redis Cache for Azure, see Tutorial: Deploy a .NET Aspire project with a Redis Cache to Azure.

Redis serialization protocol (RESP)

The Redis serialization protocol (RESP) is a binary-safe protocol that Redis uses to communicate with clients. RESP is a simple, text-based protocol that is easy to implement and efficient to parse. RESP is used to send commands to Redis and receive responses from Redis. RESP is designed to be fast and efficient, making it well-suited for use in high-performance applications. For more information, see Redis serialization protocol specification.

In addition to Redis itself, there are two well-maintained implementations of RESP for .NET:

- Garnet: Garnet is a remote cache-store from Microsoft Research that offers strong performance (throughput and latency), scalability, storage, recovery, cluster sharding, key migration, and replication features. Garnet can work with existing Redis clients.
- Valkey: A flexible distributed key-value datastore that supports both caching and beyond caching workloads.

.NET Aspire lets you easily choose either the Redis, Garnet, or Valkey RESP protocol in your applications based on your requirements. All of the .NET Aspire Redis client integrations can be used with either the Redis, Garnet, or Valkey RESP protocol.

Caching

Caching is a technique used to store frequently accessed data in memory. This helps to reduce the time it takes to retrieve the data from the original source, such as a database or a web service. Caching can significantly improve the performance of an application by reducing the number of requests made to the original source. To access the Redis IConnectionMultiplexer object, you use the Aspire.StackExchange.Redis NuGet package:

.NET Aspire Stack Exchange Redis integration

.NET Aspire Stack Exchange Redis integration (Garnet)

.NET Aspire Stack Exchange Redis integration (Valkey)

Distributed caching

Distributed caching is a type of caching that stores data across multiple servers. This allows the data to be shared between multiple instances of an application, which can help to improve scalability and performance. Distributed caching can be used to store a wide variety of data, such as session state, user profiles, and frequently accessed data. To use Redis distributed caching in your application (the IDistributedCache interface), use the Aspire.StackExchange.Redis.DistributedCaching NuGet package:

.NET Aspire Stack Exchange Redis distributed caching integration

.NET Aspire Stack Exchange Redis distributed caching integration (Garnet)

.NET Aspire Stack Exchange Redis distributed caching integration (Valkey)

Output caching

Output caching is a type of caching that stores the output of a web page or API response. This allows the response to be served directly from the cache, rather than generating it from scratch each time. Output caching can help to improve the performance of a web application by reducing the time it takes to generate a response. To use declarative Redis output caching with either the OutputCacheAttribute or the CacheOutput method in your application, use the [Aspire.StackExchange.Redis.OutputCaching](https://www.nuget.org/packages/Aspire.StackExchange.Redis.OutputCaching) NuGet package:

.NET Aspire Stack Exchange Redis output caching integration

.NET Aspire Stack Exchange Redis output caching integration (Garnet)

.NET Aspire Stack Exchange Redis output caching integration (Valkey)

See also

- Caching in .NET
- Overview of Caching in ASP.NET Core
- Distributed caching in .NET
- Distributed caching in ASP.NET Core
- Output caching middleware in ASP.NET Core

> *: *Redis is a registered trademark of Redis Ltd. Any rights therein are reserved to Redis Ltd. Any use by Microsoft is for referential purposes only and does not indicate any sponsorship, endorsement or affiliation between Redis and Microsoft.* *Return to top*?

.NET Aspire Redis®* integration

Article • 02/11/2025

Includes: ✅ Hosting integration and ✅ Client integration

Redis is the world's fastest data platform for caching, vector search, and NoSQL databases. The .NET Aspire Redis integration enables you to connect to existing Redis instances, or create new instances from .NET with the docker.io/library/redis container image.

Hosting integration

The Redis hosting integration models a Redis resource as the RedisResource type. To access this type and APIs for expressing them as resources in your app host project, add the Aspire.Hosting.Redis NuGet package:

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Redis
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Redis resource

In your app host project, call AddRedis on the `builder` instance to add a Redis resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/Redis/Redis` image, it creates a new Redis instance on your local machine. A reference to your Redis resource (the `cache` variable) is added to the `ExampleProject`.

The WithReference method configures a connection in the `ExampleProject` named `"cache"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing Redis instance, call **AddConnectionString** instead. For more information, see **Reference existing resources**.

Add Redis resource with Redis Insights

To add the Redis Insights to the Redis resource, call the `WithRedisInsight` method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
                   .WithRedisInsight();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

Redis Insights is a free graphical interface for analyzing Redis data across all operating systems and Redis deployments with the help of our AI assistant, Redis Copilot. .NET Aspire adds another container image docker.io/redis/redisinsight to the app host that runs the commander app.

> **Note**
>
> To configure the host port for the `RedisInsightResource` chain a call to the `WithHostPort` API and provide the desired port number.

Add Redis resource with Redis Commander

To add the Redis Commander to the Redis resource, call the WithRedisCommander method:

```C#
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
                   .WithRedisCommander();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

Redis Commander is a Node.js web application used to view, edit, and manage a Redis Database. .NET Aspire adds another container image docker.io/rediscommander/redis-commander to the app host that runs the commander app.

> **Tip**
>
> To configure the host port for the **RedisCommanderResource** chain a call to the **WithHostPort** API and provide the desired port number.

Add Redis resource with data volume

To add a data volume to the Redis resource, call the WithDataVolume method on the Redis resource:

```C#
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
                   .WithDataVolume(isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The data volume is used to persist the Redis data outside the lifecycle of its container. The data volume is mounted at the `/data` path in the Redis container and when a `name` parameter isn't provided, the name is generated at random. For more information on

data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add Redis resource with data bind mount

To add a data bind mount to the Redis resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
                   .WithDataBindMount(
                       source: @"C:\Redis\Data",
                       isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Redis data across container restarts. The data bind mount is mounted at the `C:\Redis\Data` on Windows (or `/Redis/Data` on Unix) path on the host machine in the Redis container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add Redis resource with persistence

To add persistence to the Redis resource, call the WithPersistence method with either the data volume or data bind mount:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
```

```
                    .WithDataVolume()
                    .WithPersistence(
                        interval: TimeSpan.FromMinutes(5),
                        keysChangedThreshold: 100);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The preceding code adds persistence to the Redis resource by taking snapshots of the Redis data at a specified interval and threshold. The `interval` is time between snapshot exports and the `keysChangedThreshold` is the number of key change operations required to trigger a snapshot. For more information on persistence, see Redis docs: Persistence.

Hosting integration health checks

The Redis hosting integration automatically adds a health check for the appropriate resource type. The health check verifies that the server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Redis NuGet package.

Client integration

To get started with the .NET Aspire Stack Exchange Redis client integration, install the Aspire.StackExchange.Redis NuGet package in the client-consuming project, that is, the project for the application that uses the Redis client. The Redis client integration registers an IConnectionMultiplexer instance that you can use to interact with Redis.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis
```

Add Redis client

In the *Program.cs* file of your client-consuming project, call the AddRedisClient extension method on any IHostApplicationBuilder to register an `IConnectionMultiplexer` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddRedisClient(connectionName: "cache");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Redis resource in the app host project. For more information, see Add Redis resource.

You can then retrieve the `IConnection` instance using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(IConnectionMultiplexer connectionMux)
{
    // Use connection multiplexer...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed Redis client

There might be situations where you want to register multiple `IConnectionMultiplexer` instances with different connection names. To register keyed Redis clients, call the AddKeyedRedisClient method:

C#

```
builder.AddKeyedRedisClient(name: "chat");
builder.AddKeyedRedisClient(name: "queue");
```

Then you can retrieve the `IConnectionMultiplexer` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("chat")] IConnectionMultiplexer chatConnectionMux,
    [FromKeyedServices("queue")] IConnectionMultiplexer queueConnectionMux)
{
    // Use connections...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire Stack Exchange Redis client integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling AddRedis:

C#

```
builder.AddRedis("cache");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the Stack Exchange Redis configuration docs.

Use configuration providers

The .NET Aspire Stack Exchange Redis integration supports Microsoft.Extensions.Configuration. It loads the StackExchangeRedisSettings from

configuration by using the `Aspire:StackExchange:Redis` key. Example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        },
        "DisableHealthChecks": true,
        "DisableTracing": false
      }
    }
  }
}
```

For the complete Redis client integration JSON schema, see Aspire.StackExchange.Redis/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings>` delegate to set up some or all the options inline, for example to configure `DisableTracing`:

C#

```
builder.AddRedisClient(
    "cache",
    static settings => settings.DisableTracing = true);
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

The .NET Aspire Stack Exchange Redis integration handles the following:

- Adds the health check when StackExchangeRedisSettings.DisableHealthChecks is `false`, which attempts to connect to the container instance.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Stack Exchange Redis integration uses the following log categories:

- `Aspire.StackExchange.Redis`

Tracing

The .NET Aspire Stack Exchange Redis integration will emit the following tracing activities using OpenTelemetry:

- `OpenTelemetry.Instrumentation.StackExchangeRedis`

Metrics

The .NET Aspire Stack Exchange Redis integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- Stack Exchange Redis docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

> *: *Redis is a registered trademark of Redis Ltd. Any rights therein are reserved to Redis Ltd. Any use by Microsoft is for referential purposes only and does not indicate any*

> *sponsorship, endorsement or affiliation between Redis and Microsoft.* *Return to top?*

.NET Aspire Redis® distributed caching integration

Article • 02/11/2025

Includes: ✅ Hosting integration and ✅ Client integration

Learn how to use the .NET Aspire Redis distributed caching integration. The `Aspire.StackExchange.Redis.DistributedCaching` library is used to register an IDistributedCache provider backed by a RR

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/Redis/Redis` image, it creates a new Redis instance on your local machine. A reference to your Redis resource (the `cache` variable) is added to the `ExampleProject`.

The WithReference method configures a connection in the `ExampleProject` named `"cache"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing Redis instance, call **AddConnectionString** instead. For more information, see **Reference existing resources**.

Add Redis resource with Redis Insights

To add the Redis Insights to the Redis resource, call the `WithRedisInsight` method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
                   .WithRedisInsight();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

Redis Insights is a free graphical interface for analyzing Redis data across all operating systems and Redis deployments with the help of our AI assistant, Redis Copilot. .NET Aspire adds another container image docker.io/redis/redisinsight to the app host that runs the commander app.

> **Note**
>
> To configure the host port for the `RedisInsightResource` chain a call to the `WithHostPort` API and provide the desired port number.

Add Redis resource with Redis Commander

To add the Redis Commander to the Redis resource, call the WithRedisCommander method:

```C#
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
                   .WithRedisCommander();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

Redis Commander is a Node.js web application used to view, edit, and manage a Redis Database. .NET Aspire adds another container image docker.io/rediscommander/redis-commander to the app host that runs the commander app.

> **Tip**
>
> To configure the host port for the **RedisCommanderResource** chain a call to the **WithHostPort** API and provide the desired port number.

Add Redis resource with data volume

To add a data volume to the Redis resource, call the WithDataVolume method on the Redis resource:

```C#
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
                   .WithDataVolume(isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The data volume is used to persist the Redis data outside the lifecycle of its container. The data volume is mounted at the `/data` path in the Redis container and when a `name` parameter isn't provided, the name is generated at random. For more information on

data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add Redis resource with data bind mount

To add a data bind mount to the Redis resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
                   .WithDataBindMount(
                       source: @"C:\Redis\Data",
                       isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Redis data across container restarts. The data bind mount is mounted at the `C:\Redis\Data` on Windows (or `/Redis/Data` on Unix) path on the host machine in the Redis container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add Redis resource with persistence

To add persistence to the Redis resource, call the WithPersistence method with either the data volume or data bind mount:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
```

```
                    .WithDataVolume()
                    .WithPersistence(
                        interval: TimeSpan.FromMinutes(5),
                        keysChangedThreshold: 100);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The preceding code adds persistence to the Redis resource by taking snapshots of the Redis data at a specified interval and threshold. The `interval` is time between snapshot exports and the `keysChangedThreshold` is the number of key change operations required to trigger a snapshot. For more information on persistence, see Redis docs: Persistence.

Hosting integration health checks

The Redis hosting integration automatically adds a health check for the appropriate resource type. The health check verifies that the server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Redis NuGet package.

Client integration

To get started with the .NET Aspire Redis distributed caching integration, install the Aspire.StackExchange.Redis.DistributedCaching NuGet package in the client-consuming project, i.e., the project for the application that uses the Redis distributed caching client. The Redis client integration registers an IDistributedCache instance that you can use to interact with Redis.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.DistributedCaching
```

Add Redis client

In the *Program.cs* file of your client-consuming project, call the AddRedisDistributedCache extension to register the required services for distributed caching and add a IDistributedCache for use via the dependency injection container.

```
C#

builder.AddRedisDistributedCache(connectionName: "cache");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Redis resource in the app host project. For more information, see **Add Redis resource**.

You can then retrieve the `IDistributedCache` instance using dependency injection. For example, to retrieve the cache from a service:

```
C#

public class ExampleService(IDistributedCache cache)
{
    // Use cache...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed Redis client

There might be situations where you want to register multiple `IDistributedCache` instances with different connection names. To register keyed Redis clients, call the AddKeyedRedisDistributedCache method:

```
C#

builder.AddKeyedRedisDistributedCache(name: "chat");
builder.AddKeyedRedisDistributedCache(name: "product");
```

Then you can retrieve the `IDistributedCache` instances using dependency injection. For example, to retrieve the connection from an example service:

```
C#

public class ExampleService(
    [FromKeyedServices("chat")] IDistributedCache chatCache,
```

```
    [FromKeyedServices("product")] IDistributedCache productCache)
{
    // Use caches...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire Redis distributed caching integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddRedisDistributedCache`:

C#

```
builder.AddRedisDistributedCache("cache");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the Stack Exchange Redis configuration docs.

Use configuration providers

The .NET Aspire Stack Exchange Redis distributed caching integration supports Microsoft.Extensions.Configuration. It loads the StackExchangeRedisSettings from configuration by using the `Aspire:StackExchange:Redis` key. Example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        },
        "DisableHealthChecks": true,
        "DisableTracing": false
      }
    }
  }
}
```

For the complete Redis distributed caching client integration JSON schema, see Aspire.StackExchange.Redis.DistributedCaching/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings>` delegate to set up some or all the options inline, for example to configure `DisableTracing`:

C#

```
builder.AddRedisDistributedCache(
    "cache",
    settings => settings.DisableTracing = true);
```

You can also set up the ConfigurationOptions using the `Action<ConfigurationOptions> configureOptions` delegate parameter of the `AddRedisDistributedCache` method. For example to set the connection timeout:

C#

```
builder.AddRedisDistributedCache(
    "cache",
     null,
     static options => options.ConnectTimeout = 3_000);
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

The .NET Aspire Redis distributed caching integration handles the following:

- Adds the health check when StackExchangeRedisSettings.DisableHealthChecks is `false`, which attempts to connect to the container instance.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Redis distributed caching integration uses the following Log categories:

- `Aspire.StackExchange.Redis`
- `Microsoft.Extensions.Caching.StackExchangeRedis`

Tracing

The .NET Aspire Redis distributed caching integration will emit the following Tracing activities using OpenTelemetry:

- `OpenTelemetry.Instrumentation.StackExchangeRedis`

Metrics

The .NET Aspire Redis Distributed caching integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- Stack Exchange Redis docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

> *: *Redis is a registered trademark of Redis Ltd. Any rights therein are reserved to Redis Ltd. Any use by Microsoft is for referential purposes only and does not indicate any sponsorship, endorsement or affiliation between Redis and Microsoft.* *Return to top*?

.NET Aspire Redis®* output caching integration

Article • 02/11/2025

Includes: ✅ Hosting integration and ✅ Client integration

Learn how to use the .NET Aspire Redis output caching integration. The `Aspire.StackExchange.Redis.OutputCaching` client integration is used to register an ASP.NET Core Output Caching provider backed by a Redis server with the docker.io/library/redis container image.

Hosting integration

The Redis hosting integration models a Redis resource as the RedisResource type. To access this type and APIs for expressing them as resources in your app host project, add the Aspire.Hosting.Redis NuGet package:

.NET CLI

```
.NET CLI
dotnet add package Aspire.Hosting.Redis
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Redis resource

In your app host project, call AddRedis on the `builder` instance to add a Redis resource:

```
C#
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/Redis/Redis` image, it creates a new Redis instance on your local machine. A reference to your Redis resource (the `cache` variable) is added to the `ExampleProject`.

The WithReference method configures a connection in the `ExampleProject` named `"cache"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing Redis instance, call **AddConnectionString** instead. For more information, see **Reference existing resources**.

Add Redis resource with Redis Insights

To add the Redis Insights to the Redis resource, call the `WithRedisInsight` method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
                   .WithRedisInsight();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

Redis Insights is a free graphical interface for analyzing Redis data across all operating systems and Redis deployments with the help of our AI assistant, Redis Copilot. .NET Aspire adds another container image docker.io/redis/redisinsight to the app host that runs the commander app.

> **Note**
>
> To configure the host port for the `RedisInsightResource` chain a call to the `WithHostPort` API and provide the desired port number.

Add Redis resource with Redis Commander

To add the Redis Commander to the Redis resource, call the WithRedisCommander method:

```C#
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
                   .WithRedisCommander();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

Redis Commander is a Node.js web application used to view, edit, and manage a Redis Database. .NET Aspire adds another container image docker.io/rediscommander/redis-commander to the app host that runs the commander app.

> **Tip**
>
> To configure the host port for the **RedisCommanderResource** chain a call to the **WithHostPort** API and provide the desired port number.

Add Redis resource with data volume

To add a data volume to the Redis resource, call the WithDataVolume method on the Redis resource:

```C#
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
                   .WithDataVolume(isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The data volume is used to persist the Redis data outside the lifecycle of its container. The data volume is mounted at the `/data` path in the Redis container and when a `name` parameter isn't provided, the name is generated at random. For more information on

data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add Redis resource with data bind mount

To add a data bind mount to the Redis resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
                   .WithDataBindMount(
                       source: @"C:\Redis\Data",
                       isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Redis data across container restarts. The data bind mount is mounted at the `C:\Redis\Data` on Windows (or `/Redis/Data` on Unix) path on the host machine in the Redis container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add Redis resource with persistence

To add persistence to the Redis resource, call the WithPersistence method with either the data volume or data bind mount:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache")
```

```
                    .WithDataVolume()
                    .WithPersistence(
                        interval: TimeSpan.FromMinutes(5),
                        keysChangedThreshold: 100);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The preceding code adds persistence to the Redis resource by taking snapshots of the Redis data at a specified interval and threshold. The `interval` is time between snapshot exports and the `keysChangedThreshold` is the number of key change operations required to trigger a snapshot. For more information on persistence, see Redis docs: Persistence.

Hosting integration health checks

The Redis hosting integration automatically adds a health check for the appropriate resource type. The health check verifies that the server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Redis NuGet package.

Client integration

To get started with the .NET Aspire Stack Exchange Redis output caching client integration, install the Aspire.StackExchange.Redis.OutputCaching NuGet package in the client-consuming project, that is, the project for the application that uses the output caching client. The Redis output caching client integration registers services required for enabling CacheOutput method calls and [OutputCache] attribute usage to rely on Redis as its caching mechanism.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.OutputCaching
```

Add output caching

In the *Program.cs* file of your client-consuming project, call the AddRedisOutputCache extension method on any IHostApplicationBuilder to register the required services for output caching.

```C#
builder.AddRedisOutputCache(connectionName: "cache");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Redis resource in the app host project. For more information, see **Add Redis resource**.

Add the middleware to the request processing pipeline by calling UseOutputCache(IApplicationBuilder):

```C#
var app = builder.Build();

app.UseOutputCache();
```

For minimal API apps, configure an endpoint to do caching by calling CacheOutput, or by applying the OutputCacheAttribute, as shown in the following examples:

```C#
app.MapGet("/cached", () => "Hello world!")
   .CacheOutput();

app.MapGet(
    "/attribute",
    [OutputCache] () => "Hello world!");
```

For apps with controllers, apply the `[OutputCache]` attribute to the action method. For Razor Pages apps, apply the attribute to the Razor page class.

Configuration

The .NET Aspire Stack Exchange Redis output caching integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling AddRedisOutputCache:

C#

```
builder.AddRedisOutputCache(connectionName: "cache");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the Stack Exchange Redis configuration docs.

Use configuration providers

The .NET Aspire Stack Exchange Redis output caching integration supports Microsoft.Extensions.Configuration. It loads the StackExchangeRedisSettings from configuration by using the `Aspire:StackExchange:Redis` key. Example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        },
        "DisableHealthChecks": true,
        "DisableTracing": false
      }
    }
  }
}
```

For the complete Redis output caching client integration JSON schema, see Aspire.StackExchange.Redis.OutputCaching/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings> configurationSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

```C#
builder.AddRedisOutputCache(
    "cache",
    static settings => settings.DisableHealthChecks  = true);
```

You can also set up the ConfigurationOptions using the `Action<ConfigurationOptions> configureOptions` delegate parameter of the AddRedisOutputCache method. For example to set the connection timeout:

```C#
builder.AddRedisOutputCache(
    "cache",
    static settings => settings.ConnectTimeout = 3_000);
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

The .NET Aspire Stack Exchange Redis output caching integration handles the following:

- Adds the health check when StackExchangeRedisSettings.DisableHealthChecks is `false`, which attempts to connect to the container instance.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Stack Exchange Redis output caching integration uses the following Log categories:

- `Aspire.StackExchange.Redis`
- `Microsoft.AspNetCore.OutputCaching.StackExchangeRedis`

Tracing

The .NET Aspire Stack Exchange Redis output caching integration will emit the following Tracing activities using OpenTelemetry:

- `OpenTelemetry.Instrumentation.StackExchangeRedis`

Metrics

The .NET Aspire Stack Exchange Redis output caching integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- Stack Exchange Redis docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

> *: *Redis is a registered trademark of Redis Ltd. Any rights therein are reserved to Redis Ltd. Any use by Microsoft is for referential purposes only and does not indicate any sponsorship, endorsement or affiliation between Redis and Microsoft.* *Return to top?*

.NET Aspire Redis®* integration

Article • 02/11/2025

Includes: ✅ Hosting integration and ✅ Client integration

Garnet is a a high-performance cache-store from Microsoft Research and complies with the Redis serialization protocol (RESP). The .NET Aspire Redis integration enables you to connect to existing Garnet instances, or create new instances from .NET with the ghcr.io/microsoft/garnet container image.

Hosting integration

The Garnet hosting integration models a Garnet resource as the GarnetResource type. To access this type and APIs that allow you to add it to your Aspire.Hosting.Garnet NuGet package in the app host project.

.NET CLI

```
.NET CLI
dotnet add package Aspire.Hosting.Garnet
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Garnet resource

In your app host project, call AddGarnet on the `builder` instance to add a Garnet resource:

```
C#
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddGarnet("cache");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `ghcr.io/microsoft/garnet` image, it creates a new Garnet instance on your local machine. A reference to your Garnet resource (the `cache` variable) is added to the `ExampleProject`.

The WithReference method configures a connection in the `ExampleProject` named `"cache"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing Garnet instance, call **AddConnectionString** instead. For more information, see **Reference existing resources**.

Add Garnet resource with data volume

To add a data volume to the Garnet resource, call the AddGarnet method on the Garnet resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddGarnet("cache")
                   .WithDataVolume(isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The data volume is used to persist the Garnet data outside the lifecycle of its container. The data volume is mounted at the `/data` path in the Garnet container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add Garnet resource with data bind mount

To add a data bind mount to the Garnet resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddGarnet("cache")
                   .WithDataBindMount(
                       source: @"C:\Garnet\Data",
                       isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Garnet data across container restarts. The data bind mount is mounted at the `C:\Garnet\Data` on Windows (or `/Garnet/Data` on Unix) path on the host machine in the Garnet container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add Garnet resource with persistence

To add persistence to the Garnet resource, call the WithPersistence method with either the data volume or data bind mount:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddGarnet("cache")
                   .WithDataVolume()
                   .WithPersistence(
                       interval: TimeSpan.FromMinutes(5),
                       keysChangedThreshold: 100);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The preceding code adds persistence to the Redis resource by taking snapshots of the Garnet data at a specified interval and threshold. The `interval` is time between snapshot exports and the `keysChangedThreshold` is the number of key change operations required to trigger a snapshot. For more information on persistence, see Redis docs: Persistence.

Hosting integration health checks

The Redis hosting integration automatically adds a health check for the appropriate resource type. The health check verifies that the server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Redis NuGet package.

Client integration

To get started with the .NET Aspire Stack Exchange Redis client integration, install the Aspire.StackExchange.Redis NuGet package in the client-consuming project, that is, the project for the application that uses the Redis client. The Redis client integration registers an IConnectionMultiplexer instance that you can use to interact with Redis.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis
```

Add Redis client

In the *Program.cs* file of your client-consuming project, call the AddRedisClient extension method on any IHostApplicationBuilder to register an `IConnectionMultiplexer` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddRedisClient(connectionName: "cache");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Garnet resource in the app host project. For more information, see Add Garnet resource.

You can then retrieve the `IConnection` instance using dependency injection. For example, to retrieve the connection from an example service:

```
C#

public class ExampleService(IConnectionMultiplexer connectionMux)
{
    // Use connection multiplexer...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed Redis client

There might be situations where you want to register multiple `IConnectionMultiplexer` instances with different connection names. To register keyed Redis clients, call the AddKeyedRedisClient method:

```
C#

builder.AddKeyedRedisClient(name: "chat");
builder.AddKeyedRedisClient(name: "queue");
```

Then you can retrieve the `IConnectionMultiplexer` instances using dependency injection. For example, to retrieve the connection from an example service:

```
C#

public class ExampleService(
    [FromKeyedServices("chat")] IConnectionMultiplexer chatConnectionMux,
    [FromKeyedServices("queue")] IConnectionMultiplexer queueConnectionMux)
{
    // Use connections...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire Stack Exchange Redis client integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling AddGarnet:

C#

```
builder.AddGarnet("cache");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the Stack Exchange Redis configuration docs.

Use configuration providers

The .NET Aspire Stack Exchange Redis integration supports Microsoft.Extensions.Configuration. It loads the StackExchangeRedisSettings from configuration by using the `Aspire:StackExchange:Redis` key. Example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        },
        "DisableHealthChecks": true,
```

```
        "DisableTracing": false
      }
    }
  }
}
```

For the complete Redis client integration JSON schema, see Aspire.StackExchange.Redis/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings>` delegate to set up some or all the options inline, for example to configure `DisableTracing`:

C#

```
builder.AddRedisClient(
    "cache",
    static settings => settings.DisableTracing = true);
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

The .NET Aspire Stack Exchange Redis integration handles the following:

- Adds the health check when StackExchangeRedisSettings.DisableHealthChecks is `false`, which attempts to connect to the container instance.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not

metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Stack Exchange Redis integration uses the following log categories:

- `Aspire.StackExchange.Redis`

Tracing

The .NET Aspire Stack Exchange Redis integration will emit the following tracing activities using OpenTelemetry:

- `OpenTelemetry.Instrumentation.StackExchangeRedis`

Metrics

The .NET Aspire Stack Exchange Redis integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- Stack Exchange Redis docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

> *: *Redis is a registered trademark of Redis Ltd. Any rights therein are reserved to Redis Ltd. Any use by Microsoft is for referential purposes only and does not indicate any sponsorship, endorsement or affiliation between Redis and Microsoft.* *Return to top*?

.NET Aspire Redis® distributed caching integration

Article • 02/11/2025

Includes: ✅ Hosting integration and ✅ Client integration

Learn how to use the .NET Aspire Redis distributed caching integration. The `Aspire.StackExchange.Redis.DistributedCaching` library is used to register an IDistributedCache provider backed by a Garnet server with the ghcr.io/microsoft/garnet container image.

Hosting integration

The Garnet hosting integration models a Garnet resource as the GarnetResource type. To access this type and APIs that allow you to add it to your Aspire.Hosting.Garnet NuGet package in the app host project.

.NET CLI

```
.NET CLI
dotnet add package Aspire.Hosting.Garnet
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Garnet resource

In your app host project, call AddGarnet on the `builder` instance to add a Garnet resource:

```
C#
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddGarnet("cache");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);
```

```
// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `ghcr.io/microsoft/garnet` image, it creates a new Garnet instance on your local machine. A reference to your Garnet resource (the `cache` variable) is added to the `ExampleProject`.

The WithReference method configures a connection in the `ExampleProject` named `"cache"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing Garnet instance, call **AddConnectionString** instead. For more information, see **Reference existing resources**.

Add Garnet resource with data volume

To add a data volume to the Garnet resource, call the AddGarnet method on the Garnet resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddGarnet("cache")
                   .WithDataVolume(isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The data volume is used to persist the Garnet data outside the lifecycle of its container. The data volume is mounted at the `/data` path in the Garnet container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add Garnet resource with data bind mount

To add a data bind mount to the Garnet resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddGarnet("cache")
                   .WithDataBindMount(
                       source: @"C:\Garnet\Data",
                       isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Garnet data across container restarts. The data bind mount is mounted at the `C:\Garnet\Data` on Windows (or `/Garnet/Data` on Unix) path on the host machine in the Garnet container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add Garnet resource with persistence

To add persistence to the Garnet resource, call the WithPersistence method with either the data volume or data bind mount:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddGarnet("cache")
                   .WithDataVolume()
                   .WithPersistence(
                       interval: TimeSpan.FromMinutes(5),
                       keysChangedThreshold: 100);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);
```

```
// After adding all resources, run the app...
```

The preceding code adds persistence to the Redis resource by taking snapshots of the Garnet data at a specified interval and threshold. The `interval` is time between snapshot exports and the `keysChangedThreshold` is the number of key change operations required to trigger a snapshot. For more information on persistence, see Redis docs: Persistence.

Hosting integration health checks

The Redis hosting integration automatically adds a health check for the appropriate resource type. The health check verifies that the server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Redis NuGet package.

Client integration

To get started with the .NET Aspire Redis distributed caching integration, install the Aspire.StackExchange.Redis.DistributedCaching NuGet package in the client-consuming project, i.e., the project for the application that uses the Redis distributed caching client. The Redis client integration registers an IDistributedCache instance that you can use to interact with Redis.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.DistributedCaching
```

Add Redis client

In the *Program.cs* file of your client-consuming project, call the AddRedisDistributedCache extension to register the required services for distributed caching and add a IDistributedCache for use via the dependency injection container.

C#

```
builder.AddRedisDistributedCache(connectionName: "cache");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Garnet resource in the app host project. For more information, see Add Garnet resource.

You can then retrieve the `IDistributedCache` instance using dependency injection. For example, to retrieve the cache from a service:

C#

```
public class ExampleService(IDistributedCache cache)
{
    // Use cache...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed Redis client

There might be situations where you want to register multiple `IDistributedCache` instances with different connection names. To register keyed Redis clients, call the AddKeyedRedisDistributedCache method:

C#

```
builder.AddKeyedRedisDistributedCache(name: "chat");
builder.AddKeyedRedisDistributedCache(name: "product");
```

Then you can retrieve the `IDistributedCache` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("chat")] IDistributedCache chatCache,
    [FromKeyedServices("product")] IDistributedCache productCache)
{
    // Use caches...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire Redis distributed caching integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddRedisDistributedCache`:

C#

```
builder.AddRedisDistributedCache("cache");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the Stack Exchange Redis configuration docs.

Use configuration providers

The .NET Aspire Stack Exchange Redis distributed caching integration supports Microsoft.Extensions.Configuration. It loads the StackExchangeRedisSettings from configuration by using the `Aspire:StackExchange:Redis` key. Example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
```

```
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        },
        "DisableHealthChecks": true,
        "DisableTracing": false
      }
    }
  }
}
```

For the complete Redis distributed caching client integration JSON schema, see Aspire.StackExchange.Redis.DistributedCaching/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings>` delegate to set up some or all the options inline, for example to configure `DisableTracing`:

C#

```
builder.AddRedisDistributedCache(
    "cache",
    settings => settings.DisableTracing = true);
```

You can also set up the ConfigurationOptions using the `Action<ConfigurationOptions> configureOptions` delegate parameter of the `AddRedisDistributedCache` method. For example to set the connection timeout:

C#

```
builder.AddRedisDistributedCache(
    "cache",
     null,
     static options => options.ConnectTimeout = 3_000);
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#

- Health checks in ASP.NET Core

The .NET Aspire Redis distributed caching integration handles the following:

- Adds the health check when StackExchangeRedisSettings.DisableHealthChecks is `false`, which attempts to connect to the container instance.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not

- .NET Aspire integrations
- .NET Aspire GitHub repo

> *: *Redis is a registered trademark of Redis Ltd. Any rights therein are reserved to Redis Ltd. Any use by Microsoft is for referential purposes only and does not indicate any sponsorship, endorsement or affiliation between Redis and Microsoft.* *Return to top*?

.NET Aspire Redis® output caching integration

Article • 02/11/2025

Includes: Hosting integration and Client integration

Learn how to use the .NET Aspire Redis output caching integration. The `Aspire.StackExchange.Redis.OutputCaching` client integration is used to register an ASP.NET Core Output Caching provider backed by a Garnet server with the ghcr.io/microsoft/garnet container image.

Hosting integration

The Garnet hosting integration models a Garnet resource as the GarnetResource type. To access this type and APIs that allow you to add it to your Aspire.Hosting.Garnet NuGet package in the app host project.

.NET CLI

```
.NET CLI
dotnet add package Aspire.Hosting.Garnet
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Garnet resource

In your app host project, call AddGarnet on the `builder` instance to add a Garnet resource:

```
C#
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddGarnet("cache");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);
```

```
// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `ghcr.io/microsoft/garnet` image, it creates a new Garnet instance on your local machine. A reference to your Garnet resource (the `cache` variable) is added to the `ExampleProject`.

The WithReference method configures a connection in the `ExampleProject` named `"cache"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing Garnet instance, call **AddConnectionString** instead. For more information, see **Reference existing resources**.

Add Garnet resource with data volume

To add a data volume to the Garnet resource, call the AddGarnet method on the Garnet resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddGarnet("cache")
                   .WithDataVolume(isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The data volume is used to persist the Garnet data outside the lifecycle of its container. The data volume is mounted at the `/data` path in the Garnet container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add Garnet resource with data bind mount

To add a data bind mount to the Garnet resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddGarnet("cache")
                   .WithDataBindMount(
                       source: @"C:\Garnet\Data",
                       isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Garnet data across container restarts. The data bind mount is mounted at the `C:\Garnet\Data` on Windows (or `/Garnet/Data` on Unix) path on the host machine in the Garnet container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add Garnet resource with persistence

To add persistence to the Garnet resource, call the WithPersistence method with either the data volume or data bind mount:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddGarnet("cache")
                   .WithDataVolume()
                   .WithPersistence(
                       interval: TimeSpan.FromMinutes(5),
                       keysChangedThreshold: 100);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);
```

```
// After adding all resources, run the app...
```

The preceding code adds persistence to the Redis resource by taking snapshots of the Garnet data at a specified interval and threshold. The `interval` is time between snapshot exports and the `keysChangedThreshold` is the number of key change operations required to trigger a snapshot. For more information on persistence, see Redis docs: Persistence.

Hosting integration health checks

The Redis hosting integration automatically adds a health check for the appropriate resource type. The health check verifies that the server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Redis NuGet package.

Client integration

To get started with the .NET Aspire Stack Exchange Redis output caching client integration, install the Aspire.StackExchange.Redis.OutputCaching NuGet package in the client-consuming project, that is, the project for the application that uses the output caching client. The Redis output caching client integration registers services required for enabling CacheOutput method calls and [OutputCache] attribute usage to rely on Redis as its caching mechanism.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.OutputCaching
```

Add output caching

In the *Program.cs* file of your client-consuming project, call the AddRedisOutputCache extension method on any IHostApplicationBuilder to register the required services for output caching.

C#

```
builder.AddRedisOutputCache(connectionName: "cache");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Garnet resource in the app host project. For more information, see Add Garnet resource.

Add the middleware to the request processing pipeline by calling UseOutputCache(IApplicationBuilder):

C#

```
var app = builder.Build();

app.UseOutputCache();
```

For minimal API apps, configure an endpoint to do caching by calling CacheOutput, or by applying the OutputCacheAttribute, as shown in the following examples:

C#

```
app.MapGet("/cached", () => "Hello world!")
   .CacheOutput();

app.MapGet(
    "/attribute",
    [OutputCache] () => "Hello world!");
```

For apps with controllers, apply the `[OutputCache]` attribute to the action method. For Razor Pages apps, apply the attribute to the Razor page class.

Configuration

The .NET Aspire Stack Exchange Redis output caching integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling AddRedisOutputCache:

C#

```
builder.AddRedisOutputCache(connectionName: "cache");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the Stack Exchange Redis configuration docs.

Use configuration providers

The .NET Aspire Stack Exchange Redis output caching integration supports Microsoft.Extensions.Configuration. It loads the StackExchangeRedisSettings from configuration by using the `Aspire:StackExchange:Redis` key. Example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        },
        "DisableHealthChecks": true,
        "DisableTracing": false
      }
    }
  }
}
```

For the complete Redis output caching client integration JSON schema, see Aspire.StackExchange.Redis.OutputCaching/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings> configurationSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddRedisOutputCache(
    "cache",
    static settings => settings.DisableHealthChecks  = true);
```

You can also set up the ConfigurationOptions using the `Action<ConfigurationOptions> configureOptions` delegate parameter of the AddRedisOutputCache method. For example to set the connection timeout:

C#

```
builder.AddRedisOutputCache(
    "cache",
    static settings => settings.ConnectTimeout = 3_000);
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

The .NET Aspire Stack Exchange Redis output caching integration handles the following:

- Adds the health check when StackExchangeRedisSettings.DisableHealthChecks is `false`, which attempts to connect to the container instance.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations

overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Stack Exchange Redis output caching integration uses the following Log categories:

- `Aspire.StackExchange.Redis`
- `Microsoft.AspNetCore.OutputCaching.StackExchangeRedis`

Tracing

The .NET Aspire Stack Exchange Redis output caching integration will emit the following Tracing activities using OpenTelemetry:

- `OpenTelemetry.Instrumentation.StackExchangeRedis`

Metrics

The .NET Aspire Stack Exchange Redis output caching integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- Stack Exchange Redis docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

> *: *Redis is a registered trademark of Redis Ltd. Any rights therein are reserved to Redis Ltd. Any use by Microsoft is for referential purposes only and does not indicate any sponsorship, endorsement or affiliation between Redis and Microsoft.* *Return to top*?

.NET Aspire Redis®* integration

Article • 02/11/2025

Includes: ✅ Hosting integration and ✅ Client integration

Valkey is a Redis fork and complies with the Redis serialization protocol (RESP). It's a high-performance key/value datastore that supports a variety of workloads such as caching, message queues, and can act as a primary database. The .NET Aspire Redis integration enables you to connect to existing Valkey instances, or create new instances from .NET with the docker.io/valkey/valkey container image.

Hosting integration

The Valkey hosting integration models a Valkey resource as the ValkeyResource type. To access this type and APIs that allow you to add it to your Aspire.Hosting.Valkey NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Valkey
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Valkey resource

In your app host project, call AddValkey on the `builder` instance to add a Valkey resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddValkey("cache");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/valkey/valkey` image, it creates a new Valkey instance on your local machine. A reference to your Valkey resource (the `cache` variable) is added to the `ExampleProject`.

The WithReference method configures a connection in the `ExampleProject` named `"cache"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing Valkey instance, call **AddConnectionString** instead. For more information, see **Reference existing resources**.

Add Valkey resource with data volume

To add a data volume to the Valkey resource, call the AddValkey method on the Valkey resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddValkey("cache")
                   .WithDataVolume(isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The data volume is used to persist the Valkey data outside the lifecycle of its container. The data volume is mounted at the `/data` path in the Valkey container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add Valkey resource with data bind mount

To add a data bind mount to the Valkey resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddValkey("cache")
                   .WithDataBindMount(
                       source: @"C:\Valkey\Data",
                       isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Valkey data across container restarts. The data bind mount is mounted at the `C:\Valkey\Data` on Windows (or `/Valkey/Data` on Unix) path on the host machine in the Valkey container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add Valkey resource with persistence

To add persistence to the Valkey resource, call the WithPersistence method with either the data volume or data bind mount:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddValkey("cache")
                   .WithDataVolume()
                   .WithPersistence(
                       interval: TimeSpan.FromMinutes(5),
                       keysChangedThreshold: 100);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The preceding code adds persistence to the Redis resource by taking snapshots of the Valkey data at a specified interval and threshold. The `interval` is time between snapshot exports and the `keysChangedThreshold` is the number of key change operations required to trigger a snapshot. For more information on persistence, see Redis docs: Persistence.

Hosting integration health checks

The Redis hosting integration automatically adds a health check for the appropriate resource type. The health check verifies that the server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Redis NuGet package.

Client integration

To get started with the .NET Aspire Stack Exchange Redis client integration, install the Aspire.StackExchange.Redis NuGet package in the client-consuming project, that is, the project for the application that uses the Redis client. The Redis client integration registers an IConnectionMultiplexer instance that you can use to interact with Redis.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis
```

Add Redis client

In the *Program.cs* file of your client-consuming project, call the AddRedisClient extension method on any IHostApplicationBuilder to register an `IConnectionMultiplexer` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddRedisClient(connectionName: "cache");
```

You can then retrieve the `IConnection` instance using dependency injection. For example, to retrieve the connection from an example service:

```
C#
```

For more information on dependency injection, see .NET dependency injection.

There might be situations where you want to register multiple `IConnectionMultiplexer`

```

```

```

```

The .NET Aspire Stack Exchange Redis client integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling AddValkey:

C#

```
builder.AddValkey("cache");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the Stack Exchange Redis configuration docs.

Use configuration providers

The .NET Aspire Stack Exchange Redis integration supports Microsoft.Extensions.Configuration. It loads the StackExchangeRedisSettings from configuration by using the `Aspire:StackExchange:Redis` key. Example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        },
        "DisableHealthChecks": true,
```

```
        "DisableTracing": false
      }
    }
  }
}
```

For the complete Redis client integration JSON schema, see Aspire.StackExchange.Redis/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings>` delegate to set up some or all the options inline, for example to configure `DisableTracing`:

C#

```
builder.AddRedisClient(
    "cache",
    static settings => settings.DisableTracing = true);
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

The .NET Aspire Stack Exchange Redis integration handles the following:

- Adds the health check when StackExchangeRedisSettings.DisableHealthChecks is `false`, which attempts to connect to the container instance.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not

metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Stack Exchange Redis integration uses the following log categories:

- `Aspire.StackExchange.Redis`

Tracing

The .NET Aspire Stack Exchange Redis integration will emit the following tracing activities using OpenTelemetry:

- `OpenTelemetry.Instrumentation.StackExchangeRedis`

Metrics

The .NET Aspire Stack Exchange Redis integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- Stack Exchange Redis docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

> *: *Redis is a registered trademark of Redis Ltd. Any rights therein are reserved to Redis Ltd. Any use by Microsoft is for referential purposes only and does not indicate any sponsorship, endorsement or affiliation between Redis and Microsoft.* *Return to top*?

.NET Aspire Redis®* distributed caching integration

Article • 02/11/2025

Includes: ✅ Hosting integration and ✅ Client integration

Learn how to use the .NET Aspire Redis distributed caching integration. The `Aspire.StackExchange.Redis.DistributedCaching` library is used to register an IDistributedCache provider backed by a Valkey server with the docker.io/valkey/valkey container image.

Hosting integration

The Valkey hosting integration models a Valkey resource as the ValkeyResource type. To access this type and APIs that allow you to add it to your Aspire.Hosting.Valkey NuGet package in the app host project.

.NET CLI

```
.NET CLI
dotnet add package Aspire.Hosting.Valkey
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Valkey resource

In your app host project, call AddValkey on the `builder` instance to add a Valkey resource:

```
C#
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddValkey("cache");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);
```

```
// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/valkey/valkey` image, it creates a new Valkey instance on your local machine. A reference to your Valkey resource (the `cache` variable) is added to the `ExampleProject`.

The WithReference method configures a connection in the `ExampleProject` named `"cache"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing Valkey instance, call **AddConnectionString** instead. For more information, see **Reference existing resources**.

Add Valkey resource with data volume

To add a data volume to the Valkey resource, call the AddValkey method on the Valkey resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddValkey("cache")
                   .WithDataVolume(isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The data volume is used to persist the Valkey data outside the lifecycle of its container. The data volume is mounted at the `/data` path in the Valkey container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add Valkey resource with data bind mount

To add a data bind mount to the Valkey resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddValkey("cache")
                   .WithDataBindMount(
                       source: @"C:\Valkey\Data",
                       isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Valkey data across container restarts. The data bind mount is mounted at the `C:\Valkey\Data` on Windows (or `/Valkey/Data` on Unix) path on the host machine in the Valkey container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add Valkey resource with persistence

To add persistence to the Valkey resource, call the WithPersistence method with either the data volume or data bind mount:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddValkey("cache")
                   .WithDataVolume()
                   .WithPersistence(
                       interval: TimeSpan.FromMinutes(5),
                       keysChangedThreshold: 100);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);
```

```
// After adding all resources, run the app...
```

The preceding code adds persistence to the Redis resource by taking snapshots of the Valkey data at a specified interval and threshold. The `interval` is time between snapshot exports and the `keysChangedThreshold` is the number of key change operations required to trigger a snapshot. For more information on persistence, see Redis docs: Persistence.

Hosting integration health checks

The Redis hosting integration automatically adds a health check for the appropriate resource type. The health check verifies that the server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Redis NuGet package.

Client integration

To get started with the .NET Aspire Redis distributed caching integration, install the Aspire.StackExchange.Redis.DistributedCaching NuGet package in the client-consuming project, i.e., the project for the application that uses the Redis distributed caching client. The Redis client integration registers an IDistributedCache instance that you can use to interact with Redis.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.DistributedCaching
```

Add Redis client

In the *Program.cs* file of your client-consuming project, call the AddRedisDistributedCache extension to register the required services for distributed caching and add a IDistributedCache for use via the dependency injection container.

C#

```
builder.AddRedisDistributedCache(connectionName: "cache");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Valkey resource in the app host project. For more information, see Add Valkey resource.

You can then retrieve the `IDistributedCache` instance using dependency injection. For example, to retrieve the cache from a service:

C#

```
public class ExampleService(IDistributedCache cache)
{
    // Use cache...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed Redis client

There might be situations where you want to register multiple `IDistributedCache` instances with different connection names. To register keyed Redis clients, call the AddKeyedRedisDistributedCache method:

C#

```
builder.AddKeyedRedisDistributedCache(name: "chat");
builder.AddKeyedRedisDistributedCache(name: "product");
```

Then you can retrieve the `IDistributedCache` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("chat")] IDistributedCache chatCache,
    [FromKeyedServices("product")] IDistributedCache productCache)
{
    // Use caches...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire Redis distributed caching integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddRedisDistributedCache`:

C#

```
builder.AddRedisDistributedCache("cache");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the Stack Exchange Redis configuration docs.

Use configuration providers

The .NET Aspire Stack Exchange Redis distributed caching integration supports Microsoft.Extensions.Configuration. It loads the StackExchangeRedisSettings from configuration by using the `Aspire:StackExchange:Redis` key. Example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
```

```
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        },
        "DisableHealthChecks": true,
        "DisableTracing": false
      }
    }
  }
}
```

For the complete Redis distributed caching client integration JSON schema, see Aspire.StackExchange.Redis.DistributedCaching/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings>` delegate to set up some or all the options inline, for example to configure `DisableTracing`:

C#

```
builder.AddRedisDistributedCache(
    "cache",
    settings => settings.DisableTracing = true);
```

You can also set up the ConfigurationOptions using the `Action<ConfigurationOptions> configureOptions` delegate parameter of the `AddRedisDistributedCache` method. For example to set the connection timeout:

C#

```
builder.AddRedisDistributedCache(
    "cache",
     null,
     static options => options.ConnectTimeout = 3_000);
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#

- Health checks in ASP.NET Core

The .NET Aspire Redis distributed caching integration handles the following:

- Adds the health check when StackExchangeRedisSettings.DisableHealthChecks is `false`, which attempts to connect to the container instance.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not

- .NET Aspire integrations
- .NET Aspire GitHub repo

> *: *Redis is a registered trademark of Redis Ltd. Any rights therein are reserved to Redis Ltd. Any use by Microsoft is for referential purposes only and does not indicate any sponsorship, endorsement or affiliation between Redis and Microsoft.* *Return to top*?

.NET Aspire Redis® * output caching integration

Article • 02/11/2025

Includes: ✅ Hosting integration and ✅ Client integration

Learn how to use the .NET Aspire Redis output caching integration. The `Aspire.StackExchange.Redis.OutputCaching` client integration is used to register an ASP.NET Core Output Caching provider backed by a Valkey server with the docker.io/valkey/valkey container image.

Hosting integration

The Valkey hosting integration models a Valkey resource as the ValkeyResource type. To access this type and APIs that allow you to add it to your Aspire.Hosting.Valkey NuGet package in the app host project.

.NET CLI

```
.NET CLI
dotnet add package Aspire.Hosting.Valkey
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Valkey resource

In your app host project, call AddValkey on the `builder` instance to add a Valkey resource:

```
C#
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddValkey("cache");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);
```

```
// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/valkey/valkey` image, it creates a new Valkey instance on your local machine. A reference to your Valkey resource (the `cache` variable) is added to the `ExampleProject`.

The WithReference method configures a connection in the `ExampleProject` named `"cache"`. For more information, see Container resource lifecycle.

> **Tip**
>
> If you'd rather connect to an existing Valkey instance, call **AddConnectionString** instead. For more information, see **Reference existing resources**.

Add Valkey resource with data volume

To add a data volume to the Valkey resource, call the AddValkey method on the Valkey resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddValkey("cache")
                   .WithDataVolume(isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

The data volume is used to persist the Valkey data outside the lifecycle of its container. The data volume is mounted at the `/data` path in the Valkey container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add Valkey resource with data bind mount

To add a data bind mount to the Valkey resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddValkey("cache")
                   .WithDataBindMount(
                       source: @"C:\Valkey\Data",
                       isReadOnly: false);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Valkey data across container restarts. The data bind mount is mounted at the `C:\Valkey\Data` on Windows (or `/Valkey/Data` on Unix) path on the host machine in the Valkey container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add Valkey resource with persistence

To add persistence to the Valkey resource, call the WithPersistence method with either the data volume or data bind mount:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddValkey("cache")
                   .WithDataVolume()
                   .WithPersistence(
                       interval: TimeSpan.FromMinutes(5),
                       keysChangedThreshold: 100);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(cache);
```

```
// After adding all resources, run the app...
```

The preceding code adds persistence to the Redis resource by taking snapshots of the Valkey data at a specified interval and threshold. The `interval` is time between snapshot exports and the `keysChangedThreshold` is the number of key change operations required to trigger a snapshot. For more information on persistence, see Redis docs: Persistence.

Hosting integration health checks

The Redis hosting integration automatically adds a health check for the appropriate resource type. The health check verifies that the server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.Redis NuGet package.

Client integration

To get started with the .NET Aspire Stack Exchange Redis output caching client integration, install the Aspire.StackExchange.Redis.OutputCaching NuGet package in the client-consuming project, that is, the project for the application that uses the output caching client. The Redis output caching client integration registers services required for enabling CacheOutput method calls and [OutputCache] attribute usage to rely on Redis as its caching mechanism.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.OutputCaching
```

Add output caching

In the *Program.cs* file of your client-consuming project, call the AddRedisOutputCache extension method on any IHostApplicationBuilder to register the required services for output caching.

C#

```
builder.AddRedisOutputCache(connectionName: "cache");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Valkey resource in the app host project. For more information, see Add Valkey resource.

Add the middleware to the request processing pipeline by calling UseOutputCache(IApplicationBuilder):

C#

```
var app = builder.Build();

app.UseOutputCache();
```

For minimal API apps, configure an endpoint to do caching by calling CacheOutput, or by applying the OutputCacheAttribute, as shown in the following examples:

C#

```
app.MapGet("/cached", () => "Hello world!")
   .CacheOutput();

app.MapGet(
    "/attribute",
    [OutputCache] () => "Hello world!");
```

For apps with controllers, apply the `[OutputCache]` attribute to the action method. For Razor Pages apps, apply the attribute to the Razor page class.

Configuration

The .NET Aspire Stack Exchange Redis output caching integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling AddRedisOutputCache:

C#

```
builder.AddRedisOutputCache(connectionName: "cache");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the Stack Exchange Redis configuration docs.

Use configuration providers

The .NET Aspire Stack Exchange Redis output caching integration supports Microsoft.Extensions.Configuration. It loads the StackExchangeRedisSettings from configuration by using the `Aspire:StackExchange:Redis` key. Example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        },
        "DisableHealthChecks": true,
        "DisableTracing": false
      }
    }
  }
}
```

For the complete Redis output caching client integration JSON schema, see Aspire.StackExchange.Redis.OutputCaching/ConfigurationSchema.json.

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings> configurationSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddRedisOutputCache(
    "cache",
    static settings => settings.DisableHealthChecks  = true);
```

You can also set up the ConfigurationOptions using the `Action<ConfigurationOptions> configureOptions` delegate parameter of the AddRedisOutputCache method. For example to set the connection timeout:

C#

```
builder.AddRedisOutputCache(
    "cache",
    static settings => settings.ConnectTimeout = 3_000);
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

The .NET Aspire Stack Exchange Redis output caching integration handles the following:

- Adds the health check when StackExchangeRedisSettings.DisableHealthChecks is `false`, which attempts to connect to the container instance.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations

overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Stack Exchange Redis output caching integration uses the following Log categories:

- `Aspire.StackExchange.Redis`
- `Microsoft.AspNetCore.OutputCaching.StackExchangeRedis`

Tracing

The .NET Aspire Stack Exchange Redis output caching integration will emit the following Tracing activities using OpenTelemetry:

- `OpenTelemetry.Instrumentation.StackExchangeRedis`

Metrics

The .NET Aspire Stack Exchange Redis output caching integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- Stack Exchange Redis docs
- .NET Aspire integrations
- .NET Aspire GitHub repo

> *: *Redis is a registered trademark of Redis Ltd. Any rights therein are reserved to Redis Ltd. Any use by Microsoft is for referential purposes only and does not indicate any sponsorship, endorsement or affiliation between Redis and Microsoft.* *Return to top*?

.NET Aspire Seq integration

Article • 02/11/2025

Includes: ✅ Hosting integration and ✅ Client integration

Seq is a self-hosted search and analysis server that handles structured application logs and trace files. It includes a JSON event store and a simple query language that make it easy to use. You can use the .NET Aspire Seq integration to send OpenTelemetry Protocol (OTLP) data to Seq. The integration supports persistent logs and traces across application restarts.

During development, .NET Aspire runs and connects to the datalust/seq container image.

Hosting integration

The Seq hosting integration models the server as the SeqResource type. To access this type and the API, add the Aspire.Hosting.Seq NuGet package in the app host project.

.NET CLI

```
.NET CLI
dotnet add package Aspire.Hosting.Seq
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add a Seq resource

In your app host project, call AddSeq to add and return a Seq resource builder.

```
C#
var builder = DistributedApplication.CreateBuilder(args);

var seq = builder.AddSeq("seq")
                 .ExcludeFromManifest()
                 .WithLifetime(ContainerLifetime.Persistent)
                 .WithEnvironment("ACCEPT_EULA"  "Y"
```

```
var myService = builder.AddProject<Projects.ExampleProject>()
                       .WithReference(seq)
                       .WaitFor(seq);

// After adding all resources, run the app...
```

> **Note**
>
> The Seq container may be slow to start, so it's best to use a *persistent* lifetime to avoid unnecessary restarts. For more information, see Container resource lifetime.

Accept the Seq End User License Agreement (EULA)

You must accept the Seq EULA for Seq to start. To accept the agreement in code, pass the environment variable `ACCEPT_EULA` to the Seq container, and set its value to `Y`. The above code passes this variable in the chained call to WithEnvironment.

Seq in the .NET Aspire manifest

Seq shouldn't be part of the .NET Aspire deployment manifest, hence the chained call to ExcludeFromManifest. It's recommended you set up a secure production Seq server outside of .NET Aspire for your production environment.

Persistent logs and traces

Register Seq with a data directory in your app host project to retain Seq's data and configuration across application restarts:

C#

```
var seq = builder.AddSeq("seq", seqDataDirectory: "./seqdata")
                 .ExcludeFromManifest()
                 .WithLifetime(ContainerLifetime.Persistent);
```

The directory specified must already exist.

Add a Seq resource with a data volume

To add a data volume to the Seq resource, call the WithDataVolume method on the Seq resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var seq = builder.AddSeq("seq")
                 .WithDataVolume()
                 .ExcludeFromManifest()
                 .WithLifetime(ContainerLifetime.Persistent);

var myService = builder.AddProject<Projects.ExampleProject>()
                       .WithReference(seq)
                       .WaitFor(seq);
```

The data volume is used to persist the Seq data outside the lifecycle of its container. The data volume is mounted at the `/data` path in the Seq container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add Seq resource with data bind mount

To add a data bind mount to the Seq resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var seq = builder.AddSeq("seq")
                 .WithDataBindMount(source: @"C:\Data")
                 .ExcludeFromManifest()
                 .WithLifetime(ContainerLifetime.Persistent);

var myService = builder.AddProject<Projects.ExampleProject>()
                       .WithReference(seq)
                       .WaitFor(seq);
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Seq data across container restarts. The data bind mount is mounted at the `C:\Data` on Windows (or `/Data` on Unix) path on the host machine in the Seq container. For more information on data bind mounts, see Docker docs: Bind mounts.

Client integration

To get started with the .NET Aspire Seq client integration, install the Aspire.Seq NuGet package in the client-consuming project, that is, the project for the application that uses the Seq client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Seq
```

Add a Seq client

In the *Program.cs* file of your client-consuming project, call the AddSeqEndpoint extension method to register OpenTelemetry Protocol exporters to send logs and traces to Seq and the .NET Aspire Dashboard. The method takes a connection name parameter.

C#

```
builder.AddSeqEndpoint(connectionName: "seq");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Seq resource in the app host project. In other words, when you call `AddSeq` and provide a name of `seq` that same name should be used when calling `AddSeqEndpoint`. For more information, see Add a Seq resource.

Configuration

The .NET Aspire Seq integration provides multiple options to configure the connection to Seq based on the requirements and conventions of your project.

Use configuration providers

The .NET Aspire Seq integration supports Microsoft.Extensions.Configuration. It loads the SeqSettings from configuration by using the `Aspire:Seq` key. The following snippet is an example of an *appsettings.json* file that configures some of the options:

JSON

```
{
  "Aspire": {
    "Seq": {
      "DisableHealthChecks": true,
      "ServerUrl": "http://localhost:5341"
    }
  }
}
```

For the complete Seq client integration JSON schema, see Aspire.Seq/ConfigurationSchema.json.

Use inline delegates

Also you can pass the `Action<SeqSettings> configureSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddSeqEndpoint("seq", static settings =>
{
    settings.DisableHealthChecks  = true;
    settings.ServerUrl = "http://localhost:5341"
});
```

Client integration health checks

By default, .NET Aspire *client integrations* have health checks enabled for all services. Similarly, many .NET Aspire *hosting integrations* also enable health check endpoints. For more information, see:

- .NET app health checks in C#
- Health checks in ASP.NET Core

The .NET Aspire Seq integration handles the following:

- Adds the health check when SeqSettings.DisableHealthChecks is `false`, which attempts to connect to the Seq server's `/health` endpoint.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire Seq integration uses the following log categories:

- `Seq`

Tracing and Metrics

The .NET Aspire Seq integration doesn't emit tracing activities and or metrics because it's a telemetry sink, not a telemetry source.

See also

- Seq
- Seq Query Language
- .NET Aspire integrations
- .NET Aspire GitHub repo

.NET Aspire SQL Server integration

Article • 02/11/2025

Includes: ✅ Hosting integration and ✅ Client integration

SQL Server is a relational database management system developed by Microsoft. The .NET Aspire SQL Server integration enables you to connect to existing SQL Server instances or create new instances from .NET with the mcr.microsoft.com/mssql/server container image.

Hosting integration

The SQL Server hosting integration models the server as the SqlServerServerResource type and the database as the SqlServerDatabaseResource type. To access these types and APIs, add the Aspire.Hosting.SqlServer NuGet package in the app host project.

.NET CLI

```
.NET CLI

dotnet add package Aspire.Hosting.SqlServer
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add SQL Server resource and database resource

In your app host project, call AddSqlServer to add and return a SQL Server resource builder. Chain a call to the returned resource builder to AddDatabase, to add SQL Server database resource.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var sql = builder.AddSqlServer("sql")
                 .WithLifetime(ContainerLifetime.Persistent);

var db = sql.AddDatabase("database");

builder.AddProject<Projects.ExampleProject>()
```

```
        .WithReference(db)
        .WaitFor(db);

// After adding all resources, run the app...
```

> **Note**
>
> The SQL Server container is slow to start, so it's best to use a *persistent* lifetime to avoid unnecessary restarts. For more information, see Container resource lifetime.

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `mcr.microsoft.com/mssql/server` image, it creates a new SQL Server instance on your local machine. A reference to your SQL Server resource builder (the `sql` variable) is used to add a database. The database is named `database` and then added to the `ExampleProject`. The SQL Server resource includes default credentials with a `username` of `sa` and a random `password` generated using the CreateDefaultPasswordParameter method.

When the app host runs, the password is stored in the app host's secret store. It's added to the `Parameters` section, for example:

JSON

```
{
  "Parameters:sql-password": "<THE_GENERATED_PASSWORD>"
}
```

The name of the parameter is `sql-password`, but really it's just formatting the resource name with a `-password` suffix. For more information, see Safe storage of app secrets in development in ASP.NET Core and Add SQL Server resource with parameters.

The WithReference method configures a connection in the `ExampleProject` named `database`.

> **Tip**
>
> If you'd rather connect to an existing SQL Server, call AddConnectionString instead. For more information, see Reference existing resources.

Add SQL Server resource with data volume

To add a data volume to the SQL Server resource, call the WithDataVolume method on the SQL Server resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var sql = builder.AddSqlServer("sql")
                 .WithDataVolume();

var db = sql.AddDatabase("database");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(db)
       .WaitFor(db);

// After adding all resources, run the app...
```

The data volume is used to persist the SQL Server data outside the lifecycle of its container. The data volume is mounted at the `/var/opt/mssql` path in the SQL Server container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

> ⚠ **Warning**
>
> The password is stored in the data volume. When using a data volume and if the password changes, it will not work until you delete the volume.

Add SQL Server resource with data bind mount

To add a data bind mount to the SQL Server resource, call the WithDataBindMount method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var sql = builder.AddSqlServer("sql")
                 .WithDataBindMount(source: @"C:\SqlServer\Data");

var db = sql.AddDatabase("database");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(db)
       .WaitFor(db);
```

```
// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the SQL Server data across container restarts. The data bind mount is mounted at the `C:\SqlServer\Data` on Windows (or `/SqlServer/Data` on Unix) path on the host machine in the SQL Server container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add SQL Server resource with parameters

When you want to explicitly provide the password used by the container image, you can provide these credentials as parameters. Consider the following alternative example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var password = builder.AddParameter("password", secret: true);

var sql = builder.AddSqlServer("sql", password);
var db = sql.AddDatabase("database");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(db)
       .WaitFor(db);

// After adding all resources, run the app...
```

For more information on providing parameters, see External parameters.

Connect to database resources

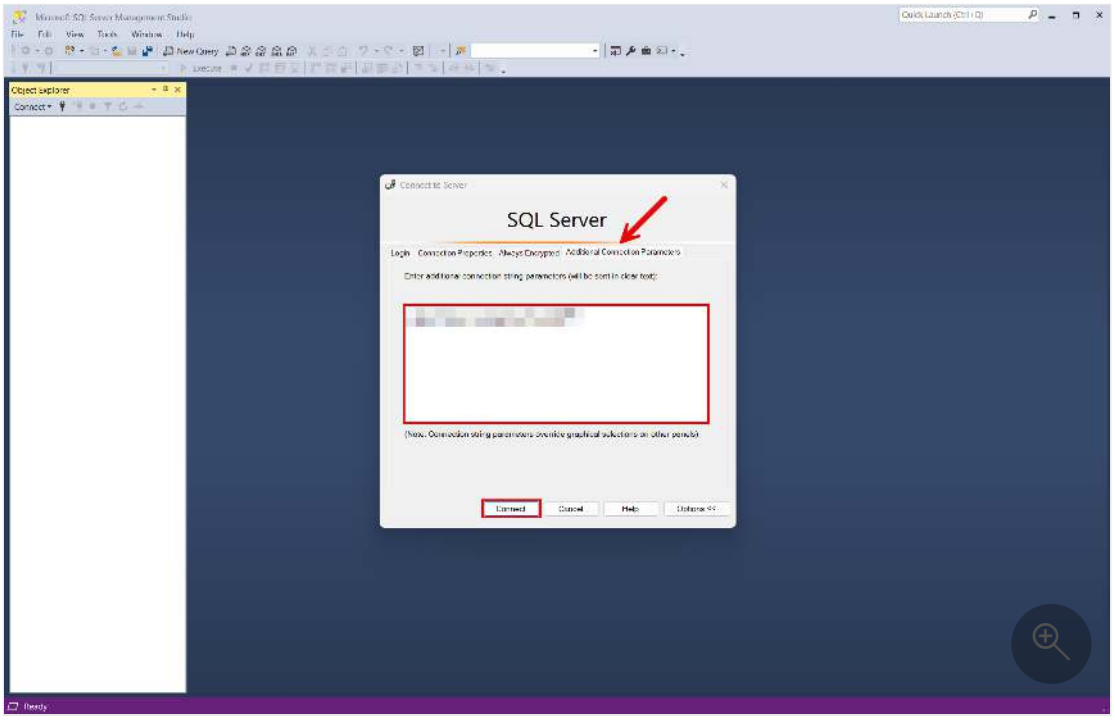
When the .NET Aspire app host runs, the server's database resources can be accessed from external tools, such as SQL Server Management Studio (SSMS) or MSSQL for Visual Studio Code. The connection string for the database resource is available in the

dependent resources environment variables and is accessed using the .NET Aspire dashboard: Resource details pane. The environment variable is named `ConnectionStrings__{name}` where `{name}` is the name of the database resource, in this example it's `database`. Use the connection string to connect to the database resource from external tools. Imagine that you have a database named `todos` with a single `dbo.Todos` table.

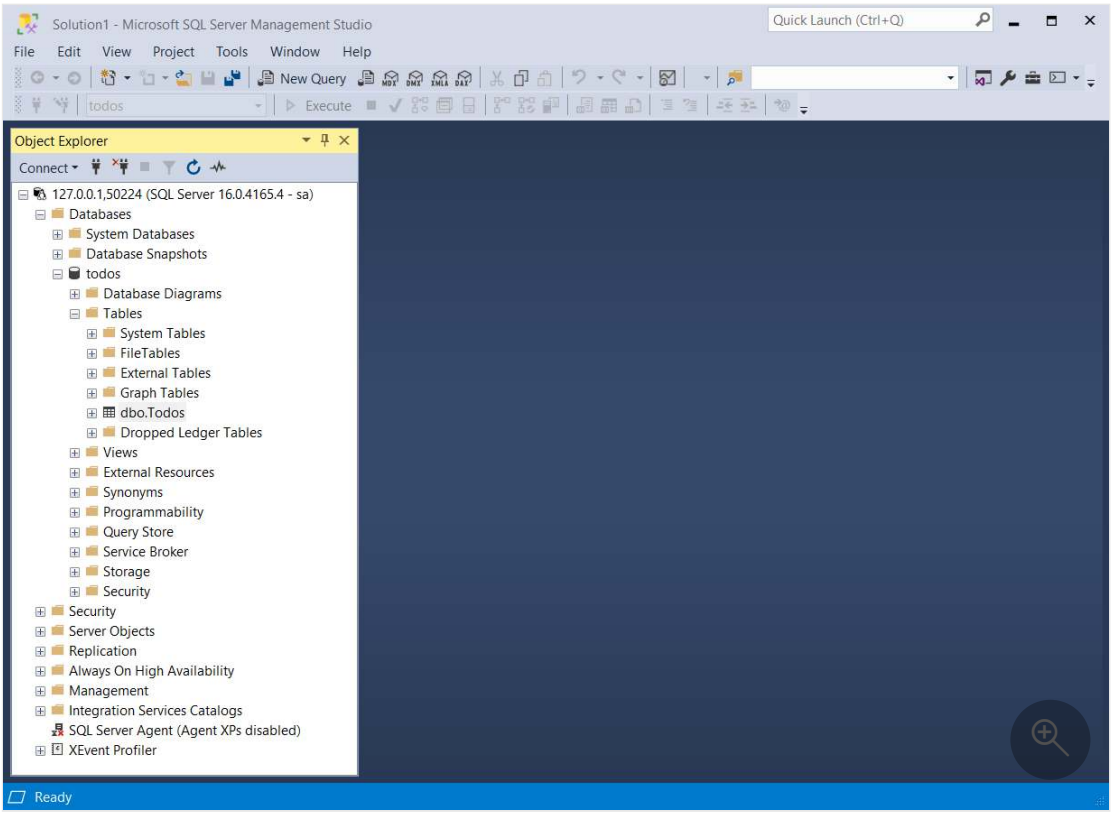
SQL Server Management Studio

To connect to the database resource from SQL Server Management Studio, follow these steps:

1. Open SSMS.
2. In the **Connect to Server** dialog, select the **Additional Connection Parameters** tab.
3. Paste the connection string into the **Additional Connection Parameters** field and select **Connect**.



4. If you're connected, you can see the database resource in the **Object Explorer**:



For more information, see SQL Server Management Studio: Connect to a server.

Hosting integration health checks

The SQL Server hosting integration automatically adds a health check for the SQL Server resource. The health check verifies that the SQL Server is running and that a connection can be established to it.

The hosting integration relies on the AspNetCore.HealthChecks.SqlServer NuGet package.

Client integration

To get started with the .NET Aspire SQL Server client integration, install the Aspire.Microsoft.Data.SqlClient NuGet package in the client-consuming project, that is, the project for the application that uses the SQL Server client. The SQL Server client integration registers a SqlConnection instance that you can use to interact with SQL Server.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Microsoft.Data.SqlClient
```

Add SQL Server client

In the *Program.cs* file of your client-consuming project, call the AddSqlServerClient extension method on any IHostApplicationBuilder to register a `SqlConnection` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddSqlServerClient(connectionName: "database");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the SQL Server database resource in the app host project. In other words, when you call `AddDatabase` and provide a name of `database` that same name should be used when calling `AddSqlServerClient`. For more information, see **Add SQL Server resource and database resource**.

You can then retrieve the SqlConnection instance using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(SqlConnection connection)
{
    // Use connection...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed SQL Server client

There might be situations where you want to register multiple `SqlConnection` instances with different connection names. To register keyed SQL Server clients, call the AddKeyedSqlServerClient method:

C#

```
builder.AddKeyedSqlServerClient(name: "mainDb");
builder.AddKeyedSqlServerClient(name: "loggingDb");
```

> **ⓘ Important**
>
> When using keyed services, it's expected that your SQL Server resource configured two named databases, one for the `mainDb` and one for the `loggingDb`.

Then you can retrieve the `SqlConnection` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("mainDb")] SqlConnection mainDbConnection,
    [FromKeyedServices("loggingDb")] SqlConnection loggingDbConnection)
{
    // Use connections...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire SQL Server integration provides multiple options to configure the connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling the AddSqlServerClient method:

C#

```
builder.AddSqlServerClient(connectionName: "sql");
```

Then the connection string is retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "database": "Data Source=myserver;Initial Catalog=master"
  }
}
```

For more information on how to format this connection string, see the ConnectionString.

Use configuration providers

The .NET Aspire SQL Server integration supports Microsoft.Extensions.Configuration. It loads the MicrosoftDataSqlClientSettings from configuration by using the `Aspire:Microsoft:Data:SqlClient` key. The following snippet is an example of a *appsettings.json* file that configures some of the options:

JSON

```
{
  "Aspire": {
    "Microsoft": {
      "Data": {
        "SqlClient": {
          "ConnectionString": "YOUR_CONNECTIONSTRING",
          "DisableHealthChecks": false,
          "DisableMetrics": true
        }
      }
    }
  }
}
```

For the complete SQL Server client integration JSON schema, see Aspire.Microsoft.Data.SqlClient/ConfigurationSchema.json.

Use inline delegates

Also you can pass the `Action<MicrosoftDataSqlClientSettings> configureSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddSqlServerClient(
    "database",
    static settings => settings.DisableHealthChecks = true);
```

Client integration health checks

By default, .NET Aspire integrations enable health checks for all services. For more information, see .NET Aspire integrations overview.

The .NET Aspire SQL Server integration:

- Adds the health check when MicrosoftDataSqlClientSettings.DisableHealthChecks is `false`, which attempts to connect to the SQL Server.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see .NET Aspire integrations overview. Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the Configuration section.

Logging

The .NET Aspire SQL Server integration currently doesn't enable logging by default due to limitations of the Microsoft.Data.SqlClient.

Tracing

The .NET Aspire SQL Server integration emits the following tracing activities using OpenTelemetry:

- `OpenTelemetry.Instrumentation.SqlClient`

Metrics

The .NET Aspire SQL Server integration will emit the following metrics using OpenTelemetry:

- Microsoft.Data.SqlClient.EventSource
  - `active-hard-connections`
  - `hard-connects`
  - `hard-disconnects`
  - `active-soft-connects`
  - `soft-connects`
  - `soft-disconnects`
  - `number-of-non-pooled-connections`
  - `number-of-pooled-connections`
  - `number-of-active-connection-pool-groups`
  - `number-of-inactive-connection-pool-groups`
  - `number-of-active-connection-pools`
  - `number-of-inactive-connection-pools`
  - `number-of-active-connections`
  - `number-of-free-connections`
  - `number-of-stasis-connections`
  - `number-of-reclaimed-connections`

See also

- Azure SQL Database
- SQL Server
- .NET Aspire database containers sample
- .NET Aspire integrations
- .NET Aspire GitHub repo
- Azure SQL & SQL Server Aspire Samples

.NET Aspire Community Toolkit

Article • 03/05/2025

The .NET Aspire Community Toolkit is part of the .NET Foundation. The community toolkit is a collection of integrations and extensions for .NET Aspire created by the community. The .NET Aspire team doesn't officially support the integrations and extensions in the community toolkit. The community provides these tools as-is for everyone to use and contribute to. You can find the source code for the toolkit on GitHub.

Why use the toolkit?

The community toolkit offers flexible, community-driven integrations that enhance the .NET Aspire ecosystem. By contributing, you help shape tools that make building cloud-native applications easier and more versatile.

What's in the toolkit?

The community toolkit is a growing project, publishing a set of NuGet packages. It aims to provide various integrations, both hosting and client alike, that aren't otherwise part of the official .NET Aspire project. Additionally, the community toolkit packages various extensions for popular services and platforms. The following sections detail some of the integrations and extensions currently available in the toolkit.

Hosting integrations

- The Azure Static Web Apps integration enables local emulator support:
  - .NET Aspire Azure Static Web Apps emulator integration.
  - CommunityToolkit.Aspire.Hosting.Azure.StaticWebApps.
- The Azure Data API Builder integration enables seamless API creation for your data:
  - .NET Aspire Azure Data API Builder integration.
  - CommunityToolkit.Aspire.Hosting.Azure.DataApiBuilder.
- The Bun integration provides support for hosting Bun applications:
  - .NET Aspire Bun hosting integration.
  - CommunityToolkit.Aspire.Hosting.Bun.
- The Golang apps integration provides support for hosting Go applications:
  - .NET Aspire Go integration.
  - CommunityToolkit.Aspire.Hosting.Golang.

- The Java integration runs Java code with a local Java Development Kit (JDK) or using a container:
  - .NET Aspire Java/Spring hosting integration.
  - CommunityToolkit.Aspire.Hosting.Java.
- The Deno integration provides support for hosting Deno applications and running tasks.
  - .NET Aspire Deno hosting integration.
  - CommunityToolkit.Aspire.Hosting.Deno.
- The Ollama integration provides extensions and resource definitions, and support for downloading models as startup.
  - .NET Aspire Ollama hosting integration.
  - CommunityToolkit.Aspire.Hosting.Ollama.
- The Meilisearch integration enables hosting Meilisearch containers.
  - .NET Aspire Meilisearch hosting integration.
  - CommunityToolkit.Aspire.Hosting.Meilisearch.
- The Rust apps integration provides support for hosting Rust applications.
  - .NET Aspire Rust hosting integration.
  - CommunityToolkit.Aspire.Hosting.Rust.
- The SQLite integration provides support for hosting SQLite databases.
  - .NET Aspire SQLite hosting integration
  - CommunityToolkit.Aspire.Hosting.SQLite.

Client integrations

The following client integrations are available in the toolkit:

- **OllamaSharp** is a .NET client for the Ollama API:
  - .NET Aspire Ollama client integration
  - CommunityToolkit.Aspire.OllamaSharp
- **Meilisearch** is a .NET client for the Meilisearch API:
  - .NET Aspire Meilisearch client integration
  - CommunityToolkit.Aspire.Meilisearch
- The SQLite integration provides support for hosting SQLite databases.
  - .NET Aspire SQLite hosting integration
  - CommunityToolkit.Aspire.Hosting.SQLite.
- The SQLite Entity Framework integration provides support for hosting SQLite databases with Entity Framework.
  - .NET Aspire SQLite EF hosting integration
  - CommunityToolkit.Aspire.Microsoft.EntityFrameworkCore.Sqlite.

> **Tip**

Always check the **GitHub repository** for the most up-to-date information on the toolkit.

Extensions

To expand the functionality provided by the .NET Aspire integrations, the Community Toolkit also provides extension packages for some hosting integrations. The following extensions are available in the toolkit:

- CommunityToolkit.Aspire.Hosting.NodeJS.Extensions
  - Docs
- CommunityToolkit.Aspire.Hosting.SqlServer.Extensions
  - Docs
- CommunityToolkit.Aspire.Hosting.PostgreSQL.Extensions
  - Docs
- CommunityToolkit.Aspire.Hosting.Redis.Extensions
  - Docs
- CommunityToolkit.Aspire.Hosting.MongoDB.Extensions
  - Docs

If you're not seeing an integration or extension you need, you can contribute to the toolkit by creating your own integration and submitting a pull request. For more information, see How to collaborate.

How to collaborate

The community toolkit is an open-source project, and contributions from the community aren't only welcomed, but encouraged. If you're interested in contributing, see the contributing guidelines. As part of the .NET Foundation, contributors of the toolkit must adhere to the .NET Foundation Code of Conduct.

.NET Aspire Azure Static Web Apps emulator integration

Article • 10/11/2024

Includes: Hosting integration not Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

In this article, you learn how to use the .NET Aspire Azure Static Web Apps emulator hosting integration to run Azure Static Web Apps locally using the emulator. The emulator provides support for proxying both the static frontend and the API backend using resources defined in the app host.

This integration requires the Azure Static Web Apps CLI to run, and only supports hosting the emulator for local development, not deploying to Azure Static Web Apps.

Hosting integration

To get started with the .NET Aspire Azure Static Web Apps emulator hosting integration, install the CommunityToolkit.Aspire.Hosting.Azure.StaticWebApps NuGet package in the AppHost project.

.NET CLI

```
.NET CLI
dotnet add package CommunityToolkit.Aspire.Hosting.Azure.StaticWebApps
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Example usage

In the *Program.cs* file of your app host project, define the backend and frontend resources. Then, call the `AddSwaEmulator` method to create the emulator and pass the

resources using the `WithAppResource` and `WithApiResource` methods.

```C#
var builder = DistributedApplication.CreateBuilder(args);

// Define the API resource
var api =
builder.AddProject<Projects.Aspire_CommunityToolkit_StaticWebApps_ApiApp>
("api");

// Define the frontend resource
var web = builder
    .AddNpmApp("web", Path.Combine("..",
"CommunityToolkit.Aspire.StaticWebApps.WebApp"), "dev")
    .WithHttpEndpoint(env: "PORT")
    .WithExternalHttpEndpoints();

// Create a SWA emulator with the frontend and API resources
_ = builder
    .AddSwaEmulator("swa")
    .WithAppResource(web)
    .WithApiResource(api);

builder.Build().Run();
```

The preceding code defines the API and frontend resources and creates an emulator with the resources. The emulator is then started using the `Run` method.

See also

- Azure Static Web Apps emulator
- Azure Static Web Apps
- .NET Aspire Community Toolkit GitHub repo
- Sample app source code

.NET Aspire Bun hosting

Article • 11/20/2024

Includes: ✅ Hosting integration not ⊗ Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

Bun is a modern, fast, and lightweight framework for building web applications with TypeScript. The .NET Aspire Bun hosting integration allows you to host Bun applications in your .NET Aspire app host project, and provide it to other resources in your application.

Hosting integration

The Bun hosting integration models a Bun application as the `Aspire.Hosting.ApplicationModel.BunAppResource` type. To access this type and APIs that allow you to add it to your app host project, install the 📦 CommunityToolkit.Aspire.Hosting.Bun NuGet package in the app host project.

This integration expects that the Bun executable has already been installed on the host machine, and that it's available in the system path.

.NET CLI

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.Hosting.Bun
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add a Bun resource

In your app host project, call the `Aspire.Hosting.BunAppExtensions.AddBunApp` on the `builder` instance to add a Bun application resource as shown in the following example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var api = builder.AddBunApp("api")
                 .WithHttpEndpoint(env: "PORT");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(api);

// After adding all resources, run the app...
```

By default the working directory of the application will be a sibling folder to the app host matching the name provided to the resource, and the entrypoint will be *:::no-loc text="index.ts":::*. Both of these can be customized by passing additional parameters to the `AddBunApp` method.

C#

```
var api = builder.AddBunApp("api", "../api-service", "start")
    .WithHttpEndpoint(env: "PORT");
```

The Bun application can be added as a reference to other resources in the app host project.

Ensuring packages are installed

To ensure that the Bun application has all the dependencies installed as defined in the lockfile, you can use the `Aspire.Hosting.BunAppExtensions.WithBunPackageInstaller` method to ensure that package installation is run before the application is started.

C#

```
var api = builder.AddBunApp("api")
                 .WithHttpEndpoint(env: "PORT")
                 .WithBunPackageInstaller();
```

See also

- .NET Aspire Community Toolkit GitHub repo
- Sample Bun app

.NET Aspire Community Toolkit Deno hosting integration

Article • 11/20/2024

Includes: ✅ Hosting integration not ⊗ Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

In this article, you learn about the .NET Aspire Community Toolkit Deno package. The extensions package brings the following features:

- Running Deno applications
- Running Node.js applications via Deno tasks
- Ensuring that the packages are installed before running the application via Deno installer

Hosting integration

To get started with the .NET Aspire Community Toolkit Deno extensions, install the CommunityToolkit.Aspire.Hosting.Deno NuGet package in the AppHost project.

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.Hosting.Deno
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Example usage

The following sections detail various usages, from running Vite applications to using specific package managers.

Run Deno apps

This integration extension adds support for running a Deno application defined in a script. Since Deno is secure by default, permission flags must be specified in `permissionFlags` argument of `AddDenoApp`.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddDenoApp("oak-demo", "main.ts", permissionFlags: ["--allow-env",
"--allow-net"])
    .WithHttpEndpoint(env: "PORT")
    .WithEndpoint();

builder.Build().Run();
```

The preceding code uses the fully qualified switches. Alternatively, you can use the equivalent alias as well. For more information, see Deno docs: Security and permissions.

Run Deno tasks

This integration extension adds support for running tasks that are either specified in a *package.json* or *deno.json*.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddDenoTask("vite-demo", taskName: "dev")
    .WithHttpEndpoint(env: "PORT")
    .WithEndpoint();

builder.Build().Run();
```

Deno package installation

This integration extension adds support for installing dependencies that utilizes `deno install` behind the scenes by simply using `WithDenoPackageInstallation`.

> ⓘ Note

This API only works when a *deno.lock* file present.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddDenoTask("vite-demo", taskName: "dev")
    .WithDenoPackageInstallation()
    .WithHttpEndpoint(env: "PORT")
    .WithEndpoint();
```

See also

- .NET Aspire Community Toolkit GitHub repo
- Sample Deno apps
- Deno Docs

.NET Aspire Go hosting

Article • 11/20/2024

Includes: ✅ Hosting integration not ⊗ Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

In this article, you learn how to use the .NET Aspire Go hosting integration to host Go applications.

Hosting integration

To get started with the .NET Aspire Go hosting integration, install the CommunityToolkit.Aspire.Hosting.Go NuGet package in the AppHost project.

.NET CLI

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.Hosting.Golang
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Example usage

In the *Program.cs* file of your app host project, call the `AddGolangApp` method to add a Go application to the builder.

C#

```
var golang = builder.AddGolangApp("golang", "../gin-api")
    .WithHttpEndpoint(env: "PORT");
```

The `PORT` environment variable is used to determine the port the Go application should listen on. By default, this port is randomly assigned by .NET Aspire. The name of the

environment variable can be changed by passing a different value to the WithHttpEndpoint method.

The Go application can be added as a reference to other resources in the AppHost project.

See also

- .NET Aspire Community Toolkit GitHub repo
- Sample Go app

.NET Aspire Java/Spring hosting integration

Article • 10/11/2024

Includes: ✅ Hosting integration not ⊗ Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

In this article, you learn how to use the .NET Aspire Java/Spring hosting integration to host Java/Spring applications using either the Java runtime or a container.

Prerequisites

This integration requires the OpenTelemetry Agent for Java to be downloaded and placed in the `agents` directory in the root of the project. Depending on your preferred shell, use either of the following commands to download the agent:

Bash

Bash

```bash
# bash/zsh
mkdir -p ./agents
wget -P ./agents \
    https://github.com/open-telemetry/opentelemetry-java-instrumentation/releases/latest/download/opentelemetry-javaagent.jar
```

Get started

To get started with the .NET Aspire Azure Static Web Apps emulator integration, install the CommunityToolkit.Aspire.Hosting.Java NuGet package in the AppHost project.

.NET CLI

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.Hosting.Java
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Example Usage

The following sections detail various example usage scenarios, from hosting a containerized Spring app to hosting an executable Spring app.

Container hosting

In the _Program.cs_file of your app host project, call the `AddSpringApp` method to define the containerized Spring app. Use the `JavaAppContainerResourceOptions` to define the containerized Spring app.

C#

```
var containerapp = builder.AddSpringApp(
    "containerapp",
    new JavaAppContainerResourceOptions
    {
        ContainerImageName = "<repository>/<image>",
        OtelAgentPath = "<agent-path>"
    });
```

See also

- Java developer resources
- .NET Aspire Community Toolkit GitHub repo

.NET Aspire Community Toolkit Node.js hosting extensions

Article • 10/11/2024

Includes: Hosting integration not

The following sections detail various usages, from running Vite applications to using specific package managers.

Run specific package managers

This integration extension adds support for running Node.js applications using Yarn or pnpm as the package manager.

yarn

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddYarnApp("yarn-demo")
       .WithExternalHttpEndpoints();
```

Run Vite apps

This integration extension adds support for running the development server for Vite applications. By default, it uses the `npm` package manager to launch, but this can be overridden with the `packageManager` argument.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddViteApp("vite-demo")
       .WithExternalHttpEndpoints();

builder.AddViteApp("yarn-demo", packageManager: "yarn")
       .WithExternalHttpEndpoints();

builder.AddViteApp("pnpm-demo", packageManager: "pnpm")
       .WithExternalHttpEndpoints();

builder.Build().Run();
```

Install packages

When using the `WithNpmPackageInstallation`, `WithYarnPackageInstallation` or `WithPnpmPackageInstallation` methods, the package manager is used to install the packages before starting the application. These methods are useful to ensure that

packages are installed before the application starts, similar to how a .NET application would restore NuGet packages before running.

See also

- Orchestrate Node.js apps in .NET Aspire
- .NET Aspire Community Toolkit GitHub repo
- Sample Node.js apps

Community Toolkit Python hosting extensions

Article • 12/03/2024

Includes: Hosting integration not Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

In this article, you learn about the .NET Aspire Community Toolkit Python hosting extensions package which provides extra functionality to the .NET Aspire Python hosting package. The extensions package lets you run Uvicorn applications.

Hosting integration

To get started with the .NET Aspire Community Toolkit Python hosting extensions, install the CommunityToolkit.Aspire.Hosting.Python.Extensions NuGet package in the AppHost project.

.NET CLI

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.Hosting.Python.Extensions
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Example usage

To work with Python apps, they need to be within a virtual environment. To create a virtual environment, refer to the Initialize the Python virtual environment section.

In the *Program.cs* file of your app host project, call the `AddUvicornApp` method to add a Uvicorn application to the builder.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var uvicorn = builder.AddUvicornApp(
        name: "uvicornapp",
        projectDirectory: "../uvicornapp-api",
        appName: "main:app"
    )
    .WithHttpEndpoint(env: "PORT");

builder.Build().Run();
```

The `PORT` environment variable is used to determine the port the Uvicorn application should listen on. By default, this port is randomly assigned by .NET Aspire. The name of the environment variable can be changed by passing a different value to the WithHttpEndpoint method.

The Uvicorn application can be added as a reference to other resources in the AppHost project.

See also

- Orchestrate Python apps in .NET Aspire
- .NET Aspire Community Toolkit GitHub repo
- Sample Python apps

.NET Aspire Community Toolkit Ollama integration

Article • 11/20/2024

Includes: ✅ Hosting integration and ✅ Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

Ollama is a powerful, open source language model that can be used to generate text based on a given prompt. The .NET Aspire Ollama integration provides a way to host Ollama models using the docker.io/ollama/ollama container image and access them via the OllamaSharp client.

Hosting integration

The Ollama hosting integration models an Ollama server as the `OllamaResource` type, and provides the ability to add models to the server using the `AddModel` extension method, which represents the model as an `OllamaModelResource` type. To access these types and APIs that allow you to add the 📦 CommunityToolkit.Aspire.Hosting.Ollama NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.Hosting.Ollama
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Ollama resource

In the app host project, register and consume the Ollama integration using the `AddOllama` extension method to add the Ollama container to the application builder. You

can then add models to the container, which downloads and run when the container starts, using the `AddModel` extension method.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var ollama = builder.AddOllama("ollama");

var phi35 = ollama.AddModel("phi3.5");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(phi35);
```

Alternatively, if you want to use a model from the Hugging Face model hub, you can use the `AddHuggingFaceModel` extension method.

C#

```
var llama = ollama.AddHuggingFaceModel("llama", "bartowski/Llama-3.2-1B-Instruct-GGUF:IQ4_XS");
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/ollama/ollama` image, it creates a new Ollama instance on your local machine. For more information, see Container resource lifecycle.

Download the LLM

When the Ollama container for this integration first spins up, it downloads the configured LLMs. The progress of this download displays in the **State** column for this integration on the .NET Aspire dashboard.

> **Important**
>
> Keep the .NET Aspire orchestration app open until the download is complete, otherwise the download will be cancelled.

Cache the LLM

One or more LLMs are downloaded into the container which Ollama is running from, and by default this container is ephemeral. If you need to persist one or more LLMs

across container restarts, you need to mount a volume into the container using the `WithDataVolume` method.

C#

```
var ollama = builder.AddOllama("ollama")
                    .WithDataVolume();

var llama = ollama.AddModel("llama3");
```

Use GPUs when available

One or more LLMs are downloaded into the container which Ollama is running from, and by default this container runs on CPU. If you need to run the container in GPU you need to pass a parameter to the container runtime args.

Docker

C#

```
var ollama = builder.AddOllama("ollama")
                    .AddModel("llama3")
                    .WithContainerRuntimeArgs("--gpus=all");
```

For more information, see GPU support in Docker Desktop.

Hosting integration health checks

The Ollama hosting integration automatically adds a health check for the Ollama server and model resources. For the Ollama server, a health check is added to verify that the Ollama server is running and that a connection can be established to it. For the Ollama model resources, a health check is added to verify that the model is running and that the model is available, meaning the resource will be marked as unhealthy until the model has been downloaded.

Open WebUI support

The Ollama integration also provided support for running Open WebUI and having it communicate with the Ollama container.

C#

```
var ollama = builder.AddOllama("ollama")
                    .AddModel("llama3")
                    .WithOpenWebUI();
```

Client integration

To get started with the .NET Aspire OllamaSharp integration, install the 📦 CommunityToolkit.Aspire.OllamaSharp NuGet package in the client-consuming project, that is, the project for the application that uses the Ollama client.

.NET CLI

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.OllamaSharp
```

Add Ollama client API

In the *Program.cs* file of your client-consuming project, call the `AddOllamaClientApi` extension to register an `IOllamaClientApi` for use via the dependency injection container. If the resource provided in the app host, and referenced in the client-consuming project, is an `OllamaModelResource`, then the `AddOllamaClientApi` method will register the model as the default model for the `IOllamaClientApi`.

C#

```
builder.AddOllamaClientApi("llama3");
```

After adding `IOllamaClientApi` to the builder, you can get the `IOllamaClientApi` instance using dependency injection. For example, to retrieve your context object from service:

C#

```
public class ExampleService(IOllamaClientApi ollama)
{
    // Use ollama...
}
```

Add keyed Ollama client API

There might be situations where you want to register multiple `IOllamaClientApi` instances with different connection names. To register keyed Ollama clients, call the `AddKeyedOllamaClientApi` method:

C#

```
builder.AddKeyedOllamaClientApi(name: "chat");
builder.AddKeyedOllamaClientApi(name: "embeddings");
```

Then you can retrieve the `IOllamaClientApi` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("chat")] IOllamaClientApi chatOllama,
    [FromKeyedServices("embeddings")] IOllamaClientApi embeddingsOllama)
{
    // Use ollama...
}
```

Configuration

The Ollama client integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling the `AddOllamaClientApi` method:

C#

```
builder.AddOllamaClientApi("llama");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "llama": "Endpoint=http//localhost:1234;Model=llama3"
  }
}
```

Integration with `Microsoft.Extensions.AI`

The Microsoft.Extensions.AI library provides an abstraction over the Ollama client API, using generic interfaces. OllamaSharp supports these interfaces, and they can be registered using the `AddOllamaSharpChatClient` and `AddOllamaSharpEmbeddingGenerator` extension methods. These methods will also register the `IOllamaClientApi` instances with the dependency injection container, and have keyed versions for multiple instances.

C#

```
builder.AddOllamaSharpChatClient("llama");
```

After adding `IChatClient` to the builder, you can get the `IChatClient` instance using dependency injection. For example, to retrieve your context object from service:

C#

```
public class ExampleService(IChatClient chatClient)
{
    // Use chat client...
}
```

See also

- Ollama
- Open WebUI
- .NET Aspire Community Toolkit GitHub repo
- OllamaSharp
- Microsoft.Extensions.AI

.NET Aspire Community Toolkit Meilisearch integration

Article • 10/25/2024

Includes: ✅ Hosting integration and ✅ Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

In this article, you learn how to use the .NET Aspire Meilisearch hosting integration to run Meilisearch container and accessing it via the Meilisearch client.

Hosting integration

To run the Meilisearch container, install the CommunityToolkit.Aspire.Hosting.Meilisearch NuGet package in the app host project.

.NET CLI

```
.NET CLI

dotnet add package CommunityToolkit.Aspire.Hosting.Meilisearch
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add Meilisearch resource

In the app host project, register and consume the Meilisearch integration using the `AddMeilisearch` extension method to add the Meilisearch container to the application builder.

```
C#

var builder = DistributedApplication.CreateBuilder(args);

var meilisearch = builder.AddMeilisearch("meilisearch");
```

```
builder.AddProject<Projects.ExampleProject>()
       .WithReference(meilisearch);

// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/getmeili/meilisearch` image, it creates a new Meilisearch instance on your local machine. A reference to your Meilisearch resource (the `meilisearch` variable) is added to the `ExampleProject`. The Meilisearch resource includes a randomly generated `master key` using the CreateDefaultPasswordParameter method when a master key wasn't provided.

For more information, see Container resource lifecycle.

Add Meilisearch resource with data volume

To add a data volume to the Meilisearch resource, call the `Aspire.Hosting.MeilisearchBuilderExtensions.WithDataVolume` method on the Meilisearch resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var meilisearch = builder.AddMeilisearch("meilisearch")
                         .WithDataVolume();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(meilisearch);

// After adding all resources, run the app...
```

The data volume is used to persist the Meilisearch data outside the lifecycle of its container. The data volume is mounted at the `/meili_data` path in the Meilisearch container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add Meilisearch resource with data bind mount

To add a data bind mount to the Meilisearch resource, call the `Aspire.Hosting.MeilisearchBuilderExtensions.WithDataBindMount` method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var meilisearch = builder.AddMeilisearch("meilisearch")
                         .WithDataBindMount(
                             source: @"C:\Meilisearch\Data");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(meilisearch);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the Meilisearch data across container restarts. The data bind mount is mounted at the `C:\Meilisearch\Data` on Windows (or `/Meilisearch/Data` on Unix) path on the host machine in the Meilisearch container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add Meilisearch resource with master key parameter

When you want to explicitly provide the master key used by the container image, you can provide these credentials as parameters. Consider the following alternative example:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var masterkey = builder.AddParameter("masterkey", secret: true);
var meilisearch = builder.AddMeilisearch("meilisearch", masterkey);

builder.AddProject<Projects.ExampleProject>()
       .WithReference(meilisearch);

// After adding all resources, run the app...
```

For more information on providing parameters, see External parameters.

Client integration

To get started with the .NET Aspire Meilisearch client integration, install the CommunityToolkit.Aspire.Meilisearch NuGet package in the client-consuming project, that is, the project for the application that uses the Meilisearch client.

.NET CLI

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.Meilisearch
```

Add Meilisearch client

In the *Program.cs* file of your client-consuming project, call the `Microsoft.Extensions.Hosting.AspireMeilisearchExtensions.AddMeilisearchClient` extension method on any IHostApplicationBuilder to register an `MeilisearchClient` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddMeilisearchClient(connectionName: "meilisearch");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the Meilisearch resource in the app host project. For more information, see **Add Meilisearch resource**.

You can then retrieve the `MeilisearchClient` instance using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(MeilisearchClient client)
{
    // Use client...
}
```

Add keyed Meilisearch client

There might be situations where you want to register multiple `MeilisearchClient` instances with different connection names. To register keyed Meilisearch clients, call the `Microsoft.Extensions.Hosting.AspireMeilisearchExtensions.AddKeyedMeilisearchClient`

```
C#

builder.AddKeyedMeilisearchClient(name: "products");
builder.AddKeyedMeilisearchClient(name: "orders");
```

Then you can retrieve the `MeilisearchClient` instances using dependency injection. For example, to retrieve the connection from an example service:

```
C#

public class ExampleService(
    [FromKeyedServices("products")] MeilisearchClient productsClient,
    [FromKeyedServices("orders")] MeilisearchClient ordersClient)
{
    // Use clients...
}
```

For more information on keyed services, see .NET dependency injection: Keyed services.

Configuration

The .NET Aspire Meilisearch client integration provides multiple options to configure the server connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddMeilisearchClient`:

```
C#

builder.AddMeilisearchClient("meilisearch");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "meilisearch": "Endpoint=http://localhost:19530/;MasterKey=123456!@#$%"
  }
}
```

Use configuration providers

The .NET Aspire Meilisearch Client integration supports Microsoft.Extensions.Configuration. It loads the `CommunityToolkit.Aspire.Meilisearch.MeilisearchClientSettings` from configuration by using the `Aspire:Meilisearch:Client` key. Consider the following example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "Meilisearch": {
      "Client": {
        "Endpoint": "http://localhost:19530/",
        "MasterKey": "123456!@#$%"
      }
    }
  }
}
```

Use inline delegates

Also you can pass the `Action<MeilisearchClientSettings> configureSettings` delegate to set up some or all the options inline, for example to set the API key from code:

C#

```
builder.AddMeilisearchClient(
    "meilisearch",
    static settings => settings.MasterKey = "123456!@#$%");
```

Client integration health checks

The .NET Aspire Meilisearch integration uses the configured client to perform a `IsHealthyAsync`. If the result is `true`, the health check is considered healthy, otherwise

it's unhealthy. Likewise, if there's an exception, the health check is considered unhealthy with the error propagating through the health check failure.

See also

- Meilisearch
- Meilisearch Client
- .NET Aspire Community Toolkit GitHub repo

.NET Aspire Rust hosting

Article • 11/20/2024

Includes: Hosting integration not Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

Rust is a general-purpose programming language emphasizing performance, type safety, and concurrency. It enforces memory safety, meaning that all references point to valid memory. The .NET Aspire Rust hosting integration allows you to host Rust applications in your .NET Aspire app host project, and provide it to other resources in your application.

Hosting integration

The Rust hosting integration models a Rust application as the `Aspire.Hosting.ApplicationModel.RustAppExecutableResource` type. To access this type and APIs that allow you to add it to your app host project, install the CommunityToolkit.Aspire.Hosting.Rust NuGet package in the app host project.

This integration expects that the Rust programming language has already been installed on the host machine and the Rust package manager cargo is available in the system path.

.NET CLI

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.Hosting.Rust
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add a Rust resource

In the *Program.cs* file of your app host project, call the `Aspire.Hosting.RustAppHostingExtension.AddRustApp` on the `builder` instance to add a Rust application resource as shown in the following example:

C#

The working directory of the application should be the root of Rust application directory. Also you can customize running behavior by passing args parameter to the `AddRustApp` method.

C#

The Rust application can be added as a reference to other resources in the app host project.

.NET Aspire Community Toolkit GitHub repo

.NET Aspire SQL Database Projects hosting integration

Article • 02/27/2025

Includes: Hosting integration not Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

In this article, you learn how to use the .NET Aspire SQL Database Projects hosting integration to publish your database schema to your SQL Server database.

Prerequisites

This integration requires a SQL Database Project based on either MSBuild.Sdk.SqlProj or Microsoft.Build.Sql.

Hosting integration

To get started with the .NET Aspire SQL Database Projects hosting integration, install the CommunityToolkit.Aspire.Hosting.SqlDatabaseProjects NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.Hosting.SqlDatabaseProjects
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Example usage

Add a reference to the MSBuild.Sdk.SqlProj or Microsoft.Build.Sql project you want to publish in your .NET Aspire app host project:

.NET CLI

```
dotnet add reference ../MySqlProj/MySqlProj.csproj
```

> **Note**
>
> Adding this reference will currently result in warning `ASPIRE004` on the project due to how references are parsed. The .NET Aspire team is aware of this and we're working on a cleaner solution.

Add the project as a resource to your .NET Aspire AppHost:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var sql = builder.AddSqlServer("sql")
                 .AddDatabase("test");

builder.AddSqlProject<Projects.MySqlProj>("mysqlproj")
       .WithReference(sql);
```

Now when you run your .NET Aspire app host project you see the SQL Database Project being published to the specified SQL Server.

NuGet Package support

Starting with version 9.2.0, you can deploy databases from referenced NuGet packages, such as those produced by MSBuild.Sdk.SqlProj or Microsoft.Build.Sql. To deploy, add the NuGet package to your Aspire app host project, for example:

.NET CLI

```
dotnet add package ErikEJ.Dacpac.Chinook
```

Next, edit your project file to set the `IsAspirePackageResource` flag to `True` for the corresponding `PackageReference`, as shown in the following example:

XML

```
<PackageReference Include="ErikEJ.Dacpac.Chinook" Version="1.0.0"
                  IsAspirePackageResource="True" />
```

Finally, add the package as a resource to your app model:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var sql = builder.AddSqlServer("sql")
                 .AddDatabase("test");

builder.AddSqlPackage<Packages.ErikEJ_Dacpac_Chinook>("chinook")
       .WithReference(sql);
```

> ⓘ Note
>
> By default, the *.dacpac* is expected to be located under `tools/<package-id>.dacpac`. In the preceding example, the *tools/ErikEJ.Dacpac.Chinook.dacpac* path is expected. If for whatever reason the *.dacpac* is under a different path within the package you can use `WithDacpac("relative/path/to/some.dacpac")` API to specify a path relative to the root of app host project directory.

Local .dacpac file support

If you are sourcing your *.dacpac* file from somewhere other than a project reference, you can also specify the path to the *.dacpac* file directly:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var sql = builder.AddSqlServer("sql")
                 .AddDatabase("test");

builder.AddSqlProject("mysqlproj")
       .WithDacpac("path/to/mysqlproj.dacpac")
       .WithReference(sql);
```

Support for existing SQL Server instances

Starting with version 9.2.0, you can publish the SQL Database project to an existing SQL Server instance by using a connection string:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

// Get an existing SQL Server connection string from the configuration
var connection = builder.AddConnectionString("Aspire");

builder.AddSqlProject<Projects.SdkProject>("mysqlproj")
       .WithReference(connection);

builder.Build().Run();
```

Deployment options support

To define options that affect the behavior of package deployment, call the `WithConfigureDacDeployOptions` API:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var sql = builder.AddSqlServer("sql")
                 .AddDatabase("test");

builder.AddSqlProject("mysqlproj")
       .WithConfigureDacDeployOptions(options =>
options.IncludeCompositeObjects = true)
       .WithReference(sql);

builder.Build().Run();
```

The preceding code:

- Adds a SQL server resource named `sql` and adds a `test` database resource to it.
- Adds a SQL project resource named `mysqlproj` and then configures the DacDeployOptions.
- The SQL project resource depends on the database resource.

Redeploy support

If you make changes to your SQL Database project while the app host is running, you can use the `Redeploy` custom action on the .NET Aspire dashboard to redeploy your

updates without having to restart the app host.

See also

- MSBuild.Sdk.SqlProj GitHub repository
- Microsoft.Build.Sql GitHub repository
- Get started with SQL database projects
- .NET Aspire Community Toolkit GitHub repo

Community Toolkit Azure Data API Builder hosting integration

Article • 11/20/2024

Includes: ✅ Hosting integration not ⊗ Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

In this article, you learn how to use the .NET Aspire Data API Builder hosting integration to run Data API Builder as a container.

Hosting integration

To get started with the .NET Aspire Azure Data API Builder hosting integration, install the 📦 CommunityToolkit.Aspire.Hosting.Azure.DataApiBuilder NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.Hosting.Azure.DataApiBuilder
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Usage

In the app host project, register and consume the Data API Builder integration using the `AddDataAPIBuilder` extension method to add the Data API Builder container to the application builder.

C#

```
 var builder = DistributedApplication.CreateBuilder();

// Add Data API Builder using dab-config.json
var dab = builder.AddDataAPIBuilder("dab");

builder.AddProject<Projects.ExampleProject>()
        .WithReference(dab);

 // After adding all resources, run the app...
```

When the .NET Aspire adds a container image to the app host, as shown in the preceding example with the `mcr.microsoft.com/azure-databases/data-api-builder` image, it creates a new Data API Builder instance on your local machin. A reference to the DAB resource (the `dab` variable) is added to the `ExampleProject` project.

Configuration

Expand table

Parameter	Description
`name`	The name of the resource is a required `string` and it's validated by the ResourceNameAttribute.
`configFilePaths`	The paths to the configuration or schema file(s) for Data API builder. These are optional and are available as a `params string[]`, meaning you can omit them altogether, or provide one or more path inline. When omitted, it defaults to `"./dab-config.json"`.
`httpPort`	The port number for the Data API Builder container is represented as a an `int?`. By default, the port is `null`, .NET Aspire assigns a port when this isn't otherwise provided.

Data API Builder container image configuration

You can specify custom container `registry/image/tag` values by using the following APIs chained to the `IResourceBuilder<DataApiBuilderContainerResource>`:

- `WithImageRegistry`: Pass the desired registry name, such as `ghcr.io` for the GitHub Container Registry or `docker.io` for Docker.
- `WithImage`: Provide the name of the image, such as `azure-databases/data-api-builder`.
- `WithImageTag`: Specify an image tag to use other than `latest`, which is the default in most cases.

Consider the following example that demonstrates chaining these APIs together, to fluently express that the Data API Builder's container image is fully qualified as `mcr.microsoft.com/azure-databases/data-api-builder:latest`:

C#

```
var dab = builder.AddDataAPIBuilder("dab")
    .WithImageRegistry("mcr.microsoft.com")
    .WithImage("azure-databases/data-api-builder")
    .WithImageTag("latest");
```

Database Configuration

If you need to configure your own local database, you can refer to the SQL Server integration documentation.

Once you have your database added as a resource, you can reference it using the following APIs chained to the `IResourceBuilder<DataApiBuilderContainerResource>`:

C#

```
var dab = builder.AddDataAPIBuilder("dab")
    .WithReference(sqlDatabase)
    .WaitFor(sqlDatabase);
```

The `WaitFor` method ensures that the database is ready before starting the Data API Builder container.

Referencing the `sqlDatabase` resource will inject its connection string into the Data API Builder container with the name `ConnectionStrings__<DATABASE_RESOURCE_NAME>`. Next, update the `dab-config.json` file to include the connection string for the database:

JSON

```
"data-source": {
    "connection-string":
"@env('ConnectionStrings__<DATABASE_RESOURCE_NAME>')",
}
```

Using multiple data sources

You can pass multiple configuration files to the `AddDataAPIBuilder` method:

C#

```
var dab = builder.AddDataAPIBuilder("dab",
        "./dab-config-1.json",
        "./dab-config-2.json")
    .WithReference(sqlDatabase1)
    .WaitFor(sqlDatabase1)
    .WithReference(sqlDatabase2)
    .WaitFor(sqlDatabase2);
```

> **Note**
>
> All files are mounted/copied to the same `/App` folder.

See also

- .NET Aspire Community Toolkit GitHub repo
- Sample DAB
- Further usage examples

.NET Aspire Community Toolkit EventStore integration

Article • 11/22/2024

Includes: ✅ Hosting integration and ✅ Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

In this article, you learn how to use the .NET Aspire EventStore hosting integration to run EventStore container and accessing it via the EventStore client.

Hosting integration

To run the EventStore container, install the CommunityToolkit.Aspire.Hosting.EventStore NuGet package in the app host project.

.NET CLI

```
.NET CLI

dotnet add package CommunityToolkit.Aspire.Hosting.EventStore
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add EventStore resource

In the app host project, register and consume the EventStore integration using the `AddEventStore` extension method to add the EventStore container to the application builder.

```
C#

var builder = DistributedApplication.CreateBuilder(args);

var eventstore = builder.AddEventStore("eventstore");
```

```
builder.AddProject<Projects.ExampleProject>()
       .WithReference(eventstore);

// After adding all resources, run the app...
```

When .NET Aspire adds a container image to the app host, as shown in the preceding example with the `docker.io/eventstore/eventstore` image, it creates a new EventStore instance on your local machine. A reference to your EventStore resource (the `eventstore` variable) is added to the `ExampleProject`.

For more information, see Container resource lifecycle.

Add EventStore resource with data volume

To add a data volume to the EventStore resource, call the `Aspire.Hosting.EventStoreBuilderExtensions.WithDataVolume` method on the EventStore resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var eventstore = builder.AddEventStore("eventstore")
                        .WithDataVolume();

builder.AddProject<Projects.ExampleProject>()
       .WithReference(eventstore);

// After adding all resources, run the app...
```

The data volume is used to persist the EventStore data outside the lifecycle of its container. The data volume is mounted at the `/var/lib/eventstore` path in the EventStore container and when a `name` parameter isn't provided, the name is generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

Add EventStore resource with data bind mount

To add a data bind mount to the EventStore resource, call the `Aspire.Hosting.EventStoreBuilderExtensions.WithDataBindMount` method:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var eventstore = builder.AddEventStore("eventstore")
                        .WithDataBindMount(source: @"C:\EventStore\Data");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(eventstore);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the EventStore data across container restarts. The data bind mount is mounted at the `C:\EventStore\Data` on Windows (or `/EventStore/Data` on Unix) path on the host machine in the EventStore container. For more information on data bind mounts, see Docker docs: Bind mounts.

Add EventStore resource with log volume

To add a log volume to the EventStore resource, call the WithVolume extension method on the EventStore resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var eventstore = builder.AddEventStore("eventstore")
                        .WithVolume(name: "eventstore_logs", target:
"/var/log/eventstore");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(eventstore);

// After adding all resources, run the app...
```

The data volume is used to persist the EventStore logs outside the lifecycle of its container. The data volume must be mounted at the `/var/log/eventstore` target path in the EventStore container and when a `name` parameter isn't provided, the name is

generated at random. For more information on data volumes and details on why they're preferred over bind mounts, see Docker docs: Volumes.

For more information about EventStore logs location, see EventStore Resources: Logs.

Add EventStore resource with log bind mount

To add a log bind mount to the EventStore resource, call the WithBindMount extension method on the EventStore resource:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var eventstore = builder.AddEventStore("eventstore")
                        .WithBindMount(@"C:\EventStore\Logs",
"/var/log/eventstore");

builder.AddProject<Projects.ExampleProject>()
       .WithReference(eventstore);

// After adding all resources, run the app...
```

> **Important**
>
> Data bind mounts have limited functionality compared to volumes, which offer better performance, portability, and security, making them more suitable for production environments. However, bind mounts allow direct access and modification of files on the host system, ideal for development and testing where real-time changes are needed.

Data bind mounts rely on the host machine's filesystem to persist the EventStore logs across container restarts. The data bind mount is mounted at the `C:\EventStore\Logs` on Windows (or `/EventStore/Logs` on Unix) path on the host machine in the EventStore container. The target path must be set to the log folder used by the EventStore container (`/var/log/eventstore`).

For more information about EventStore logs location, see EventStore Resources: Logs.

For more information on data bind mounts, see Docker docs: Bind mounts.

Client integration

To get started with the .NET Aspire EventStore client integration, install the CommunityToolkit.Aspire.EventStore NuGet package in the client-consuming project, that is, the project for the application that uses the EventStore client.

.NET CLI

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.EventStore
```

Add EventStore client

In the *Program.cs* file of your client-consuming project, call the `Microsoft.Extensions.Hosting.AspireEventStoreExtensions.AddEventStoreClient` extension method on any IHostApplicationBuilder to register an `EventStoreClient` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddEventStoreClient(connectionName: "eventstore");
```

> **Tip**
>
> The `connectionName` parameter must match the name used when adding the EventStore resource in the app host project. For more information, see **Add EventStore resource**.

You can then retrieve the `EventStoreClient` instance using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(EventStoreClient client)
{
    // Use client...
}
```

Add keyed EventStore client

There might be situations where you want to register multiple `EventStoreClient` instances with different connection names. To register keyed EventStore clients, call the `Microsoft.Extensions.Hosting.AspireEventStoreExtensions.AddKeyedEventStoreClient`

C#

Then you can retrieve the `EventStoreClient` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

For more information on keyed services, see .NET dependency injection: Keyed services.

The .NET Aspire EventStore client integration provides multiple options to configure the server connection based on the requirements and conventions of your project.

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

```
{
  "ConnectionStrings": {
    "eventstore": "esdb://localhost:22113?tls=false"
  }
}
```

Use configuration providers

The .NET Aspire EventStore Client integration supports Microsoft.Extensions.Configuration. It loads the `CommunityToolkit.Aspire.EventStore.EventStoreSettings` from configuration by using the `Aspire:EventStore:Client` key. Consider the following example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "EventStore": {
      "Client": {
        "ConnectionString": "esdb://localhost:22113?tls=false",
        "DisableHealthChecks": true
      }
    }
  }
}
```

Use inline delegates

Also you can pass the `Action<EventStoreSettings> configureSettings` delegate to set up some or all the options inline, for example to set the API key from code:

C#

```
builder.AddEventStoreClient(
    "eventstore",
    static settings => settings.DisableHealthChecks = true);
```

Client integration health checks

The .NET Aspire EventStore integration uses the configured client to perform a `IsHealthyAsync`. If the result is `true`, the health check is considered healthy, otherwise

it's unhealthy. Likewise, if there's an exception, the health check is considered unhealthy with the error propagating through the health check failure.

See also

- EventStore
- EventStore Client
- .NET Aspire Community Toolkit GitHub repo

.NET Aspire Community Toolkit SQLite Entity Framework integration

Article • 03/05/2025

Includes: ✅ Hosting integration and ✅ Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

SQLite is a lightweight, serverless, self-contained SQL database engine that is widely used for local data storage in applications. The .NET Aspire SQLite integration provides a way to use SQLite databases within your .NET Aspire applications, and access them via the Microsoft.EntityFrameworkCore.Sqlite Entity Framework support package.

Hosting integration

The SQLite hosting integration models a SQLite database as the `SQLiteResource` type and will create the database file in the specified location. To access these types and APIs that allow you to add the CommunityToolkit.Aspire.Hosting.SQLite NuGet package in the app host project.

.NET CLI

```
.NET CLI
dotnet add package CommunityToolkit.Aspire.Hosting.SQLite
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add SQLite resource

In the app host project, register and consume the SQLite integration using the `AddSQLite` extension method to add the SQLite database to the application builder.

```
C#
```

```
var builder = DistributedApplication.CreateBuilder(args);

var sqlite = builder.AddSQLite("my-database");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(sqlite);
```

When .NET Aspire adds a SQLite database to the app host, as shown in the preceding example, it creates a new SQLite database file in the users temp directory.

Alternatively, if you want to specify a custom location for the SQLite database file, provide the relevant arguments to the `AddSqlite` method.

C#

```
var sqlite = builder.AddSQLite("my-database", "C:\\Database\\Location", "my-database.db");
```

Add SQLiteWeb resource

When adding the SQLite resource, you can also add the SQLiteWeb resource, which provides a web interface to interact with the SQLite database. To do this, use the `WithSqliteWeb` extension method.

C#

```
var sqlite = builder.AddSQLite("my-database")
                    .WithSqliteWeb();
```

This code adds a container based on `ghcr.io/coleifer/sqlite-web` to the app host, which provides a web interface to interact with the SQLite database it is connected to. Each SQLiteWeb instance is connected to a single SQLite database, meaning that if you add multiple SQLiteWeb instances, there will be multiple SQLiteWeb containers.

Adding SQLite extensions

SQLite supports extensions that can be added to the SQLite database. Extensions can either be provided via a NuGet package, or via a location on disk. Use either the `WithNuGetExtension` or `WithLocalExtension` extension methods to add extensions to the SQLite database.

> ⓘ **Note**

The SQLite extensions support is considered experimental and produces a `CTASPIRE002` warning.

Client integration

To get started with the .NET Aspire SQLite EF client integration, install the CommunityToolkit.Aspire.Microsoft.EntityFrameworkCore.Sqlite NuGet package in the client-consuming project, that is, the project for the application that uses the SQLite client. The SQLite client integration registers a SqliteConnection instance that you can use to interact with SQLite.

.NET CLI

.NET CLI

```
dotnet add package
CommunityToolkit.Aspire.Microsoft.EntityFrameworkCore.Sqlite
```

Add Sqlite client

In the *Program.cs* file of your client-consuming project, call the `Microsoft.Extensions.Hosting.AspireEFSqliteExtensions.AddSqliteDbContext` extension method on any IHostApplicationBuilder to register your DbContext subclass for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddSqliteDbContext<YourDbContext>(connectionName: "sqlite");
```

Tip

The `connectionName` parameter must match the name used when adding the SQLite resource in the app host project. For more information, see Add SQLite resource.

After adding `YourDbContext` to the builder, you can get the `YourDbContext` instance using dependency injection. For example, to retrieve your data source object from an example service define it as a constructor parameter and ensure the `ExampleService` class is registered with the dependency injection container:

C#

```
public class ExampleService(YourDbContext context)
{
    // Use context...
}
```

For more information on dependency injection, see .NET dependency injection.

Enrich a SQLite database context

You may prefer to use the standard Entity Framework method to obtain the database contextand add it to the dependency injection container:

C#

```
builder.Services.AddDbContext<YourDbContext>(options =>
    options.UseSqlite(builder.Configuration.GetConnectionString("sqlite")
        ?? throw new InvalidOperationException("Connection string 'sqlite'
not found.")));
```

> **Note**
>
> The connection string name that you pass to the GetConnectionString method must match the name used when adding the SQLite resource in the app host project. For more information, see Add SQLite resource.

Configuration

The SQLite client integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling the `Microsoft.Extensions.Hosting.AspireEFSqliteExtensions.AddSqliteDbContext` method:

C#

```
builder.AddSqliteDbContext<YourDbContext>("sqlite");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section.

JSON

```
{
  "ConnectionStrings": {
    "sqlite": "Data Source=C:\\Database\\Location\\my-database.db"
  }
}
```

Use configuration providers

The SQLite client integration supports Microsoft.Extensions.Configuration. It loads the `Microsoft.Extensions.Hosting.SqliteConnectionSettings` from the *appsettings.json* or other configuration providers by using the `Aspire:Sqlite:EntityFrameworkCore:Sqlite` key. Example _appsettings.json that configures some of the options:

JSON

```
{
  "Aspire": {
    "Sqlite": {
      "EntityFrameworkCore": {
        "Sqlite": {
          "ConnectionString": "Data Source=C:\\Database\\Location\\my-
database.db",
          "DisableHealthCheck": true
        }
      }
    }
  }
}
```

.NET Aspire Community Toolkit SQLite integration

Article • 03/05/2025

Includes: ✅ Hosting integration and ✅ Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

SQLite is a lightweight, serverless, self-contained SQL database engine that is widely used for local data storage in applications. The .NET Aspire SQLite integration provides a way to use SQLite databases within your .NET Aspire applications, and access them via the Microsoft.Data.Sqlite client.

Hosting integration

The SQLite hosting integration models a SQLite database as the `SQLiteResource` type and will create the database file in the specified location. To access these types and APIs that allow you to add the CommunityToolkit.Aspire.Hosting.SQLite NuGet package in the app host project.

.NET CLI

```
.NET CLI
dotnet add package CommunityToolkit.Aspire.Hosting.SQLite
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Add SQLite resource

In the app host project, register and consume the SQLite integration using the `AddSQLite` extension method to add the SQLite database to the application builder.

```
C#
```

```
var builder = DistributedApplication.CreateBuilder(args);

var sqlite = builder.AddSQLite("my-database");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
                            .WithReference(sqlite);
```

When .NET Aspire adds a SQLite database to the app host, as shown in the preceding example, it creates a new SQLite database file in the users temp directory.

Alternatively, if you want to specify a custom location for the SQLite database file, provide the relevant arguments to the `AddSqlite` method.

C#

```
var sqlite = builder.AddSQLite("my-database", "C:\\Database\\Location", "my-database.db");
```

Add SQLiteWeb resource

When adding the SQLite resource, you can also add the SQLiteWeb resource, which provides a web interface to interact with the SQLite database. To do this, use the `WithSqliteWeb` extension method.

C#

```
var sqlite = builder.AddSQLite("my-database")
                    .WithSqliteWeb();
```

This code adds a container based on `ghcr.io/coleifer/sqlite-web` to the app host, which provides a web interface to interact with the SQLite database it is connected to. Each SQLiteWeb instance is connected to a single SQLite database, meaning that if you add multiple SQLiteWeb instances, there will be multiple SQLiteWeb containers.

Adding SQLite extensions

SQLite supports extensions that can be added to the SQLite database. Extensions can either be provided via a NuGet package, or via a location on disk. Use either the `WithNuGetExtension` or `WithLocalExtension` extension methods to add extensions to the SQLite database.

> ⓘ **Note**

The SQLite extensions support is considered experimental and produces a `CTASPIRE002` warning.

Client integration

To get started with the .NET Aspire SQLite client integration, install the CommunityToolkit.Aspire.Microsoft.Data.Sqlite NuGet package in the client-consuming project, that is, the project for the application that uses the SQLite client. The SQLite client integration registers a SqliteConnection instance that you can use to interact with SQLite.

.NET CLI

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.Microsoft.Data.Sqlite
```

Add Sqlite client

In the *Program.cs* file of your client-consuming project, call the `Microsoft.Extensions.Hosting.AspireSqliteExtensions.AddSqliteConnection` extension method on any IHostApplicationBuilder to register a `SqliteConnection` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddSqliteConnection(connectionName: "sqlite");
```

Tip

The `connectionName` parameter must match the name used when adding the SQLite resource in the app host project. For more information, see Add SQLite resource.

After adding `SqliteConnection` to the builder, you can get the `SqliteConnection` instance using dependency injection. For example, to retrieve your connection object from an example service define it as a constructor parameter and ensure the `ExampleService` class is registered with the dependency injection container:

C#

```
public class ExampleService(SqliteConnection connection)
{
    // Use connection...
}
```

For more information on dependency injection, see .NET dependency injection.

Add keyed Sqlite client

There might be situations where you want to register multiple `SqliteConnection` instances with different connection names. To register keyed Sqlite clients, call the `Microsoft.Extensions.Hosting.AspireSqliteExtensions.AddKeyedSqliteConnection` method:

C#

```
builder.AddKeyedSqliteConnection(name: "chat");
builder.AddKeyedSqliteConnection(name: "queue");
```

Then you can retrieve the `SqliteConnection` instances using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(
    [FromKeyedServices("chat")] SqliteConnection chatConnection,
    [FromKeyedServices("queue")] SqliteConnection queueConnection)
{
    // Use connections...
}
```

Configuration

The SQLite client integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling the `Microsoft.Extensions.Hosting.AspireSqliteExtensions.AddSqliteConnection` method:

C#

```
builder.AddSqliteConnection("sqlite");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section.

JSON

```
{
  "ConnectionStrings": {
    "sqlite": "Data Source=C:\\Database\\Location\\my-database.db"
  }
}
```

Use configuration providers

The SQLite client integration supports Microsoft.Extensions.Configuration. It loads the `Microsoft.Extensions.Hosting.SqliteConnectionSettings` from the *appsettings.json* or other configuration providers by using the `Aspire:Sqlite:Client` key. Example *appsettings.json* that configures some of the options:

JSON

```
{
  "Aspire": {
    "Sqlite": {
      "Client": {
        "ConnectionString": "Data Source=C:\\Database\\Location\\my-
database.db",
        "DisableHealthCheck": true
      }
    }
  }
}
```

.NET Aspire Community Toolkit SQL Server hosting extensions

Article • 03/05/2025

Includes: ✅ Hosting integration not ⊗ Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

In this article, you learn about the .NET Aspire Community Toolkit SQL Server hosting extensions package which provides extra functionality to the .NET Aspire SQL Server hosting package.

This package provides the following features:

- DbGate management UI

Hosting integration

To get started with the .NET Aspire Community Toolkit SQL Server hosting extensions, install the CommunityToolkit.Aspire.Hosting.SqlServer.Extensions NuGet package in the AppHost project.

.NET CLI

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.Hosting.SqlServer.Extensions
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Example usage

To add the DbGate management UI to your SQL Server resource, call the `WithDbGate` method on the `SqlServerResourceBuilder` instance.

C#

```
var sqlserver = builder.AddSqlServer("sqlserver")
    .WithDbGate();
```

This will add a new resource to the app host which will be available from the .NET Aspire dashboard.

.NET Aspire Community Toolkit MongoDB hosting extensions

Article • 03/05/2025

Includes: Hosting integration not Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

In this article, you learn about the .NET Aspire Community Toolkit MongoDB hosting extensions package which provides extra functionality to the .NET Aspire MongoDB hosting package.

This package provides the following features:

- DbGate management UI

Hosting integration

To get started with the .NET Aspire Community Toolkit MongoDB hosting extensions, install the CommunityToolkit.Aspire.Hosting.MongoDB.Extensions NuGet package in the AppHost project.

.NET CLI

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.Hosting.MongoDB.Extensions
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Example usage

To add the DbGate management UI to your MongoDB resource, call the `WithDbGate` method on the `MongoDBResourceBuilder` instance.

C#

```
var MongoDB = builder.AddMongoDB("MongoDB")
    .WithDbGate();
```

This will add a new resource to the app host which will be available from the .NET Aspire dashboard.

.NET Asp nmunity Toolkit Redis hosting e sions

Article • 03/05/2025

Includes: Hosting tegration not Client integration

article, you learn ab hosting extensions package which p spire Redis hosting package .

This package provides the follo

- DbGate management UI

To get started with the .NET Aspire Community Toolkit Redis hosting extensions, install the CommunityToolkit.Aspire.Hosting.Redis.Extensions NuGet package in the AppHost project.

.NET CLI

For more information, see dotnet add package or Manage package dependencies in .NET applications.

To add the DbGate management UI to your Redis resource, call the `WithDbGate` method on the `RedisResource` instance.

C#

```
var redis = builder.AddRedis("Redis")
    .WithDbGate();
```

This will add a new resource to the app host which will be available from the .NET Aspire dashboard.

.NET Aspire Community Toolkit PostgreSQL hosting extensions

Article • 03/05/2025

Includes: Hosting integration not Client integration

> **Note**
>
> This integration is part of the .NET Aspire Community Toolkit and *isn't* officially supported by the .NET Aspire team.

In this article, you learn about the .NET Aspire Community Toolkit PostgreSQL hosting extensions package which provides extra functionality to the .NET Aspire PostgreSQL hosting package.

This package provides the following features:

- DbGate management UI

Hosting integration

To get started with the .NET Aspire Community Toolkit PostgreSQL hosting extensions, install the CommunityToolkit.Aspire.Hosting.PostgreSQL.Extensions NuGet package in the AppHost project.

.NET CLI

```
dotnet add package CommunityToolkit.Aspire.Hosting.PostgreSQL.Extensions
```

For more information, see dotnet add package or Manage package dependencies in .NET applications.

Example usage

To add the DbGate management UI to your PostgreSQL resource, call the `WithDbGate` method on the `PostgresServerResource` instance.

C#

```
var postgresServer = builder.AddPostgreSQL("PostgreSQL")
    .WithDbGate();
```

This will add a new resource to the app host which will be available from the .NET Aspire dashboard.

Create custom .NET Aspire hosting integrations

Article • 11/11/2024

.NET Aspire improves the development experience by providing reusable building blocks that can be used to quickly arrange application dependencies and expose them to your own code. One of the key building blocks of an Aspire-based application is the *resource*. Consider the code below:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var redis = builder.AddRedis("cache");

var db = builder.AddPostgres("pgserver")
                .AddDatabase("inventorydb");

builder.AddProject<Projects.InventoryService>("inventoryservice")
       .WithReference(redis)
       .WithReference(db);
```

In the preceding code there are four resources represented:

1. `cache`: A Redis container.
2. `pgserver`: A Postgres container.
3. `inventorydb`: A database hosted on `pgserver`.
4. `inventoryservice`: An ASP.NET Core application.

Most .NET Aspire-related code that the average developer writes, centers around adding resources to the app model and creating references between them.

Key elements of a .NET Aspire custom resource

Building a custom resource in .NET Aspire requires the following:

1. A custom resource type that implements IResource
2. An extension method for IDistributedApplicationBuilder named `Add{CustomResource}` where `{CustomResource}` is the name of the custom resource.

When custom resource requires optional configuration, developers may wish to implement `With*` suffixed extension methods to make these configuration options

discoverable using the *builder pattern*.

A practical example: MailDev

To help understand how to develop custom resources, this article shows an example of how to build a custom resource for MailDev. MailDev is an open-source tool which provides a local mail server designed to allow developers to test e-mail sending behaviors within their app. For more information, see the MailDev GitHub repository.

In this example you create a new .NET Aspire project as a test environment for the MailDev resource that you create. While you can create custom resources in existing .NET Aspire projects it's a good idea to consider whether the custom resource might be used across multiple .NET Aspire-based solutions and should be developed as a reusable integration.

Set up the starter project

Create a new .NET Aspire project that is used to test out the new resource that we're developing.

.NET CLI

```
dotnet new aspire -o MailDevResource
cd MailDevResource
dir
```

Once the project is created, you should see a listing containing the following:

- `MailDevResource.AppHost`: The app host used to test out the custom resource.
- `MailDevResource.ServiceDefaults`: The service defaults project for use in service-related projects.
- `MailDevResource.sln`: The solution file referencing both projects.

Verify that the project can build and run successfully by executing the following command:

.NET CLI

```
dotnet run --project MailDevResource.AppHost/MailDevResource.AppHost.csproj
```

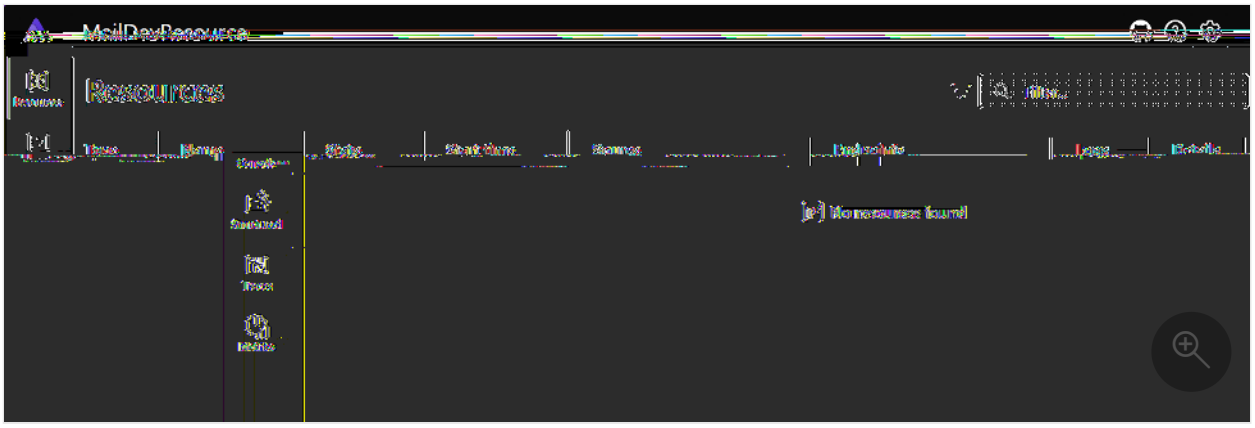
The console output should look similar to the following:

```
```

.NET CLI

```
Building...
info: Aspire.Hosting.DistributedApplication[0]
      Aspire version: 9.0.0
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application starting.
info: Aspire.Hosting.DistributedApplication[0]
      Application host directory is:
      ..\docs-
aspire\docs\extensibility\snippets\MailDevResource\MailDevResource.AppHost
info: Aspire.Hosting.DistributedApplication[0]
      Now listening on: https://localhost:17251
info: Aspire.Hosting.DistributedApplication[0]
      Login to the dashboard at https://localhost:17251/login?
t=928db244c720c5022a7a9bf5cf3a3526
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application started. Press Ctrl+C to shut down.
```

Select the dashboard link in the browser to see the .NET Aspire dashboard:



Press `Ctrl` + `C` to shut down the app (you can close the browser tab).

Create library for resource extension

.NET Aspire resources are just classes and methods contained within a class library that references the .NET Aspire Hosting library (`Aspire.Hosting`). By placing the resource in a separate project, you can more easily share it between .NET Aspire-based apps and potentially package and share it on NuGet.

1. Create the class library project named *MailDev.Hosting*.

   .NET CLI

   ```
   dotnet new classlib -o MailDev.Hosting
   ```

2. Add `Aspire.Hosting` to the class library as a package reference.

.NET CLI

```
dotnet add ./MailDev.Hosting/MailDev.Hosting.csproj package
Aspire.Hosting --version 9.0.0
```

> **Important**
>
> The version you specify here should match the version of the .NET Aspire workload installed.

3. Add class library reference to the *MailDevResource.AppHost* project.

.NET CLI

```
dotnet add ./MailDevResource.AppHost/MailDevResource.AppHost.csproj
reference ./MailDev.Hosting/MailDev.Hosting.csproj
```

4. Add class library project to the solution file.

.NET CLI

```
dotnet sln ./MailDevResource.sln add
./MailDev.Hosting/MailDev.Hosting.csproj
```

Once the following steps are performed, you can launch the project:

.NET CLI

```
dotnet run --project
./MailDevResource.AppHost/MailDevResource.AppHost.csproj
```

This results in a warning being displayed to the console:

Output

```
.\.nuget\packages\aspire.hosting.apphost\9.0.0\build\Aspire.Hosting.AppHost.
targets(174,5): warning ASPIRE004:
'..\MailDev.Hosting\MailDev.Hosting.csproj' is referenced by an A
spire Host project, but it is not an executable. Did you mean to set
IsAspireProjectResource="false"? [D:\source\repos\docs-
aspire\docs\extensibility\snippets\MailDevResource\MailDevResource.AppHost\M
```

```
ailDevRe
source.AppHost.csproj]
```

This is because .NET Aspire treats project references in the app host as if they're service projects. To tell .NET Aspire that the project reference should be treated as a nonservice project modify the *MailDevResource.AppHostMailDevResource.AppHost.csproj* files reference to the `MailDev.Hosting` project to be the following:

XML

```xml
<ItemGroup>
  <!-- The IsAspireProjectResource attribute tells .NET Aspire to treat this
       reference as a standard project reference and not attempt to generate
       a metadata file -->
  <ProjectReference Include="..\MailDev.Hosting\MailDev.Hosting.csproj"
                    IsAspireProjectResource="false" />
</ItemGroup>
```

Now when you launch the app host, there's no warning displayed to the console.

Define the resource types

The *MailDev.Hosting* class library contains the resource type and extension methods for adding the resource to the app host. You should first think about the experience that you want to give developers when using your custom resource. In the case of this custom resource, you would want developers to be able to write code like the following:

C#

```csharp
var builder = DistributedApplication.CreateBuilder(args);

var maildev = builder.AddMailDev("maildev");

builder.AddProject<Projects.NewsletterService>("newsletterservice")
       .WithReference(maildev);
```

To achieve this, you need a custom resource named `MailDevResource` which implements IResourceWithConnectionString so that consumers can use it with WithReference extension to inject the connection details for the MailDev server as a connection string.

MailDev is available as a container resource, so you'll also want to derive from ContainerResource so that we can make use of various pre-existing container-focused extensions in .NET Aspire.

Replace the contents of the *Class1.cs* file in the `MailDev.Hosting` project, and rename the file to *MailDevResource.cs* with the following code:

C#

```
// For ease of discovery, resource types should be placed in
// the Aspire.Hosting.ApplicationModel namespace. If there is
// likelihood of a conflict on the resource name consider using
// an alternative namespace.
namespace Aspire.Hosting.ApplicationModel;

public sealed class MailDevResource(string name) : ContainerResource(name),
IResourceWithConnectionString
{
    // Constants used to refer to well known-endpoint names, this is
specific
    // for each resource type. MailDev exposes an SMTP endpoint and a HTTP
    // endpoint.
    internal const string SmtpEndpointName = "smtp";
    internal const string HttpEndpointName = "http";

    // An EndpointReference is a core .NET Aspire type used for keeping
    // track of endpoint details in expressions. Simple literal values
cannot
    // be used because endpoints are not known until containers are
launched.
    private EndpointReference? _smtpReference;

    public EndpointReference SmtpEndpoint =>
        _smtpReference ??= new(this, SmtpEndpointName);

    // Required property on IResourceWithConnectionString. Represents a
connection
    // string that applications can use to access the MailDev server. In
this case
    // the connection string is composed of the SmtpEndpoint endpoint
reference.
    public ReferenceExpression ConnectionStringExpression =>
        ReferenceExpression.Create(
            $"smtp://{SmtpEndpoint.Property(EndpointProperty.Host)}:
{SmtpEndpoint.Property(EndpointProperty.Port)}"
        );
}
```

In the preceding custom resource, the EndpointReference and ReferenceExpression are examples of several types which implement a collection of interfaces, such as IManifestExpressionProvider, IValueProvider, and IValueWithReferences. For more information about these types and their role in .NET Aspire, see technical details.

Define the resource extensions

To make it easy for developers to use the custom resource an extension method named `AddMailDev` needs to be added to the *MailDev.Hosting* project. The `AddMailDev` extension method is responsible for configuring the resource so it can start successfully as a container.

Add the following code to a new file named *MailDevResourceBuilderExtensions.cs* in the *MailDev.Hosting* project:

C#

```
using Aspire.Hosting.ApplicationModel;

// Put extensions in the Aspire.Hosting namespace to ease discovery as
referencing
// the .NET Aspire hosting package automatically adds this namespace.
namespace Aspire.Hosting;

public static class MailDevResourceBuilderExtensions
{
    /// <summary>
    /// Adds the <see cref="MailDevResource"/> to the given
    /// <paramref name="builder"/> instance. Uses the "2.1.0" tag.
    /// </summary>
    /// <param name="builder">The <see
cref="IDistributedApplicationBuilder"/>.</param>
    /// <param name="name">The name of the resource.</param>
    /// <param name="httpPort">The HTTP port.</param>
    /// <param name="smtpPort">The SMTP port.</param>
    /// <returns>
    /// An <see cref="IResourceBuilder{MailDevResource}"/> instance that
    /// represents the added MailDev resource.
    /// </returns>
    public static IResourceBuilder<MailDevResource> AddMailDev(
        this IDistributedApplicationBuilder builder,
        string name,
        int? httpPort = null,
        int? smtpPort = null)
    {
        // The AddResource method is a core API within .NET Aspire and is
        // used by resource developers to wrap a custom resource in an
        // IResourceBuilder<T> instance. Extension methods to customize
        // the resource (if any exist) target the builder interface.
        var resource = new MailDevResource(name);

        return builder.AddResource(resource)
                      .WithImage(MailDevContainerImageTags.Image)
                      .WithImageRegistry(MailDevContainerImageTags.Registry)
                      .WithImageTag(MailDevContainerImageTags.Tag)
                      .WithHttpEndpoint(
                          targetPort: 1080,
                          port: httpPort,
                          name: MailDevResource.HttpEndpointName)
```

```
                        .WithEndpoint(
                            targetPort: 1025,
                            port: smtpPort,
                            name: MailDevResource.SmtpEndpointName);
    }
}

// This class just contains constant strings that can be updated
periodically
// when new versions of the underlying container are released.
internal static class MailDevContainerImageTags
{
    internal const string Registry = "docker.io";

    internal const string Image = "maildev/maildev";

    internal const string Tag = "2.1.0";
}
```

Validate custom integration inside the app host

Now that the basic structure for the custom resource is complete it's time to test it in a real AppHost project. Open the *Program.cs* file in the *MailDevResource.AppHost* project and update it with the following code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var maildev = builder.AddMailDev("maildev");

builder.Build().Run();
```

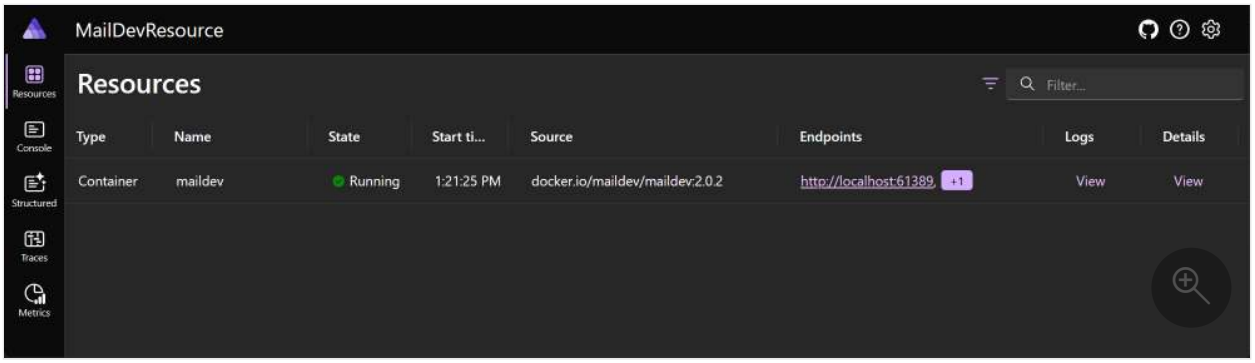
After updating the *Program.cs* file, launch the app host project and open the dashboard:

.NET CLI

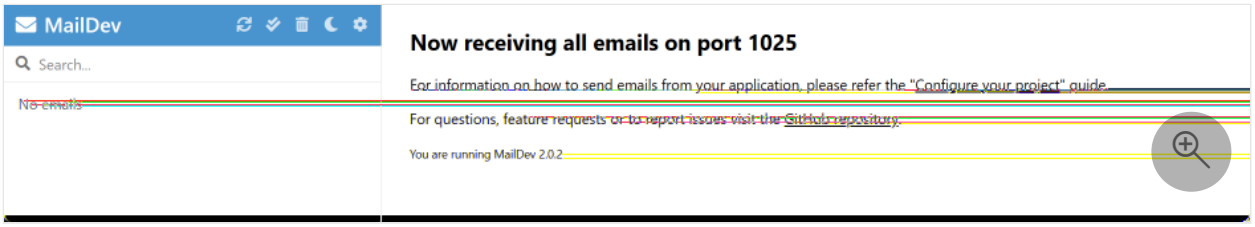
```
dotnet run --project
./MailDevResource.AppHost/MailDevResource.AppHost.csproj
```

After a few moments the dashboard shows that the `maildev` resource is running and a hyperlink will be available that navigates to the MailDev web app, which shows the content of each e-mail that your app sends.

The .NET Aspire dashboard should look similar to the following:



The MailDev web app should look similar to the following:



Add a .NET service project to the app host for testing

Once .NET Aspire can successfully launch the MailDev integration, it's time to consume the connection information for MailDev within a .NET project. In .NET Aspire it's common for there to be a *hosting package* and one or more *component packages*. For example consider:

- **Hosting package**: Used to represent resources within the app model.
  - `Aspire.Hosting.Redis`
- **Component packages**: Used to configure and consume client libraries.
  - `Aspire.StackExchange.Redis`
  - `Aspire.StackExchange.Redis.DistributedCaching`
  - `Aspire.StackExchange.Redis.OutputCaching`

In the case of the MailDev resource, the .NET platform already has a simple mail transfer protocol (SMTP) client in the form of SmtpClient. In this example you use this existing API for the sake of simplicity, although other resource types may benefit from custom integration libraries to assist developers.

In order to test the end-to-end scenario, you need a .NET project which we can inject the connection information into for the MailDev resource. Add a Web API project:

1. Create a new .NET project named *MailDevResource.NewsletterService*.

   .NET CLI

```
dotnet new webapi --use-minimal-apis -o
MailDevResource.NewsletterService
```

2. Add a reference to the *MailDev.Hosting* project.

.NET CLI

```
dotnet add
./MailDevResource.NewsletterService/MailDevResource.NewsletterService.c
sproj reference ./MailDev.Hosting/MailDev.Hosting.csproj
```

3. Add a reference to the *MailDevResource.AppHost* project.

.NET CLI

```
dotnet add ./MailDevResource.AppHost/MailDevResource.AppHost.csproj
reference
./MailDevResource.NewsletterService/MailDevResource.NewsletterService.c
sproj
```

4. Add the new project to the solution file.

.NET CLI

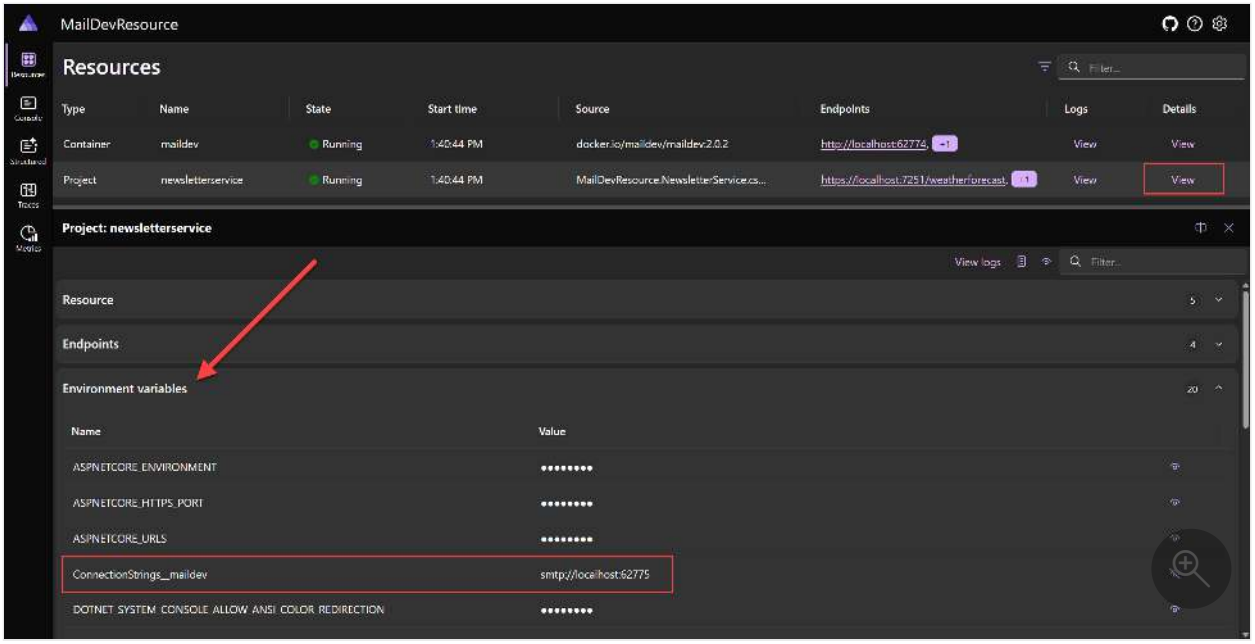
```
dotnet sln ./MailDevResource.sln add
./MailDevResource.NewsletterService/MailDevResource.NewsletterService.c
sproj
```

After the project has been added and references have been updated, open the *Program.cs* of the *MailDevResource.AppHost.csproj* project, and update the source file to look like the following:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var maildev = builder.AddMailDev("maildev");

builder.AddProject<Projects.MailDevResource_NewsletterService>
("newsletterservice")
       .WithReference(maildev);

builder.Build().Run();
```

After updating the *Program.cs* file, launch the app host again. Then verify that the Newsletter Service started and that the environment variable `ConnectionStrings__maildev` was added to the process. From the **Resources** page, find the `newsletterservice` row, and select the **View** link on the **Details** column:



The preceding screenshot shows the environment variables for the `newsletterservice` project. The `ConnectionStrings__maildev` environment variable is the connection string that was injected into the project by the `maildev` resource.

Use connection string to send messages

To use the SMTP connection details that were injected into the newsletter service project, you inject an instance of SmtpClient into the dependency injection container as a singleton. Add the following code to the *Program.cs* file in the *MailDevResource.NewsletterService* project to set up the singleton service. In the `Program` class, immediately following the `// Add services to the container` comment, add the following code:

C#

```
builder.Services.AddSingleton<SmtpClient>(sp =>
{
    var smtpUri = new
Uri(builder.Configuration.GetConnectionString("maildev")!);

    var smtpClient = new SmtpClient(smtpUri.Host, smtpUri.Port);

    return smtpClient;
});
```

> **Tip**
>
> This code snippet relies on the official `SmtpClient`, however; this type is obsolete on some platforms and not recommended on others. For a more modern approach using MailKit, see Create custom .NET Aspire client integrations.

To test the client, add two simple `subscribe` and `unsubscribe` POST methods to the newsletter service. Add the following code replacing the "weatherforecast" `MapGet` call in the *Program.cs* file of the *MailDevResource.NewsletterService* project to set up the ASP.NET Core routes:

C#

```
app.MapPost("/subscribe", async (SmtpClient smtpClient, string email) =>
{
    using var message = new MailMessage("newsletter@yourcompany.com", email)
    {
        Subject = "Welcome to our newsletter!",
        Body = "Thank you for subscribing to our newsletter!"
    };

    await smtpClient.SendMailAsync(message);
});

app.MapPost("/unsubscribe", async (SmtpClient smtpClient, string email) =>
{
    using var message = new MailMessage("newsletter@yourcompany.com", email)
    {
        Subject = "You are unsubscribed from our newsletter!",
        Body = "Sorry to see you go. We hope you will come back soon!"
    };

    await smtpClient.SendMailAsync(message);
});
```

> **Tip**
>
> Remember to reference the `System.Net.Mail` and `Microsoft.AspNetCore.Mvc` namespaces in *Program.cs* if your code editor doesn't automatically add them.

Once the *Program.cs* file is updated, launch the app host and use your browser, or `curl` to hit the following URLs (alternatively if you're using Visual Studio you can use `.http` files):

HTTP

```
POST /subscribe?email=test@test.com HTTP/1.1
Host: localhost:7251
Content-Type: application/json
```

To use this API, you can use `curl` to send the request. The following `curl` command sends an HTTP `POST` request to the `subscribe` endpoint, and it expects an `email` query string value to subscribe to the newsletter. The `Content-Type` header is set to `application/json` to indicate that the request body is in JSON format.:

Windows

PowerShell

```
curl -H @{ ContentType = "application/json" } -Method POST
https://localhost:7251/subscribe?email=test@test.com
```

The next API is the `unsubscribe` endpoint. This endpoint is used to unsubscribe from the newsletter.

HTTP

```
POST /unsubscribe?email=test@test.com HTTP/1.1
Host: localhost:7251
Content-Type: application/json
```

To unsubscribe from the newsletter, you can use the following `curl` command, passing an `email` parameter to the `unsubscribe` endpoint as a query string:

Windows

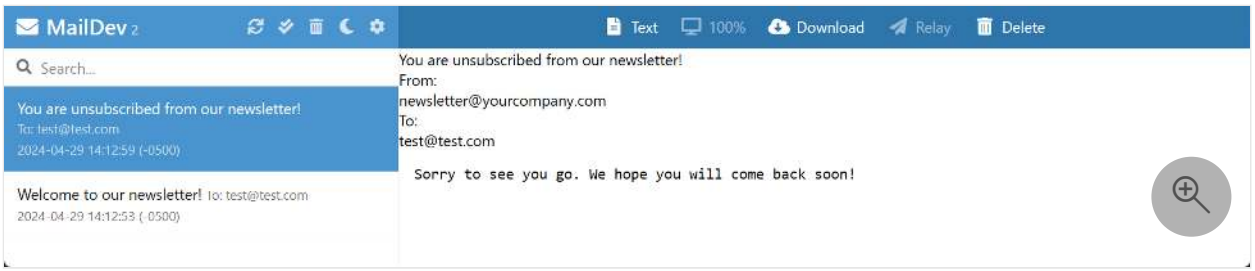
PowerShell

```
curl -H @{ ContentType = "application/json" } -Method POST
https://localhost:7251/unsubscribe?email=test@test.com
```

Tip

Make sure that you replace the `https://localhost:7251` with the correct localhost port (the URL of the app host that you are running).

If those API calls return a successful response (HTTP 200, Ok) then you should be able to select on the `maildev` resource the dashboard and the MailDev UI will show the emails that have been sent to the SMTP endpoint.



Technical details

In the following sections, various technical details are discussed which are important to understand when developing custom resources for .NET Aspire.

Secure networking

In this example, the MailDev resource is a container resource which is exposed to the host machine over HTTP and SMTP. The MailDev resource is a development tool and isn't intended for production use. To instead use HTTPS, see MailDev: Configure HTTPS.

When developing custom resources that expose network endpoints, it's important to consider the security implications of the resource. For example, if the resource is a database, it's important to ensure that the database is secure and that the connection string isn't exposed to the public internet.

The `ReferenceExpression` and `EndpointReference` type

In the preceding code, the `MailDevResource` had two properties:

- `SmtpEndpoint`: EndpointReference type.
- `ConnectionStringExpression`: ReferenceExpression type.

These types are among several which are used throughout .NET Aspire to represent configuration data, which isn't finalized until the .NET Aspire project is either run or published to the cloud via a tool such as Azure Developer CLI (azd).

The fundamental problem that these types help to solve, is deferring resolution of concrete configuration information until *all* the information is available.

For example, the `MailDevResource` exposes a property called `ConnectionStringExpression` as required by the IResourceWithConnectionString interface. The type of the property is ReferenceExpression and is created by passing in an interpolated string to the Create method.

C#

```
public ReferenceExpression ConnectionStringExpression =>
    ReferenceExpression.Create(
        $"smtp://{SmtpEndpoint.Property(EndpointProperty.Host)}:
{SmtpEndpoint.Property(EndpointProperty.Port)}"
    );
```

The signature for the Create method is as follows:

C#

```
public static ReferenceExpression Create(
    in ExpressionInterpolatedStringHandler handler)
```

This isn't a regular String argument. The method makes use of the interpolated string handler pattern, to capture the interpolated string template and the values referenced within it to allow for custom processing. In the case of .NET Aspire, these details are captured in a ReferenceExpression which can be evaluated as each value referenced in the interpolated string becomes available.

Here's how the flow of execution works:

1. A resource which implements IResourceWithConnectionString is added to the model (for example, `AddMailDev(...)`).
2. The `IResourceBuilder<MailDevResource>` is passed to the WithReference which has a special overload for handling IResourceWithConnectionString implementors.
3. The `WithReference` wraps the resource in a ConnectionStringReference instance and the object is captured in a EnvironmentCallbackAnnotation which is evaluated after the .NET Aspire project is built and starts running.
4. As the process that references the connection string starts .NET Aspire starts evaluating the expression. It first gets the ConnectionStringReference and calls IValueProvider.GetValueAsync.
5. The `GetValueAsync` method gets the value of the ConnectionStringExpression property to get the ReferenceExpression instance.
6. The IValueProvider.GetValueAsync method then calls GetValueAsync to process the previously captured interpolated string.

7. Because the interpolated string contains references to other reference types such as EndpointReference they're also evaluated and real value substituted (which at this time is now available).

Manifest publishing

The IManifestExpressionProvider interface is designed to solve the problem of sharing connection information between resources at deployment. The solution for this particular problem is described in the .NET Aspire inner-loop networking overview. Similarly to local development, many of the values are necessary to configure the app, yet they can't be determined until the app is being deployed via a tool, such as `azd` (Azure Developer CLI).

To solve this problem .NET Aspire produces a manifest file which `azd` and other deployment tools interpret. Rather than specifying concrete values for connection information between resources an expression syntax is used which deployment tools evaluate. Generally the manifest file isn't visible to developers but it's possible to generate one for manual inspection. The command below can be used on the app host to produce a manifest.

.NET CLI

```
dotnet run --project MailDevResource.AppHost/MailDevResource.AppHost.csproj
-- --publisher manifest --output-path aspire-manifest.json
```

This command produces a manifest file like the following:

JSON

```
{
  "resources": {
    "maildev": {
      "type": "container.v0",
      "connectionString": "smtp://{maildev.bindings.smtp.host}:
{maildev.bindings.smtp.port}",
      "image": "docker.io/maildev/maildev:2.1.0",
      "bindings": {
        "http": {
          "scheme": "http",
          "protocol": "tcp",
          "transport": "http",
          "targetPort": 1080
        },
        "smtp": {
          "scheme": "tcp",
          "protocol": "tcp",
```

```
          "transport": "tcp",
          "targetPort": 1025
        }
      }
    },
    "newsletterservice": {
      "type": "project.v0",
      "path":
"../MailDevResource.NewsletterService/MailDevResource.NewsletterService.cspr
oj",
      "env": {
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EXCEPTION_LOG_ATTRIBUTES":
"true",
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EVENT_LOG_ATTRIBUTES": "true",
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_RETRY": "in_memory",
        "ASPNETCORE_FORWARDEDHEADERS_ENABLED": "true",
        "ConnectionStrings__maildev": "{maildev.connectionString}"
      },
      "bindings": {
        "http": {
          "scheme": "http",
          "protocol": "tcp",
          "transport": "http"
        },
        "https": {
          "scheme": "https",
          "protocol": "tcp",
          "transport": "http"
        }
      }
    }
  }
}
```

Because the `MailDevResource` implements IResourceWithConnectionString the manifest publishing logic in .NET Aspire knows that even though `MailDevResource` is a container resource, it also needs a `connectionString` field. The `connectionString` field references other parts of the `maildev` resource in the manifest to produce the final string:

JSON

```
{
    // ... other content omitted.
    "connectionString": "smtp://{maildev.bindings.smtp.host}:
{maildev.bindings.smtp.port}"
}
```

.NET Aspire knows how to form this string because it looks at ConnectionStringExpression and builds up the final string via the

IManifestExpressionProvider interface (in much the same way as the IValueProvider interface is used).

The `MailDevResource` automatically gets included in the manifest because it's derived from ContainerResource. Resource authors can choose to suppress outputting content to the manifest by using the ExcludeFromManifest extension method on the resource builder.

C#

```
public static IResourceBuilder<MailDevResource> AddMailDev(
    this IDistributedApplicationBuilder builder,
    string name,
    int? httpPort = null,
    int? smtpPort = null)
{
    var resource = new MailDevResource(name);

    return builder.AddResource(resource)
                  .WithImage(MailDevContainerImageTags.Image)
                  .WithImageRegistry(MailDevContainerImageTags.Registry)
                  .WithImageTag(MailDevContainerImageTags.Tag)
                  .WithHttpEndpoint(
                      targetPort: 1080,
                      port: httpPort,
                      name: MailDevResource.HttpEndpointName)
                  .WithEndpoint(
                      targetPort: 1025,
                      port: smtpPort,
                      name: MailDevResource.SmtpEndpointName)
                  .ExcludeFromManifest(); // This line was added
}
```

Careful consideration should be given as to whether the resource should be present in the manifest, or whether it should be suppressed. If the resource is being added to the manifest, it should be configured in such a way that it's safe and secure to use.

Summary

In the custom resource tutorial, you learned how to create a custom .NET Aspire resource which uses an existing containerized application (MailDev). You then used that to improve the local development experience by making it easy to test e-mail capabilities that might be used within an app. These learnings can be applied to building out other custom resources that can be used in .NET Aspire-based applications. This specific example didn't include any custom integrations, but it's possible to build out custom integrations to make it easier for developers to use the resource. In this

scenario you were able to rely on the existing `SmtpClient` class in the .NET platform to send e-mails.

Next steps

Create custom .NET Aspire client integrations

Create custom .NET Aspire client integrations

Article • 09/24/2024

This article is a continuation of the Create custom .NET Aspire hosting integrations article. It guides you through creating a .NET Aspire client integration that uses MailKit to send emails. This integration is then added into the Newsletter app you previously built. The previous example omitted the creation of a client integration and instead relied on the existing .NET `SmtpClient`. It's best to use MailKit's `SmtpClient` over the official .NET `SmtpClient` for sending emails, as it's more modern and supports more features/protocols. For more information, see .NET SmtpClient: Remarks.

Prerequisites

If you're following along, you should have a Newsletter app from the steps in the Create custom .NET Aspire hosting integration article.

> **Tip**
>
> This article is inspired by existing .NET Aspire integrations, and based on the team's official guidance. There are places where said guidance varies, and it's important to understand the reasoning behind the differences. For more information, see **.NET Aspire integration requirements**.

Create library for integration

.NET Aspire integrations are delivered as NuGet packages, but in this example, it's beyond the scope of this article to publish a NuGet package. Instead, you create a class library project that contains the integration and reference it as a project. .NET Aspire integration packages are intended to wrap a client library, such as MailKit, and provide production-ready telemetry, health checks, configurability, and testability. Let's start by creating a new class library project.

1. Create a new class library project named `MailKit.Client` in the same directory as the *MailDevResource.sln* from the previous article.

   .NET CLI

```
dotnet new classlib -o MailKit.Client
```

2. Add the project to the solution.

.NET CLI

```
dotnet sln ./MailDevResource.sln add
MailKit.Client/MailKit.Client.csproj
```

The next step is to add all the NuGet packages that the integration relies on. Rather than having you add each package one-by-one from the .NET CLI, it's likely easier to copy and paste the following XML into the *MailKit.Client.csproj* file.

XML

```
<ItemGroup>
  <PackageReference Include="MailKit" Version="4.11.0" />
  <PackageReference Include="Microsoft.Extensions.Configuration.Binder"
Version="9.0.3" />
  <PackageReference Include="Microsoft.Extensions.Resilience"
Version="9.3.0" />
  <PackageReference Include="Microsoft.Extensions.Hosting.Abstractions"
Version="9.0.3" />
  <PackageReference Include="Microsoft.Extensions.Diagnostics.HealthChecks"
Version="9.0.3" />
  <PackageReference Include="OpenTelemetry.Extensions.Hosting"
Version="1.11.2" />
</ItemGroup>
```

Define integration settings

Whenever you're creating a .NET Aspire integration, it's best to understand the client library that you're mapping to. With MailKit, you need to understand the configuration settings that are required to connect to a Simple Mail Transfer Protocol (SMTP) server. But it's also important to understand if the library has support for *health checks*, *tracing* and *metrics*. MailKit supports *tracing* and *metrics*, through its Telemetry.SmtpClient class. When adding *health checks*, you should use any established or existing health checks where possible. Otherwise, you might consider implementing your own in the integration. Add the following code to the `MailKit.Client` project in a file named *MailKitClientSettings.cs*:

C#

```
using System.Data.Common;

namespace MailKit.Client;

/// <summary>
/// Provides the client configuration settings for connecting MailKit to an
SMTP server.
/// </summary>
public sealed class MailKitClientSettings
{
    internal const string DefaultConfigSectionName = "MailKit:Client";

    /// <summary>
    /// Gets or sets the SMTP server <see cref="Uri"/>.
    /// </summary>
    /// <value>
    /// The default value is <see langword="null"/>.
    /// </value>
    public Uri? Endpoint { get; set; }

    /// <summary>
    /// Gets or sets a boolean value that indicates whether the database
health check is disabled or not.
    /// </summary>
    /// <value>
    /// The default value is <see langword="false"/>.
    /// </value>
    public bool DisableHealthChecks { get; set; }

    /// <summary>
    /// Gets or sets a boolean value that indicates whether the
OpenTelemetry tracing is disabled or not.
    /// </summary>
    /// <value>
    /// The default value is <see langword="false"/>.
    /// </value>
    public bool DisableTracing { get; set; }

    /// <summary>
    /// Gets or sets a boolean value that indicates whether the
OpenTelemetry metrics are disabled or not.
    /// </summary>
    /// <value>
    /// The default value is <see langword="false"/>.
    /// </value>
    public bool DisableMetrics { get; set; }

    internal void ParseConnectionString(string? connectionString)
    {
        if (string.IsNullOrWhiteSpace(connectionString))
        {
            throw new InvalidOperationException($"""
                    ConnectionString is missing.
                    It should be provided in 'ConnectionStrings:
```

```
<connectionName>'
                    or '{DefaultConfigSectionName}:Endpoint' key.'
                    configuration section.
                    """);
        }

        if (Uri.TryCreate(connectionString, UriKind.Absolute, out var uri))
        {
            Endpoint = uri;
        }
        else
        {
            var builder = new DbConnectionStringBuilder
            {
                ConnectionString = connectionString
            };

            if (builder.TryGetValue("Endpoint", out var endpoint) is false)
            {
                throw new InvalidOperationException($"""
                        The 'ConnectionStrings:<connectionName>' (or
'Endpoint' key in
                        '{DefaultConfigSectionName}') is missing.
                        """);
            }

            if (Uri.TryCreate(endpoint.ToString(), UriKind.Absolute, out
uri) is false)
            {
                throw new InvalidOperationException($"""
                        The 'ConnectionStrings:<connectionName>' (or
'Endpoint' key in
                        '{DefaultConfigSectionName}') isn't a valid URI.
                        """);
            }

            Endpoint = uri;
        }
    }
}
```

The preceding code defines the `MailKitClientSettings` class with:

- `Endpoint` property that represents the connection string to the SMTP server.
- `DisableHealthChecks` property that determines whether health checks are enabled.
- `DisableTracing` property that determines whether tracing is enabled.
- `DisableMetrics` property that determines whether metrics are enabled.

Parse connection string logic

The settings class also contains a `ParseConnectionString` method that parses the connection string into a valid `Uri`. The configuration is expected to be provided in the following format:

- `ConnectionStrings:<connectionName>`: The connection string to the SMTP server.
- `MailKit:Client:ConnectionString`: The connection string to the SMTP server.

If neither of these values are provided, an exception is thrown.

The goal of .NET Aspire integrations is to expose the underlying client library to consumers through dependency injection. With MailKit and for this example, the `SmtpClient` class is what you want to expose. You're not wrapping any functionality, but rather mapping configuration settings to an `SmtpClient` class. It's common to expose both standard and keyed-service registrations for integrations. Standard registrations are used when there's only one instance of a service, and keyed-service registrations are used when there are multiple instances of a service. Sometimes, to achieve multiple registrations of the same type you use a factory pattern. Add the following code to the `MailKit.Client` project in a file named *MailKitClientFactory.cs*:

```
C#
```

```
    /// <remarks>
    /// Since both the connection and authentication are considered
expensive operations,
    /// the <see cref="ISmtpClient"/> returned is intended to be used for
the duration of a request
    /// (registered as 'Scoped') and is automatically disposed of.
    /// </remarks>
    public async Task<ISmtpClient> GetSmtpClientAsync(
        CancellationToken cancellationToken = default)
    {
        var client = new SmtpClient();
        try
        {
            if (settings.Endpoint is not null)
            {
                await client.ConnectAsync(settings.Endpoint,
cancellationToken)
                            .ConfigureAwait(false);
            }
            return client;
        }
        catch
        {
            await client.DisconnectAsync(true, cancellationToken)
            client.Dispose();
            throw;
        }
    }
}
```

The `MailKitClientFactory` class is a factory that creates an `ISmtpClient` instance based on the configuration settings. It's responsible for returning an `ISmtpClient` implementation that has an active connection to a configured SMTP server. Next, you need to expose the functionality for the consumers to register this factory with the dependency injection container. Add the following code to the `MailKit.Client` project in a file named *MailKitExtensions.cs*:

C#

```
using MailKit;
using MailKit.Client;
using MailKit.Net.Smtp;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace Microsoft.Extensions.Hosting;

/// <summary>
/// Provides extension methods for registering a <see cref="SmtpClient"/> as
a
/// scoped-lifetime service in the services provided by the <see
```

```
cref="IHostApplicationBuilder"/>.
/// </summary>
public static class MailKitExtensions
{
    /// <summary>
    /// Registers 'Scoped' <see cref="MailKitClientFactory" /> for creating
    /// connected <see cref="SmtpClient"/> instance for sending emails.
    /// </summary>
    /// <param name="builder">
    /// The <see cref="IHostApplicationBuilder" /> to read config from and
add services to.
    /// </param>
    /// <param name="connectionName">
    /// A name used to retrieve the connection string from the
ConnectionStrings configuration section.
    /// </param>
    /// <param name="configureSettings">
    /// An optional delegate that can be used for customizing options.
    /// It's invoked after the settings are read from the configuration.
    /// </param>
    public static void AddMailKitClient(
        this IHostApplicationBuilder builder,
        string connectionName,
        Action<MailKitClientSettings>? configureSettings = null) =>
        AddMailKitClient(
            builder,
            MailKitClientSettings.DefaultConfigSectionName,
            configureSettings,
            connectionName,
            serviceKey: null);

    /// <summary>
    /// Registers 'Scoped' <see cref="MailKitClientFactory" /> for creating
    /// connected <see cref="SmtpClient"/> instance for sending emails.
    /// </summary>
    /// <param name="builder">
    /// The <see cref="IHostApplicationBuilder" /> to read config from and
add services to.
    /// </param>
    /// <param name="name">
    /// The name of the component, which is used as the <see
cref="ServiceDescriptor.ServiceKey"/> of the
    /// service and also to retrieve the connection string from the
ConnectionStrings configuration section.
    /// </param>
    /// <param name="configureSettings">
    /// An optional method that can be used for customizing options. It's
invoked after the settings are
    /// read from the configuration.
    /// </param>
    public static void AddKeyedMailKitClient(
        this IHostApplicationBuilder builder,
        string name,
        Action<MailKitClientSettings>? configureSettings = null)
    {
```

```
        ArgumentNullException.ThrowIfNull(name);

        AddMailKitClient(
            builder,
            $"{MailKitClientSettings.DefaultConfigSectionName}:{name}",
            configureSettings,
            connectionName: name,
            serviceKey: name);
    }

    private static void AddMailKitClient(
        this IHostApplicationBuilder builder,
        string configurationSectionName,
        Action<MailKitClientSettings>? configureSettings,
        string connectionName,
        object? serviceKey)
    {
        ArgumentNullException.ThrowIfNull(builder);

        var settings = new MailKitClientSettings();

        builder.Configuration
               .GetSection(configurationSectionName)
               .Bind(settings);

        if (builder.Configuration.GetConnectionString(connectionName) is
string connectionString)
        {
            settings.ParseConnectionString(connectionString);
        }

        configureSettings?.Invoke(settings);

        if (serviceKey is null)
        {
            builder.Services.AddScoped(CreateMailKitClientFactory);
        }
        else
        {
            builder.Services.AddKeyedScoped(serviceKey, (sp, key) =>
CreateMailKitClientFactory(sp));
        }

        MailKitClientFactory CreateMailKitClientFactory(IServiceProvider _)
        {
            return new MailKitClientFactory(settings);
        }

        if (settings.DisableHealthChecks is false)
        {
            builder.Services.AddHealthChecks()
                .AddCheck<MailKitHealthCheck>(
                    name: serviceKey is null ? "MailKit" :
$"MailKit_{connectionName}",
                    failureStatus: default,
```

```
                    tags: []);
        }

        if (settings.DisableTracing is false)
        {
            builder.Services.AddOpenTelemetry()
                .WithTracing(
                    traceBuilder => traceBuilder.AddSource(
                        Telemetry.SmtpClient.ActivitySourceName));
        }

        if (settings.DisableMetrics is false)
        {
            // Required by MailKit to enable metrics
            Telemetry.SmtpClient.Configure();

            builder.Services.AddOpenTelemetry()
                .WithMetrics(
                    metricsBuilder => metricsBuilder.AddMeter(
                        Telemetry.SmtpClient.MeterName));
        }
    }
}
```

The preceding code adds two extension methods on the `IHostApplicationBuilder` type, one for the standard registration of MailKit and another for keyed-registration of MailKit.

> **Tip**
>
> Extension methods for .NET Aspire integrations should extend the `IHostApplicationBuilder` type and follow the `Add<MeaningfulName>` naming convention where the `<MeaningfulName>` is the type or functionality you're adding. For this article, the `AddMailKitClient` extension method is used to add the MailKit client. It's likely more in-line with the official guidance to use `AddMailKitSmtpClient` instead of `AddMailKitClient`, since this only registers the `SmtpClient` and not the entire MailKit library.

Both extensions ultimately rely on the private `AddMailKitClient` method to register the `MailKitClientFactory` with the dependency injection container as a scoped service. The reason for registering the `MailKitClientFactory` as a scoped service is because the connection operations are considered expensive and should be reused within the same scope where possible. In other words, for a single request, the same `ISmtpClient` instance should be used. The factory holds on to the instance of the `SmtpClient` that it creates and disposes of it.

Configuration binding

One of the first things that the private implementation of the `AddMailKitClient` methods does, is to bind the configuration settings to the `MailKitClientSettings` class. The settings class is instantiated and then `Bind` is called with the specific section of configuration. Then the optional `configureSettings` delegate is invoked with the current settings. This allows the consumer to further configure the settings, ensuring that manual code settings are honored over configuration settings. After that, depending on whether the `serviceKey` value was provided, the `MailKitClientFactory` should be registered with the dependency injection container as either a standard or keyed service.

> **ⓘ Important**
>
> It's intentional that the `implementationFactory` overload is called when registering services. The `CreateMailKitClientFactory` method throws when the configuration is invalid. This ensures that creation of the `MailKitClientFactory` is deferred until it's needed and it prevents the app from erroring out before logging is available.

The registration of health checks, and telemetry are described in a bit more detail in the following sections.

Add health checks

Health checks are a way to monitor the health of an integration. With MailKit, you can check if the connection to the SMTP server is healthy. Add the following code to the `MailKit.Client` project in a file named *MailKitHealthCheck.cs*:

C#

```
using Microsoft.Extensions.Diagnostics.HealthChecks;

namespace MailKit.Client;

internal sealed class MailKitHealthCheck(MailKitClientFactory factory) :
IHealthCheck
{
    public async Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context,
        CancellationToken cancellationToken = default)
    {
        try
        {
            // The factory connects (and authenticates).
            _ = await factory.GetSmtpClientAsync(cancellationToken);
```

```
            return HealthCheckResult.Healthy();
        }
        catch (Exception ex)
        {
            return HealthCheckResult.Unhealthy(exception: ex);
        }
    }
}
```

The preceding health check implementation:

- Implements the `IHealthCheck` interface.
- Accepts the `MailKitClientFactory` as a primary constructor parameter.
- Satisfies the `CheckHealthAsync` method by:
  - Attempting to get an `ISmtpClient` instance from the `factory`. If successful, it returns `HealthCheckResult.Healthy`.
  - If an exception is thrown, it returns `HealthCheckResult.Unhealthy`.

As previously shared in the registration of the `MailKitClientFactory`, the `MailKitHealthCheck` is conditionally registered with the `IHeathChecksBuilder`:

C#

```
if (settings.DisableHealthChecks is false)
{
    builder.Services.AddHealthChecks()
        .AddCheck<MailKitHealthCheck>(
            name: serviceKey is null ? "MailKit" :
$"MailKit_{connectionName}",
            failureStatus: default,
            tags: []);
}
```

The consumer could choose to omit health checks by setting the `DisableHealthChecks` property to `true` in the configuration. A common pattern for integrations is to have optional features and .NET Aspire integrations strongly encourages these types of configurations. For more information on health checks and a working sample that includes a user interface, see .NET Aspire ASP.NET Core HealthChecksUI sample.

Wire up telemetry

As a best practice, the MailKit client library exposes telemetry. .NET Aspire can take advantage of this telemetry and display it in the .NET Aspire dashboard. Depending on

whether or not tracing and metrics are enabled, telemetry is wired up as shown in the following code snippet:

C#

```
if (settings.DisableTracing is false)
{
    builder.Services.AddOpenTelemetry()
        .WithTracing(
            traceBuilder => traceBuilder.AddSource(
                Telemetry.SmtpClient.ActivitySourceName));
}

if (settings.DisableMetrics is false)
{
    // Required by MailKit to enable metrics
    Telemetry.SmtpClient.Configure();

    builder.Services.AddOpenTelemetry()
        .WithMetrics(
            metricsBuilder => metricsBuilder.AddMeter(
                Telemetry.SmtpClient.MeterName));
}
```

Update the Newsletter service

With the integration library created, you can now update the Newsletter service to use the MailKit client. The first step is to add a reference to the `MailKit.Client` project. Add the *MailKit.Client.csproj* project reference to the `MailDevResource.NewsletterService` project:

.NET CLI

```
dotnet add
./MailDevResource.NewsletterService/MailDevResource.NewsletterService.csproj
reference MailKit.Client/MailKit.Client.csproj
```

Next, add a reference to the `ServiceDefaults` project:

.NET CLI

```
dotnet add
./MailDevResource.NewsletterService/MailDevResource.NewsletterService.csproj
reference
MailDevResource.ServiceDefaults/MailDevResource.ServiceDefaults.csproj
```

The final step is to replace the existing *Program.cs* file in the `MailDevResource.NewsletterService` project with the following C# code:

C#

```
using System.Net.Mail;
using MailKit.Client;
using MailKit.Net.Smtp;
using MimeKit;

var builder = WebApplication.CreateBuilder(args);

builder.AddServiceDefaults();

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

// Add services to the container.
builder.AddMailKitClient("maildev");

var app = builder.Build();

app.MapDefaultEndpoints();

// Configure the HTTP request pipeline.

app.UseSwagger();
app.UseSwaggerUI();
app.UseHttpsRedirection();

app.MapPost("/subscribe",
    async (MailKitClientFactory factory, string email) =>
{
    ISmtpClient client = await factory.GetSmtpClientAsync();

    using var message = new MailMessage("newsletter@yourcompany.com", email)
    {
        Subject = "Welcome to our newsletter!",
        Body = "Thank you for subscribing to our newsletter!"
    };

    await client.SendAsync(MimeMessage.CreateFromMailMessage(message));
});

app.MapPost("/unsubscribe",
    async (MailKitClientFactory factory, string email) =>
{
    ISmtpClient client = await factory.GetSmtpClientAsync();

    using var message = new MailMessage("newsletter@yourcompany.com", email)
    {
        Subject = "You are unsubscribed from our newsletter!",
        Body = "Sorry to see you go. We hope you will come back soon!"
    };
```

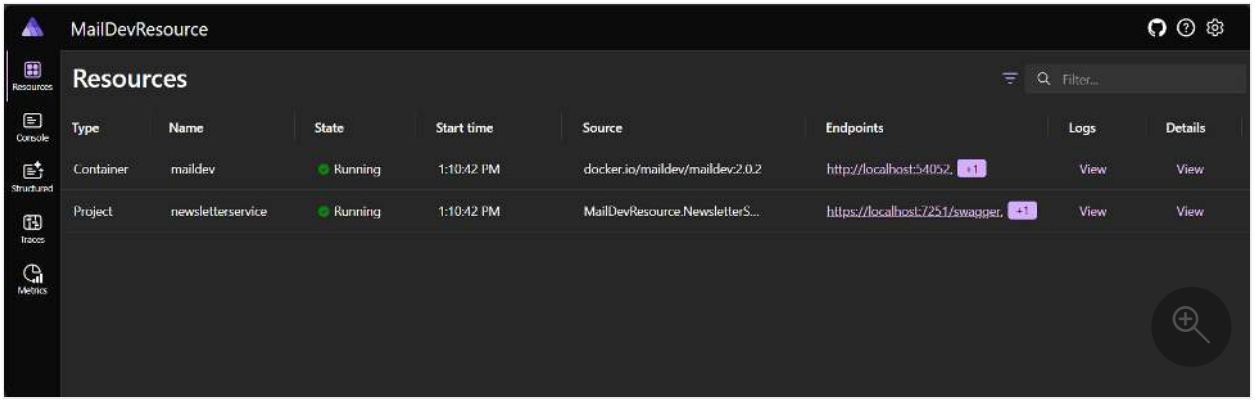
```
    await client.SendAsync(MimeMessage.CreateFromMailMessage(message));
});

app.Run();
```

The most notable changes in the preceding code are:

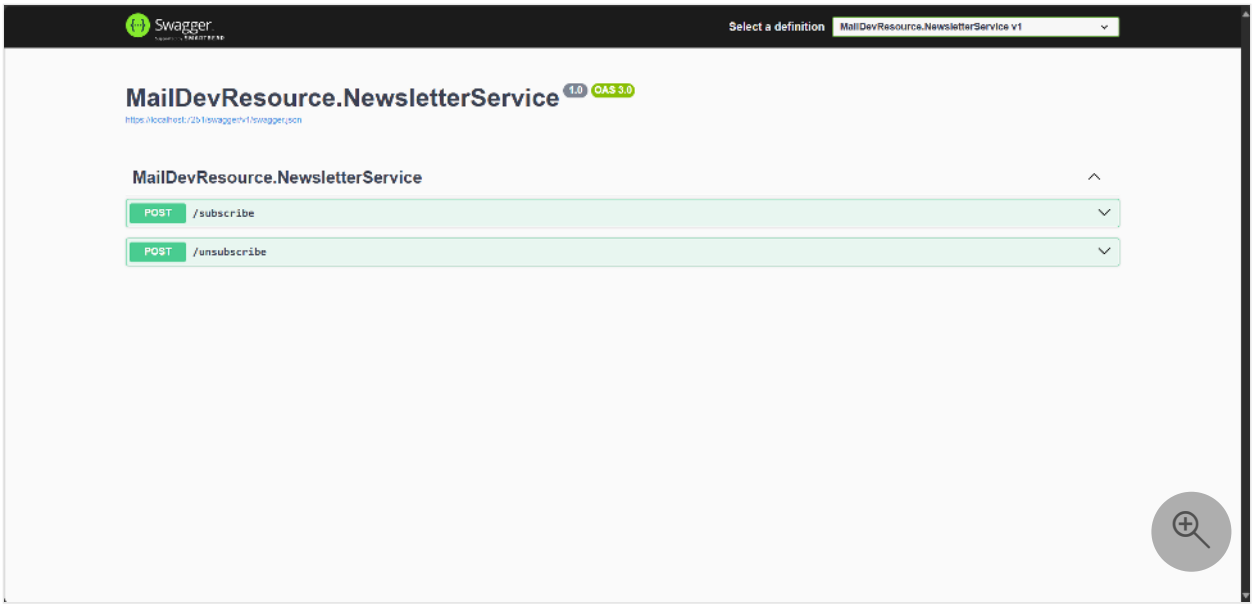
- The updated `using` statements that include the `MailKit.Client`, `MailKit.Net.Smtp`, and `MimeKit` namespaces.
- The replacement of the registration for the official .NET `SmtpClient` with the call to the `AddMailKitClient` extension method.
- The replacement of both `/subscribe` and `/unsubscribe` map post calls to instead inject the `MailKitClientFactory` and use the `ISmtpClient` instance to send the email.

Run the sample

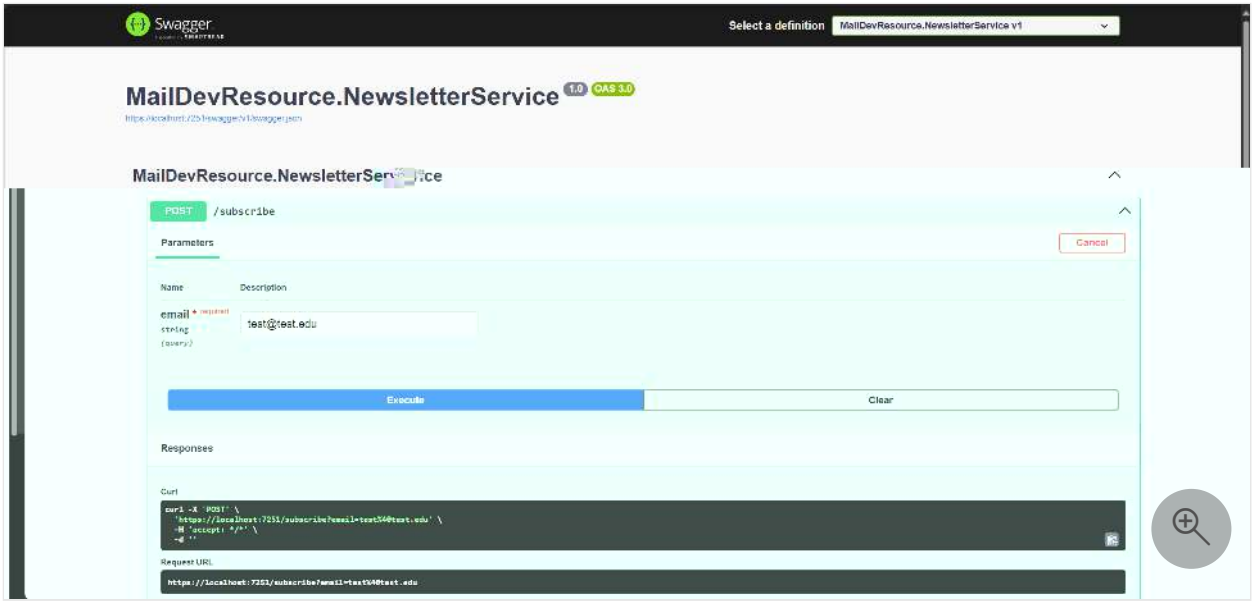
Now that you've created the MailKit client integration and updated the Newsletter service to use it, you can run the sample. From your IDE, select `F5` or run `dotnet run` from the root directory of the solution to start the application—you should see the .NET Aspire dashboard:



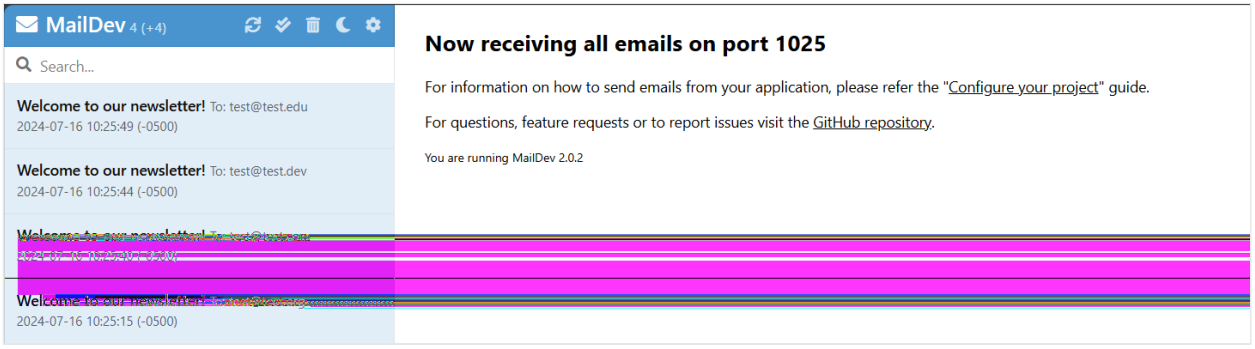
Once the application is running, navigate to the Swagger UI at https://localhost:7251/swagger and test the `/subscribe` and `/unsubscribe` endpoints. Select the down arrow to expand the endpoint:



Then select the `Try it out` button. Enter an email address, and then select the `Execute` button.



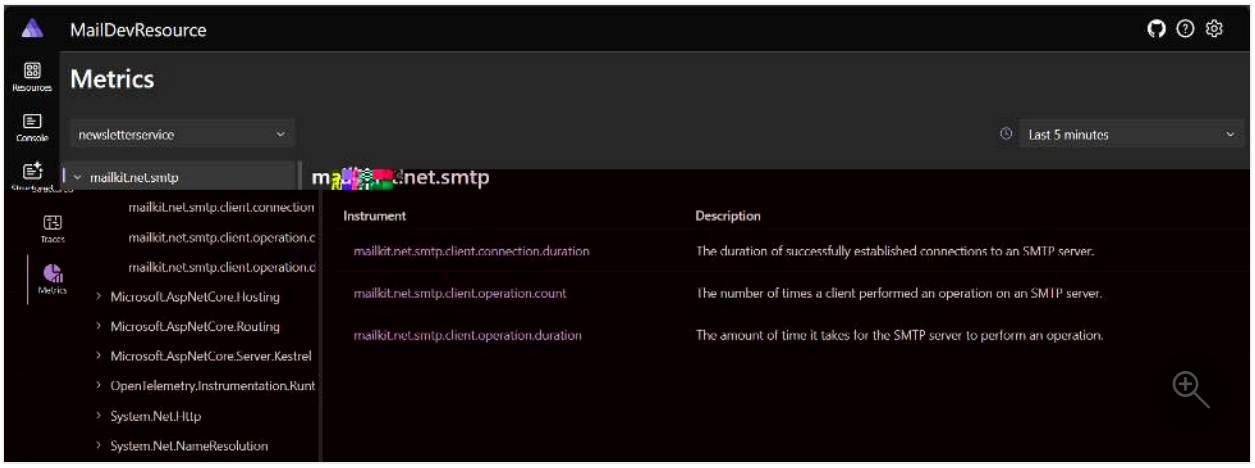
Repeat this several times, to add multiple email addresses. You should see the email sent to the MailDev inbox:



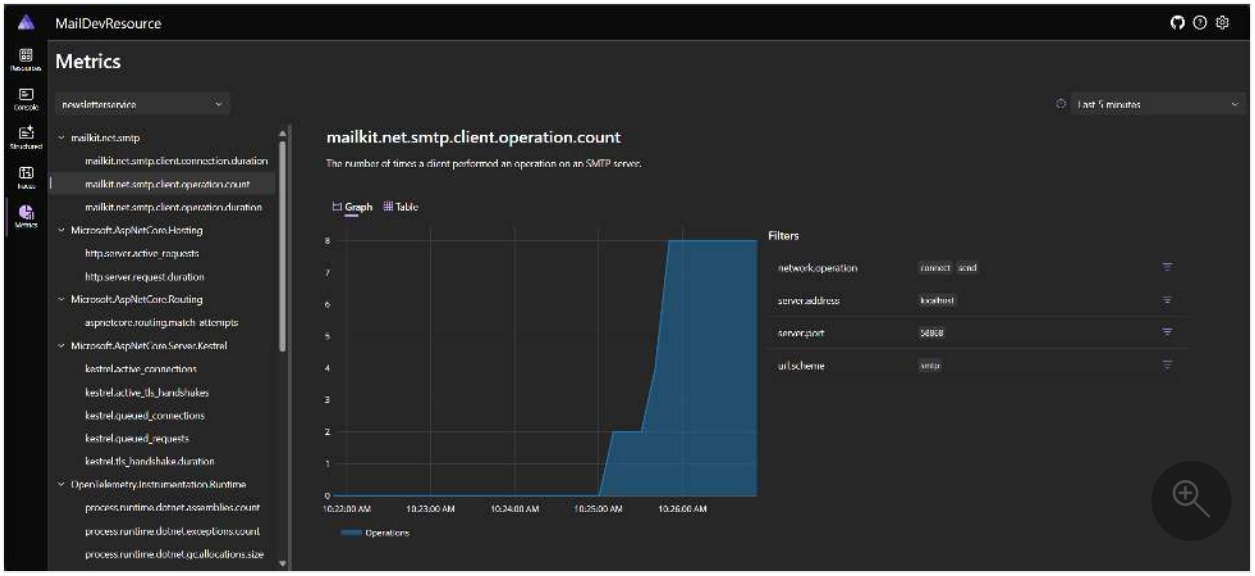
Stop the application by selecting `Ctrl` + `C` in the terminal window where the application is running, or by selecting the stop button in your IDE.

View MailKit telemetry

The MailKit client library exposes telemetry that can be viewed in the .NET Aspire dashboard. To view the telemetry, navigate to the .NET Aspire dashboard at https://localhost:7251. Select the `newsletter` resource to view the telemetry on the **Metrics** page:



Open up the Swagger UI again, and make some requests to the `/subscribe` and `/unsubscribe` endpoints. Then, navigate back to the .NET Aspire dashboard and select the `newsletter` resource. Select a metric under the **mailkit.net.smtp** node, such as `mailkit.net.smtp.client.operation.count`. You should see the telemetry for the MailKit client:



Summary

In this article, you learned how to create a .NET Aspire integration that uses MailKit to send emails. You also learned how to integrate this integration into the Newsletter app you previously built. You learned about the core principles of .NET Aspire integrations,

such as exposing the underlying client library to consumers through dependency injection, and how to add health checks and telemetry to the integration. You also learned how to update the Newsletter service to use the MailKit client.

Go forth and build your own .NET Aspire integrations. If you believe that there's enough community value in the integration you're building, consider publishing it as a NuGet package for others to use. Furthermore, consider submitting a pull request to the .NET Aspire GitHub repository for consideration to be included in the official .NET Aspire integrations.

Next steps

Secure communication between hosting and client integrations

Secure communication between hosting and client integrations

Article • 09/24/2024

This article is a continuation of two previous articles demonstrating the creation of custom hosting integrations and custom client integrations.

One of the primary benefits to .NET Aspire is how it simplifies the configurability of resources and consuming clients (or integrations). This article demonstrates how to share authentication credentials from a custom resource in a hosting integration, to the consuming client in a custom client integration. The custom resource is a MailDev container that allows for either incoming or outgoing credentials. The custom client integration is a MailKit client that sends emails.

Prerequisites

Since this article continues from previous content, you should have already created the resulting solution as a starting point for this article. If you haven't already, complete the following articles:

- Create custom .NET Aspire hosting integrations
- Create custom .NET Aspire client integrations

The resulting solution from these previous articles contains the following projects:

- *MailDev.Hosting*: Contains the custom resource type for the MailDev container.
- *MailDevResource.AppHost*: The app host that uses the custom resource and defines it as a dependency for a Newsletter service.
- *MailDevResource.NewsletterService*: An ASP.NET Core Web API project that sends emails using the MailDev container.
- *MailDevResource.ServiceDefaults*: Contains the default service configurations intended for sharing.
- *MailKit.Client*: Contains the custom client integration that exposes the MailKit `SmtpClient` through a factory.

Update the MailDev resource

To flow authentication credentials from the MailDev resource to the MailKit integration, you need to update the MailDev resource to include the username and password parameters.

The MailDev container supports basic authentication for both incoming and outgoing simple mail transfer protocol (SMTP). To configure the credentials for incoming, you need to set the `MAILDEV_INCOMING_USER` and `MAILDEV_INCOMING_PASS` environment variables. For more information, see MailDev: Usage. Update the *MailDevResource.cs* file in the `MailDev.Hosting` project, by replacing its contents with the following C# code:

C#

```
// For ease of discovery, resource types should be placed in
// the Aspire.Hosting.ApplicationModel namespace. If there is
// likelihood of a conflict on the resource name consider using
// an alternative namespace.
namespace Aspire.Hosting.ApplicationModel;

public sealed class MailDevResource(
    string name,
    ParameterResource? username,
    ParameterResource password)
        : ContainerResource(name), IResourceWithConnectionString
{
    // Constants used to refer to well known-endpoint names, this is
specific
    // for each resource type. MailDev exposes an SMTP and HTTP endpoints.
    internal const string SmtpEndpointName = "smtp";
    internal const string HttpEndpointName = "http";

    private const string DefaultUsername = "mail-dev";

    // An EndpointReference is a core .NET Aspire type used for keeping
    // track of endpoint details in expressions. Simple literal values
cannot
    // be used because endpoints are not known until containers are
launched.
    private EndpointReference? _smtpReference;

    /// <summary>
    /// Gets the parameter that contains the MailDev SMTP server username.
    /// </summary>
    public ParameterResource? UsernameParameter { get; } = username;

    internal ReferenceExpression UserNameReference =>
        UsernameParameter is not null ?
        ReferenceExpression.Create($"{UsernameParameter}") :
        ReferenceExpression.Create($"{DefaultUsername}");

    /// <summary>
    /// Gets the parameter that contains the MailDev SMTP server password.
    /// </summary>
    public ParameterResource PasswordParameter { get; } = password;

    public EndpointReference SmtpEndpoint =>
        _smtpReference ??= new(this, SmtpEndpointName);
```

```
    // Required property on IResourceWithConnectionString. Represents a
connection
    // string that applications can use to access the MailDev server. In
this case
    // the connection string is composed of the SmtpEndpoint endpoint
reference.
    public ReferenceExpression ConnectionStringExpression =>
        ReferenceExpression.Create(

$"Endpoint=smtp://{SmtpEndpoint.Property(EndpointProperty.Host)}:
{SmtpEndpoint.Property(EndpointProperty.Port)};Username=
{UserNameReference};Password={PasswordParameter}"
        );
}
```

These updates add a `UsernameParameter` and `PasswordParameter` property. These properties are used to store the parameters for the MailDev username and password. The `ConnectionStringExpression` property is updated to include the username and password parameters in the connection string. Next, update the *MailDevResourceBuilderExtensions.cs* file in the `MailDev.Hosting` project with the following C# code:

C#

```
using Aspire.Hosting.ApplicationModel;

// Put extensions in the Aspire.Hosting namespace to ease discovery as
referencing
// the .NET Aspire hosting package automatically adds this namespace.
namespace Aspire.Hosting;

public static class MailDevResourceBuilderExtensions
{
    private const string UserEnvVarName = "MAILDEV_INCOMING_USER";
    private const string PasswordEnvVarName = "MAILDEV_INCOMING_PASS";

    /// <summary>
    /// Adds the <see cref="MailDevResource"/> to the given
    /// <paramref name="builder"/> instance. Uses the "2.1.0" tag.
    /// </summary>
    /// <param name="builder">The <see
cref="IDistributedApplicationBuilder"/>.</param>
    /// <param name="name">The name of the resource.</param>
    /// <param name="httpPort">The HTTP port.</param>
    /// <param name="smtpPort">The SMTP port.</param>
    /// <returns>
    /// An <see cref="IResourceBuilder{MailDevResource}"/> instance that
    /// represents the added MailDev resource.
    /// </returns>
    public static IResourceBuilder<MailDevResource> AddMailDev(
```

```
        this IDistributedApplicationBuilder builder,
        string name,
        int? httpPort = null,
        int? smtpPort = null,
        IResourceBuilder<ParameterResource>? userName = null,
        IResourceBuilder<ParameterResource>? password = null)
    {
        var passwordParameter = password?.Resource ??

ParameterResourceBuilderExtensions.CreateDefaultPasswordParameter(
                builder, $"{name}-password");

        // The AddResource method is a core API within .NET Aspire and is
        // used by resource developers to wrap a custom resource in an
        // IResourceBuilder<T> instance. Extension methods to customize
        // the resource (if any exist) target the builder interface.
        var resource = new MailDevResource(
            name, userName?.Resource, passwordParameter);

        return builder.AddResource(resource)
                      .WithImage(MailDevContainerImageTags.Image)
                      .WithImageRegistry(MailDevContainerImageTags.Registry)
                      .WithImageTag(MailDevContainerImageTags.Tag)
                      .WithHttpEndpoint(
                          targetPort: 1080,
                          port: httpPort,
                          name: MailDevResource.HttpEndpointName)
                      .WithEndpoint(
                          targetPort: 1025,
                          port: smtpPort,
                          name: MailDevResource.SmtpEndpointName)
                      .WithEnvironment(context =>
                      {
                          context.EnvironmentVariables[UserEnvVarName] =
resource.UserNameReference;
                          context.EnvironmentVariables[PasswordEnvVarName] =
resource.PasswordParameter;
                      });
    }
}

// This class just contains constant strings that can be updated
periodically
// when new versions of the underlying container are released.
internal static class MailDevContainerImageTags
{
    internal const string Registry = "docker.io";

    internal const string Image = "maildev/maildev";

    internal const string Tag = "2.1.0";
}
```

The preceding code updates the `AddMailDev` extension method to include the `userName` and `password` parameters. The `WithEnvironment` method is updated to include the `UserEnvVarName` and `PasswordEnvVarName` environment variables. These environment variables are used to set the MailDev username and password.

Update the app host

Now that the resource is updated to include the username and password parameters, you need to update the app host to include these parameters. Update the *Program.cs* file in the `MailDevResource.AppHost` project with the following C# code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var mailDevUsername = builder.AddParameter("maildev-username");
var mailDevPassword = builder.AddParameter("maildev-password");

var maildev = builder.AddMailDev(
    name: "maildev",
    userName: mailDevUsername,
    password: mailDevPassword);

builder.AddProject<Projects.MailDevResource_NewsletterService>
("newsletterservice")
       .WithReference(maildev);

builder.Build().Run();
```

The preceding code adds two parameters for the MailDev username and password. It assigns these parameters to the `MAILDEV_INCOMING_USER` and `MAILDEV_INCOMING_PASS` environment variables. The `AddMailDev` method has two chained calls to `WithEnvironment` which includes these environment variables. For more information on parameters, see External parameters.

Next, configure the secrets for these parameters. Right-click on the `MailDevResource.AppHost` project and select `Manage User Secrets`. Add the following JSON to the *secrets.json* file:

JSON

```
{
  "Parameters:maildev-username": "@admin",
  "Parameters:maildev-password": "t3st1ng"
}
```

> ⚠ **Warning**
>
> These credentials are for demonstration purposes only and MailDev is intended for local development. These credentials are fictitious and shouldn't be used in a production environment.

Update the MailKit integration

It's good practice for client integrations to expect connection strings to contain various key/value pairs, and to parse these pairs into the appropriate properties. Update the *MailKitClientSettings.cs* file in the `MailKit.Client` project with the following C# code:

C#

```
using System.Data.Common;
using System.Net;

namespace MailKit.Client;

/// <summary>
/// Provides the client configuration settings for connecting MailKit to an
SMTP server.
/// </summary>
public sealed class MailKitClientSettings
{
    internal const string DefaultConfigSectionName = "MailKit:Client";

    /// <summary>
    /// Gets or sets the SMTP server <see cref="Uri"/>.
    /// </summary>
    /// <value>
    /// The default value is <see langword="null"/>.
    /// </value>
    public Uri? Endpoint { get; set; }

    /// <summary>
    /// Gets or sets the network credentials that are optionally
configurable for SMTP
    /// server's that require authentication.
    /// </summary>
    /// <value>
    /// The default value is <see langword="null"/>.
    /// </value>
    public NetworkCredential? Credentials { get; set; }

    /// <summary>
    /// Gets or sets a boolean value that indicates whether the database
health check is disabled or not.
    /// </summary>
```

```
    /// <value>
    /// The default value is <see langword="false"/>.
    /// </value>
    public bool DisableHealthChecks { get; set; }

    /// <summary>
    /// Gets or sets a boolean value that indicates whether the
OpenTelemetry tracing is disabled or not.
    /// </summary>
    /// <value>
    /// The default value is <see langword="false"/>.
    /// </value>
    public bool DisableTracing { get; set; }

    /// <summary>
    /// Gets or sets a boolean value that indicates whether the
OpenTelemetry metrics are disabled or not.
    /// </summary>
    /// <value>
    /// The default value is <see langword="false"/>.
    /// </value>
    public bool DisableMetrics { get; set; }

    internal void ParseConnectionString(string? connectionString)
    {
        if (string.IsNullOrWhiteSpace(connectionString))
        {
            throw new InvalidOperationException($"""
                    ConnectionString is missing.
                    It should be provided in 'ConnectionStrings:
<connectionName>'
                    or '{DefaultConfigSectionName}:Endpoint' key.'
                    configuration section.
                    """);
        }

        if (Uri.TryCreate(connectionString, UriKind.Absolute, out var uri))
        {
            Endpoint = uri;
        }
        else
        {
            var builder = new DbConnectionStringBuilder
            {
                ConnectionString = connectionString
            };

            if (builder.TryGetValue("Endpoint", out var endpoint) is false)
            {
                throw new InvalidOperationException($"""
                        The 'ConnectionStrings:<connectionName>' (or
'Endpoint' key in
                        '{DefaultConfigSectionName}') is missing.
                        """);
            }
```

```
            if (Uri.TryCreate(endpoint.ToString(), UriKind.Absolute, out
uri) is false)
            {
                throw new InvalidOperationException($"""
                        The 'ConnectionStrings:<connectionName>' (or
'Endpoint' key in
                        '{DefaultConfigSectionName}') isn't a valid URI.
                        """);
            }

            Endpoint = uri;

            if (builder.TryGetValue("Username", out var username) &&
                builder.TryGetValue("Password", out var password))
            {
                Credentials = new(
                    username.ToString(), password.ToString());
            }
        }
    }
}
```

The preceding settings class, now includes a `Credentials` property of type `NetworkCredential`. The `ParseConnectionString` method is updated to parse the `Username` and `Password` keys from the connection string. If the `Username` and `Password` keys are present, a `NetworkCredential` is created and assigned to the `Credentials` property.

With the settings class updated to understand and populate the credentials, update the factory to conditionally use the credentials if they're configured. Update the *MailKitClientFactory.cs* file in the `MailKit.Client` project with the following C# code:

C#

```
using System.Net;
using MailKit.Net.Smtp;

namespace MailKit.Client;

/// <summary>
/// A factory for creating <see cref="ISmtpClient"/> instances
/// given a <paramref name="smtpUri"/> (and optional <paramref
name="credentials"/>).
/// </summary>
/// <param name="settings">
/// The <see cref="MailKitClientSettings"/> settings for the SMTP server
/// </param>
public sealed class MailKitClientFactory(MailKitClientSettings settings) :
IDisposable
```

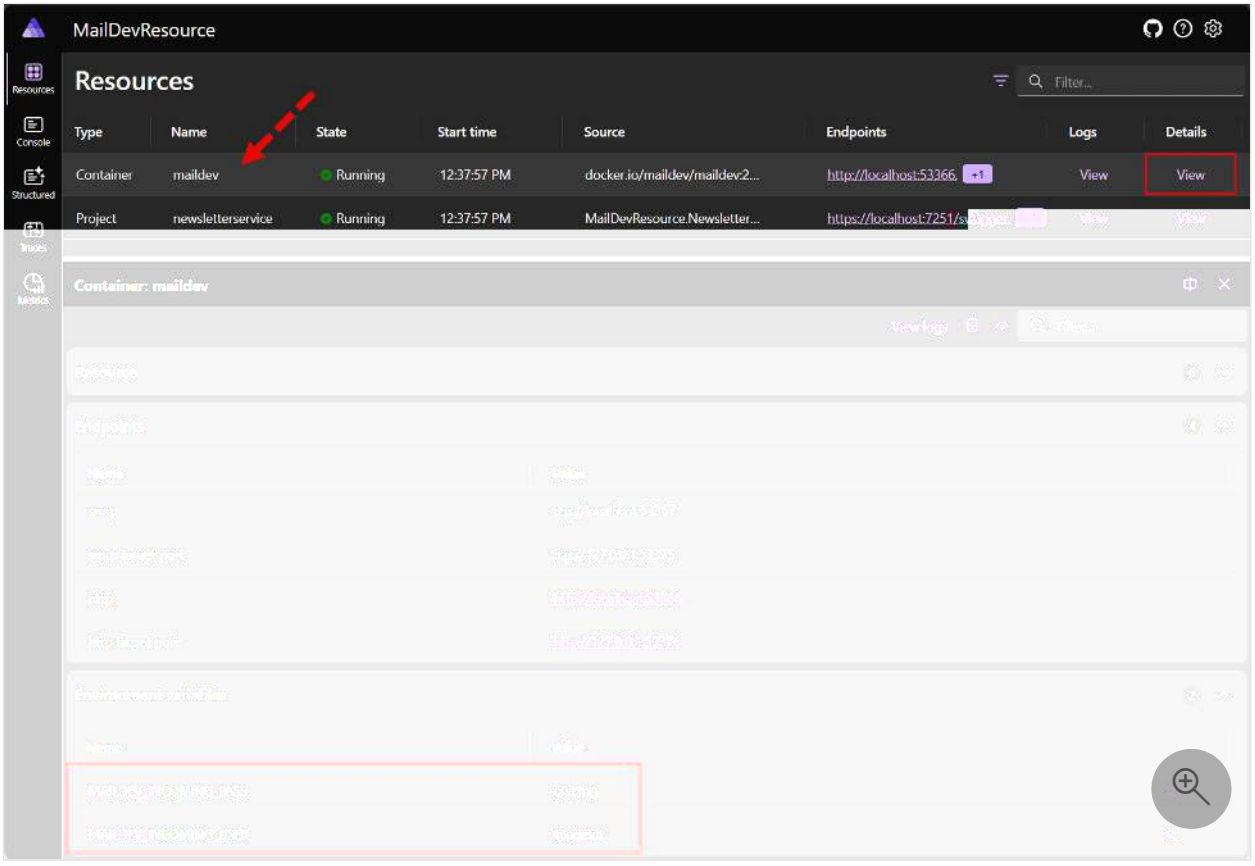
```
{
    private readonly SemaphoreSlim _semaphore = new(1, 1);

    private SmtpClient? _client;

    /// <summary>
    /// Gets an <see cref="ISmtpClient"/> instance in the connected state
    /// (and that's been authenticated if configured).
    /// </summary>
    /// <param name="cancellationToken">Used to abort client creation and
connection.</param>
    /// <returns>A connected (and authenticated) <see cref="ISmtpClient"/>
instance.</returns>
    /// <remarks>
    /// Since both the connection and authentication are considered
expensive operations,
    /// the <see cref="ISmtpClient"/> returned is intended to be used for
the duration of a request
    /// (registered as 'Scoped') and is automatically disposed of.
    /// </remarks>
    public async Task<ISmtpClient> GetSmtpClientAsync(
        CancellationToken cancellationToken = default)
    {
        await _semaphore.WaitAsync(cancellationToken);

        try
        {
            if (_client is null)
            {
                _client = new SmtpClient();

                await _client.ConnectAsync(settings.Endpoint,
cancellationToken)
                             .ConfigureAwait(false);

                if (settings.Credentials is not null)
                {
                    await _client.AuthenticateAsync(settings.Credentials,
cancellationToken)
                                 .ConfigureAwait(false);
                }
            }
        }
        finally
        {
            _semaphore.Release();
        }

        return _client;
    }

    public void Dispose()
    {
        _client?.Dispose();
        _semaphore.Dispose();
```

```
    }
}
```

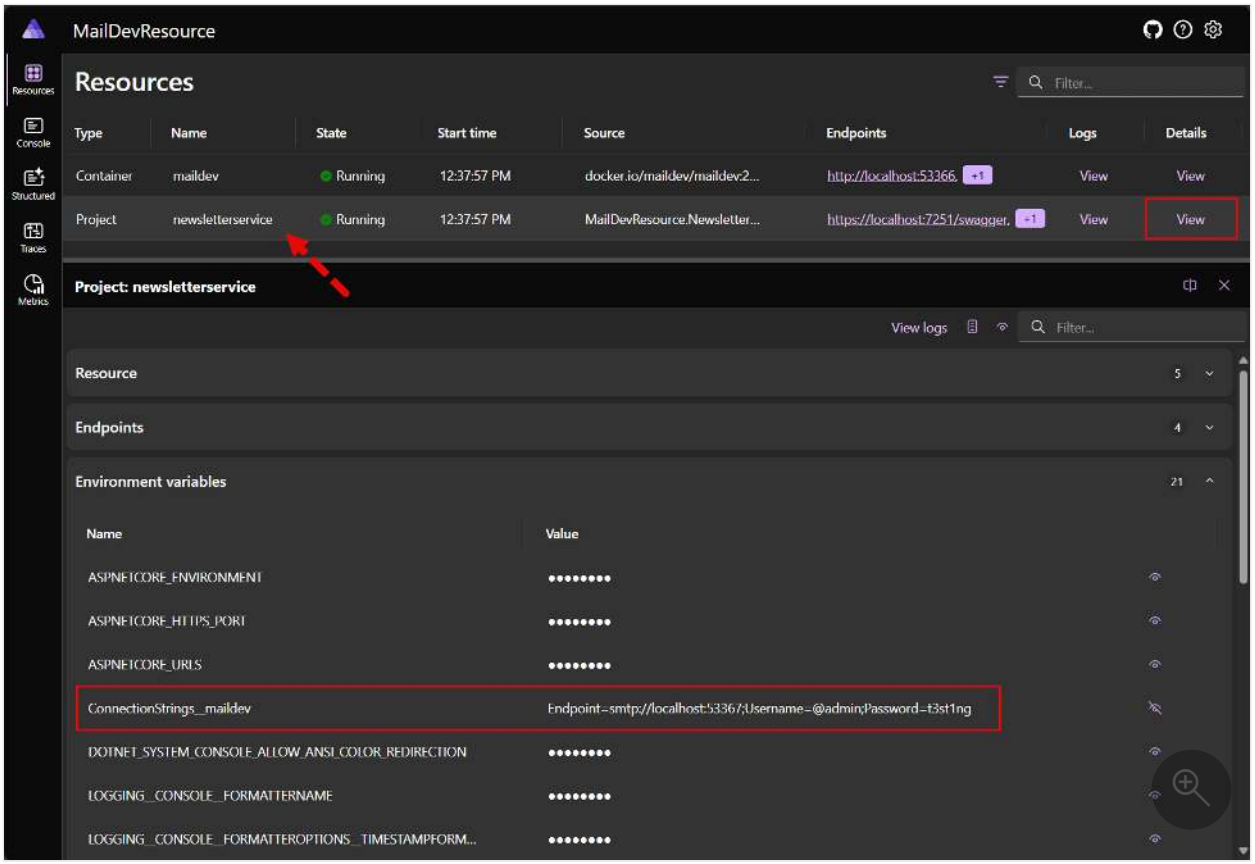
When the factory determines that credentials have been configured, it authenticates with the SMTP server after connecting before returning the `SmtpClient`.

Run the sample

Now that you've updated the resource, corresponding integration projects, and the app host, you're ready to run the sample app. To run the sample from your IDE, select `F5` or use `dotnet run` from the root directory of the solution to start the application—you should see the .NET Aspire dashboard. Navigate to the `maildev` container resource and view the details. You should see the username and password parameters in the resource details, under the **Environment Variables** section:



Likewise, you should see the connection string in the `newsletterservice` resource details, under the **Environment Variables** section:



Validate that everything is working as expected.

Summary

This article demonstrated how to flow authentication credentials from a custom resource to a custom client integration. The custom resource is a MailDev container that allows for either incoming or outgoing credentials. The custom client integration is a MailKit client that sends emails. By updating the resource to include the `username` and `password` parameters, and updating the integration to parse and use these parameters, authentication flows credentials from the hosting integration to the client integration.

.NET Aspire deployments

Article • 06/15/2024

.NET Aspire projects are built with cloud-agnostic principles, allowing deployment flexibility across various platforms supporting .NET and containers. Users can adapt the provided guidelines for deployment on other cloud environments or local hosting. The manual deployment process, while feasible, involves exhaustive steps prone to errors. Users prefer leveraging CI/CD pipelines and cloud-specific tooling for a more streamlined deployment experience tailored to their chosen infrastructure.

Deployment manifest

To enable deployment tools from Microsoft and other cloud providers to understand the structure of .NET Aspire projects, specialized targets of the AppHost project can be executed to generate a manifest file describing the projects/services used by the app and the properties necessary for deployment, such as environment variables.

For more information on the schema of the manifest and how to run app host project targets, see .NET Aspire manifest format for deployment tool builders.

Deploy to Azure

.NET Aspire enables deployment to Azure Container Apps. The number of environments .NET Aspire can deploy to will grow over time.

Azure Container Apps

.NET Aspire projects are designed to run in containerized environments. Azure Container Apps is a fully managed environment that enables you to run microservices and containerized applications on a serverless platform. The Azure Container Apps topic describes how to deploy Aspire apps to ACA manually, using bicep, or using the Azure Developer CLI (azd).

Use Application Insights for .NET Aspire telemetry

.NET Aspire projects are designed to emit telemetry using OpenTelemetry which uses a provider model. .NET Aspire projects can direct their telemetry to Azure Monitor / Application Insights using the Azure Monitor telemetry distro. For more information, see Use Application Insights for .NET Aspire telemetry for step-by-step instructions.

Deploy to Kubernetes

Kubernetes is a popular container orchestration platform that can run .NET Aspire projects. To deploy .NET Aspire projects to Kubernetes clusters, you need to map the .NET Aspire JSON manifest to a Kubernetes YAML manifest file. There are two ways to do this: by using the Aspir8 project, or by manually creating Kubernetes manifests.

The Aspir8 project

Aspir8, an open-source project, handles the generation of deployment YAML based on the .NET Aspire app host manifest. The project outputs a .NET global tool that can be used to perform a series of tasks, resulting in the generation of Kubernetes manifests:

- `aspirate init`: Initializes the **Aspir8** project in the current directory.
- `aspirate generate`: Generates Kubernetes manifests based on the .NET Aspire app host manifest.
- `aspirate apply`: Applies the generated Kubernetes manifests to the Kubernetes cluster.
- `aspirate destroy`: Deletes the resources created by the `apply` command.

With these commands, you can build your apps, containerize them, and deploy them to Kubernetes clusters. For more information, see Aspir8.

Manually create Kubernetes manifests

Alternatively, the Kubernetes manifests can be created manually. This involves more effort and is more time consuming. For more information, see Deploy a .NET microservice to Kubernetes.

Deploy a .NET Aspire project to Azure Container Apps

Article • 06/21/2024

.NET Aspire projects are designed to run in containerized environments. Azure Container Apps is a fully managed environment that enables you to run microservices and containerized applications on a serverless platform. This article will walk you through creating a new .NET Aspire solution and deploying it to Microsoft Azure Container Apps using the Azure Developer CLI (`azd`). You'll learn how to complete the following tasks:

- ✓ Provision an Azure resource group and Container Registry
- ✓ Publish the .NET Aspire projects as container images in Azure Container Registry
- ✓ Provision a Redis container in Azure
- ✓ Deploy the apps to an Azure Container Apps environment
- ✓ View application console logs to troubleshoot application issues

Prerequisites

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0
- An OCI compliant container runtime, such as:
  - Docker Desktop or Podman. For more information, see Container runtime.
- An Integrated Developer Environment (IDE) or code editor, such as:
  - Visual Studio 2022 version 17.9 or higher (Optional)
  - Visual Studio Code (Optional)
    - C# Dev Kit: Extension (Optional)
  - JetBrains Rider with .NET Aspire plugin (Optional)

For more information, see .NET Aspire setup and tooling, and .NET Aspire SDK.

As an alternative to this tutorial and for a more in-depth guide, see Deploy a .NET Aspire project to Azure Container Apps using azd (in-depth guide).

Deploy .NET Aspire projects with `azd`

With .NET Aspire and Azure Container Apps (ACA), you have a great hosting scenario for building out your cloud-native apps with .NET. We built some great new features into the Azure Developer CLI (`azd`) specific for making .NET Aspire development and

deployment to Azure a friction-free experience. You can still use the Azure CLI and/or Bicep options when you need a granular level of control over your deployments. But for new projects, you won't find an easier path to success for getting a new microservice topology deployed into the cloud.

Create a .NET Aspire project

As a starting point, this article assumes that you've created a .NET Aspire project from the **.NET Aspire Starter Application** template. For more information, see Quickstart: Build your first .NET Aspire project.

Resource naming

When you create new Azure resources, it's important to follow the naming requirements. For Azure Container Apps, the name must be 2-32 characters long and consist of lowercase letters, numbers, and hyphens. The name must start with a letter and end with an alphanumeric character.

For more information, see Naming rules and restrictions for Azure resources.

Install the Azure Developer CLI

The process for installing `azd` varies based on your operating system, but it is widely available via `winget`, `brew`, `apt`, or directly via `curl`. To install `azd`, see Install Azure Developer CLI.

Initialize the template

1. Open a new terminal window and `cd` into the directory of your .NET Aspire solution.

2. Execute the `azd init` command to initialize your project with `azd`, which will inspect the local directory structure and determine the type of app.

   Azure Developer CLI

   ```
   azd init
   ```

   For more information on the `azd init` command, see azd init.

3. Select **Use code in the current directory** when `azd` prompts you with two app initialization options.

```
Output
```

4. After scanning the directory, `azd` prompts you to confirm that it found the correct .NET Aspire *AppHost* project. Select the **Confirm and continue initializing my app** option.

```
Output
```

5. Enter an environment name, which is used to name provisioned resources in Azure and managing different environments such as `dev` and

```
```

- *azure.yaml*: Describes the services of the app, such as .NET Aspire AppHost project, and maps them to Azure resources.
- *.azure/config.json*: Configuration file that informs `azd` what the current active environment is.
- *.azure/aspireazddev/.env*: Contains environment specific overrides.

Deploy the template

1. Once an `azd` template is initialized, the provisioning and deployment process can be executed as a single command from the *AppHost* project directory using azd up:

   Azure Developer CLI

   ```
   azd up
   ```

2. Select the subscription you'd like to deploy to from the list of available options:

   Output

   ```
   Select an Azure Subscription to use:  [Use arrows to move, type to
   filter]
     1. SampleSubscription01 (xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx)
     2. SamepleSubscription02 (xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx)
   ```

3. Select the desired Azure location to use from the list of available options:

   Output

   ```
   Select an Azure location to use:  [Use arrows to move, type to filter]
     42. (US) Central US (centralus)
     43. (US) East US (eastus)
   > 44. (US) East US 2 (eastus2)
     46. (US) North Central US (northcentralus)
     47. (US) South Central US (southcentralus)
   ```

After you make your selections, `azd` executes the provisioning and deployment process.

Output

```
By default, a service can only be reached from inside the Azure Container
Apps environment it is running in. Selecting a service here will also allow
it to be reached from the Internet.
? Select which services to expose to the Internet webfrontend
? Select an Azure Subscription to use:  1. <YOUR SUBSCRIPTION>
```

```
? Select an Azure location to use: 1. <YOUR LOCATION>

Packaging services (azd package)


Provisioning Azure resources (azd provision)
Provisioning Azure resources can take some time.

Subscription: <YOUR SUBSCRIPTION>
Location: <YOUR LOCATION>

  You can view detailed progress in the Azure Portal:
  <LINK TO DEPLOYMENT>

  (✓) Done: Resource group: <YOUR RESOURCE GROUP>
  (✓) Done: Container Registry: <ID>
  (✓) Done: Log Analytics workspace: <ID>
  (✓) Done: Container Apps Environment: <ID>

SUCCESS: Your application was provisioned in Azure in 1 minute 13 seconds.
You can view the resources created under the resource group <YOUR RESOURCE
GROUP> in Azure Portal:
<LINK TO RESOURCE GROUP OVERVIEW>

Deploying services (azd deploy)

  (✓) Done: Deploying service apiservice
  - Endpoint: <YOUR UNIQUE apiservice APP>.azurecontainerapps.io/

  (✓) Done: Deploying service webfrontend
  - Endpoint: <YOUR UNIQUE webfrontend APP>.azurecontainerapps.io/

Aspire Dashboard: <LINK TO DEPLOYED .NET ASPIRE DASHBOARD>

SUCCESS: Your up workflow to provision and deploy to Azure completed in 3
minutes 50 seconds.
```

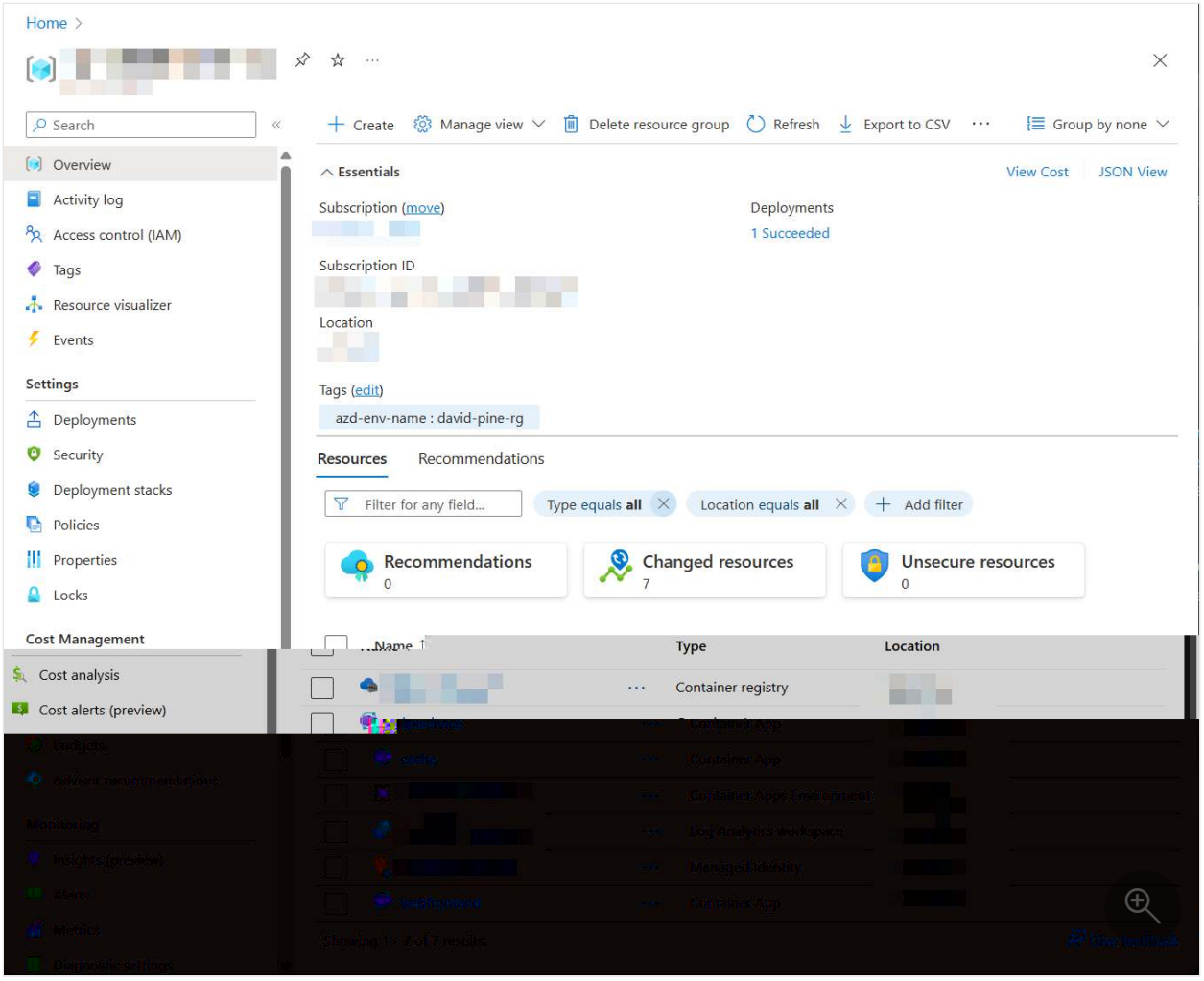
The `azd up` command acts as wrapper for the following individual `azd` commands to provision and deploy your resources in a single step:

1. azd package: The app projects and their dependencies are packaged into containers.
2. azd provision: The Azure resources the app will need are provisioned.
3. azd deploy: The projects are pushed as containers into an Azure Container Registry instance, and then used to create new revisions of Azure Container Apps in which the code will be hosted.

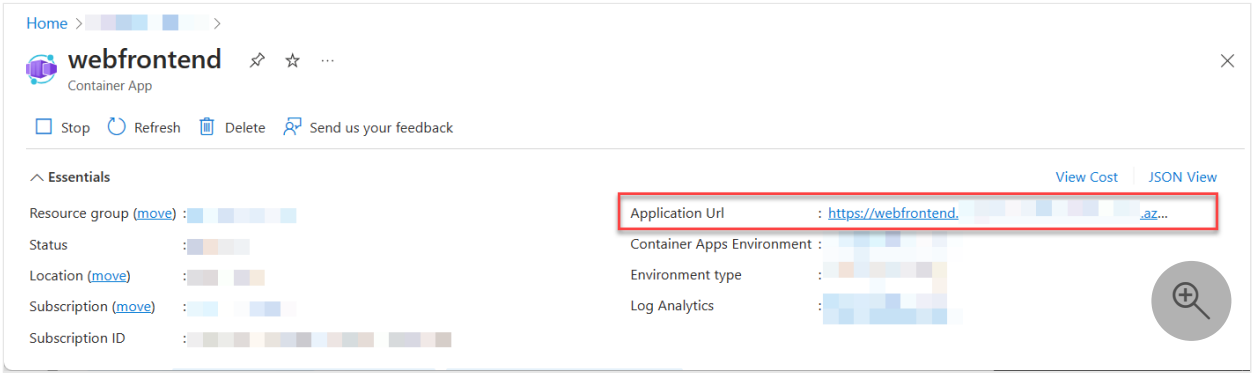
When the `azd up` stages complete, your app will be available on Azure, and you can open the Azure portal to explore the resources. `azd` also outputs URLs to access the deployed apps directly.

Test the deployed app

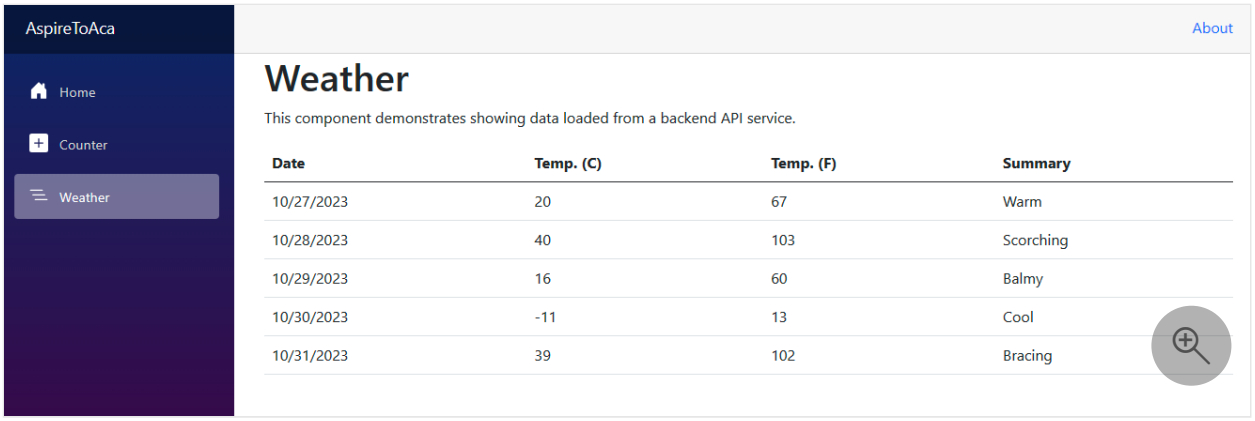
Now that the app has been provisioned and deployed, you can browse to the Azure portal. In the resource group where you deployed the app, you'll see the three container apps and other resources.



Click on the `web` Container App to open it up in the portal.



Click the **Application URL** link to open the front end in the browser.



When you click the "Weather" node in the navigation bar, the front end `web` container app makes a call to the `apiservice` container app to get data. The front end's output will be cached using the `redis` container app and the .NET Aspire Redis Output Caching integration. As you refresh the front end a few times, you'll notice that the weather data is cached. It will update after a few seconds.

Deploy the .NET Aspire Dashboard

You can deploy the .NET Aspire dashboard as part of your hosted app. This feature is now fully supported. When deploying, the azd output logs print an additional URL to the deployed dashboard.

You can run `azd monitor` to automatically launch the dashboard.

Azure Developer CLI

```
azd monitor
```

Clean up resources

Run the following Azure CLI command to delete the resource group when you no longer need the Azure resources you created. Deleting the resource group also deletes the resources contained inside of it.

Azure CLI

```
az group delete --name <your-resource-group-name>
```

For more information, see Clean up resources in Azure.

Deploy a .NET Aspire project to Azure Container Apps using Visual Studio

Article • 06/21/2024

.NET Aspire projects are designed to run in containerized environments. Azure Container Apps is a fully managed environment that enables you to run microservices and containerized applications on a serverless platform. This article will walk you through creating a new .NET Aspire solution and deploying it to Microsoft Azure Container Apps using the Visual Studio. You'll learn how to complete the following tasks:

- ✓ Provision an Azure resource group and Container Registry
- ✓ Publish the .NET Aspire projects as container images in Azure Container Registry
- ✓ Provision a Redis container in Azure
- ✓ Deploy the apps to an Azure Container Apps environment
- ✓ View application console logs to troubleshoot application issues

Prerequisites

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0
- An OCI compliant container runtime, such as:
  - Docker Desktop or Podman. For more information, see Container runtime.
- An Integrated Developer Environment (IDE) or code editor, such as:
  - Visual Studio 2022 version 17.9 or higher (Optional)
  - Visual Studio Code (Optional)
    - C# Dev Kit: Extension (Optional)
  - JetBrains Rider with .NET Aspire plugin (Optional)

For more information, see .NET Aspire setup and tooling, and .NET Aspire SDK.

Create a .NET Aspire project

As a starting point, this article assumes that you've created a .NET Aspire project from the **.NET Aspire Starter Application** template. For more information, see Quickstart: Build your first .NET Aspire project.

Resource naming

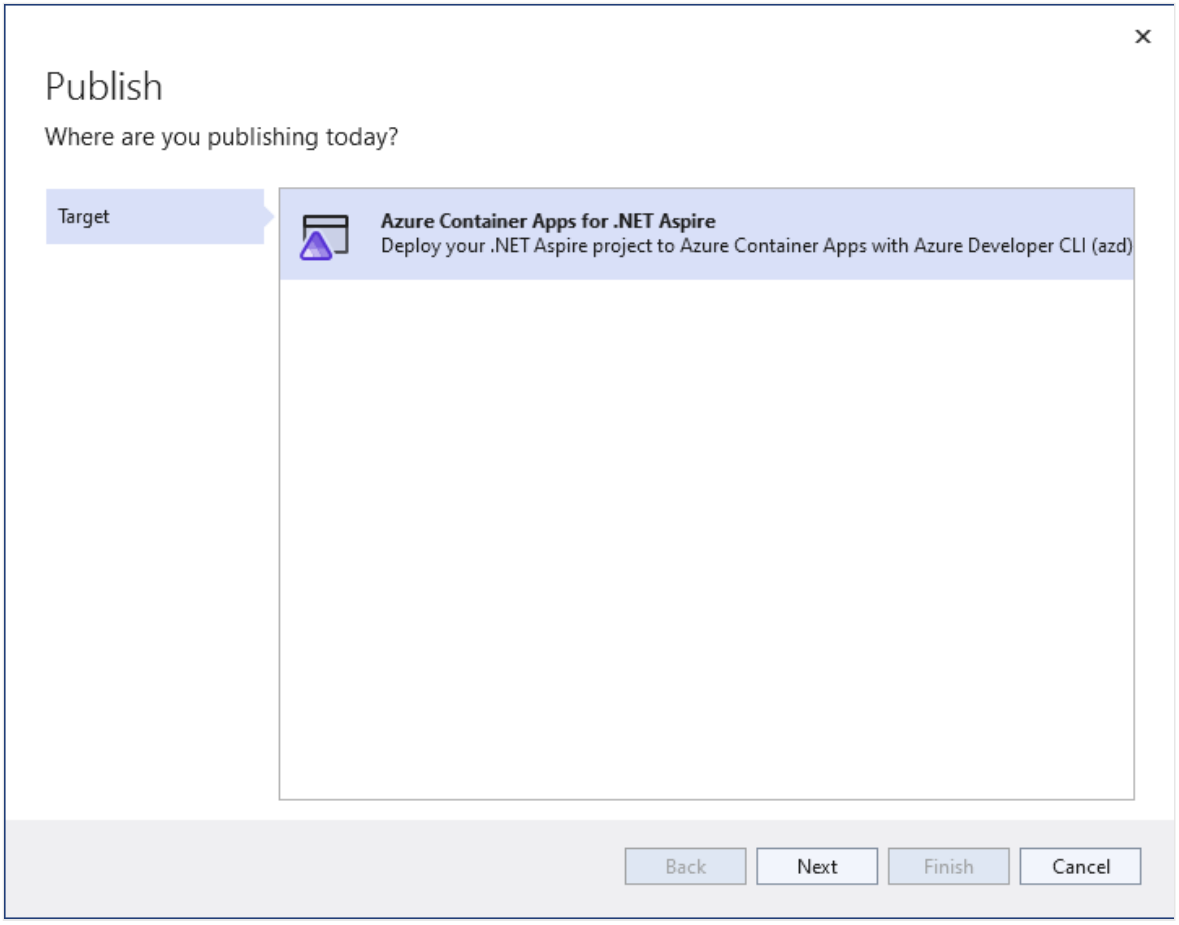
When you create new Azure resources, it's important to follow the naming requirements. For Azure Container Apps, the name must be 2-32 characters long and consist of lowercase letters, numbers, and hyphens. The name must start with a letter and end with an alphanumeric character.

For more information, see Naming rules and restrictions for Azure resources.

Deploy the app

1. In the solution explorer, right-click on the **.AppHost** project and select **Publish** to open the **Publish** dialog.

2. Select **Azure Container Apps for .NET Aspire** as the publishing target.



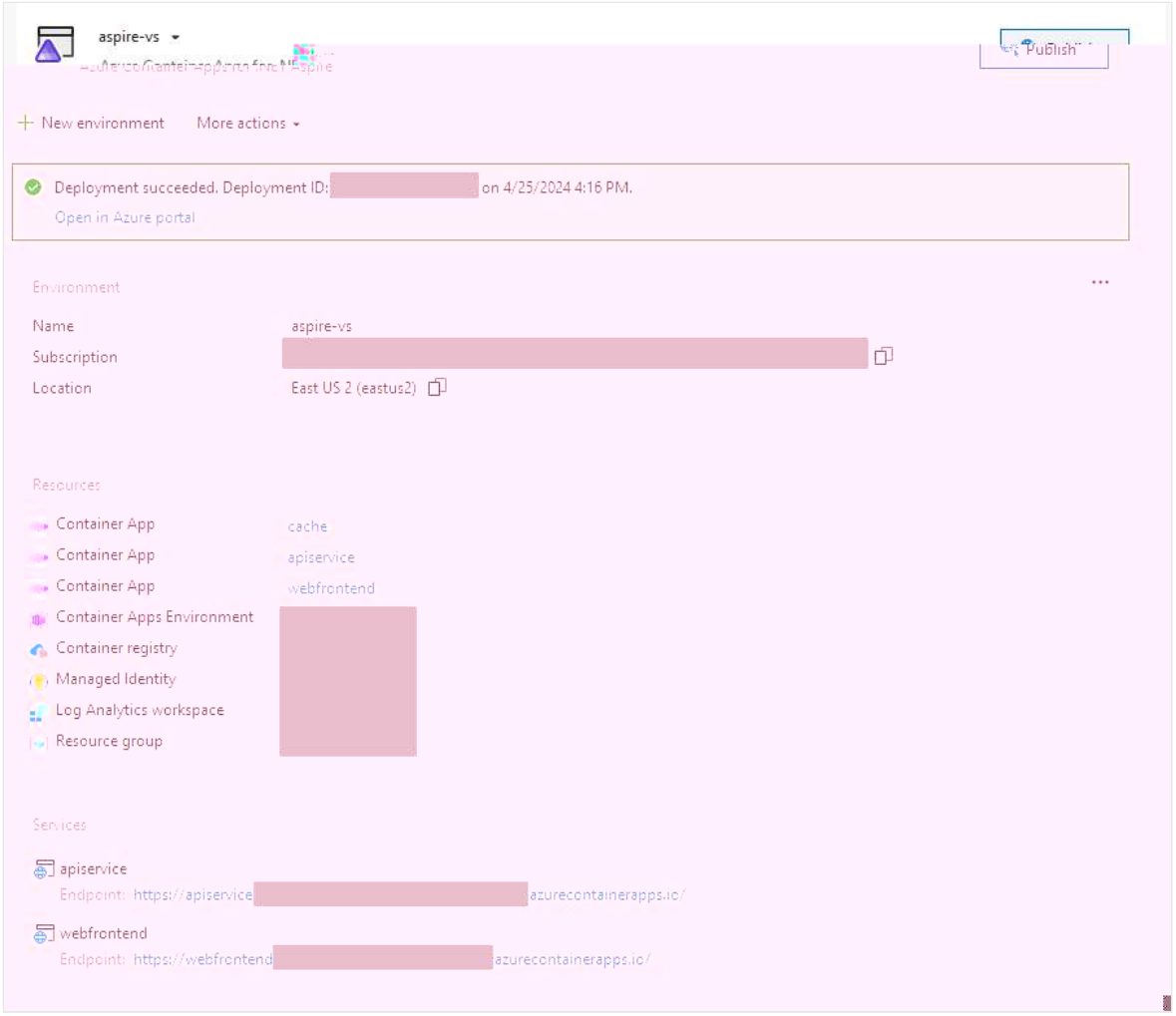
3. On the **AzDev Environment** step, select your desired **Subscription** and **Location** values and then enter an **Environment name** such as *aspire-vs*. The environment name determines the naming of Azure Container Apps environment resources.

4. Select **Finish** to close the dialog workflow and view the deployment environment summary.

5. Select **Publish** to provision and deploy the resources on Azure. This process may take several minutes to complete. Visual Studio provides status updates on the

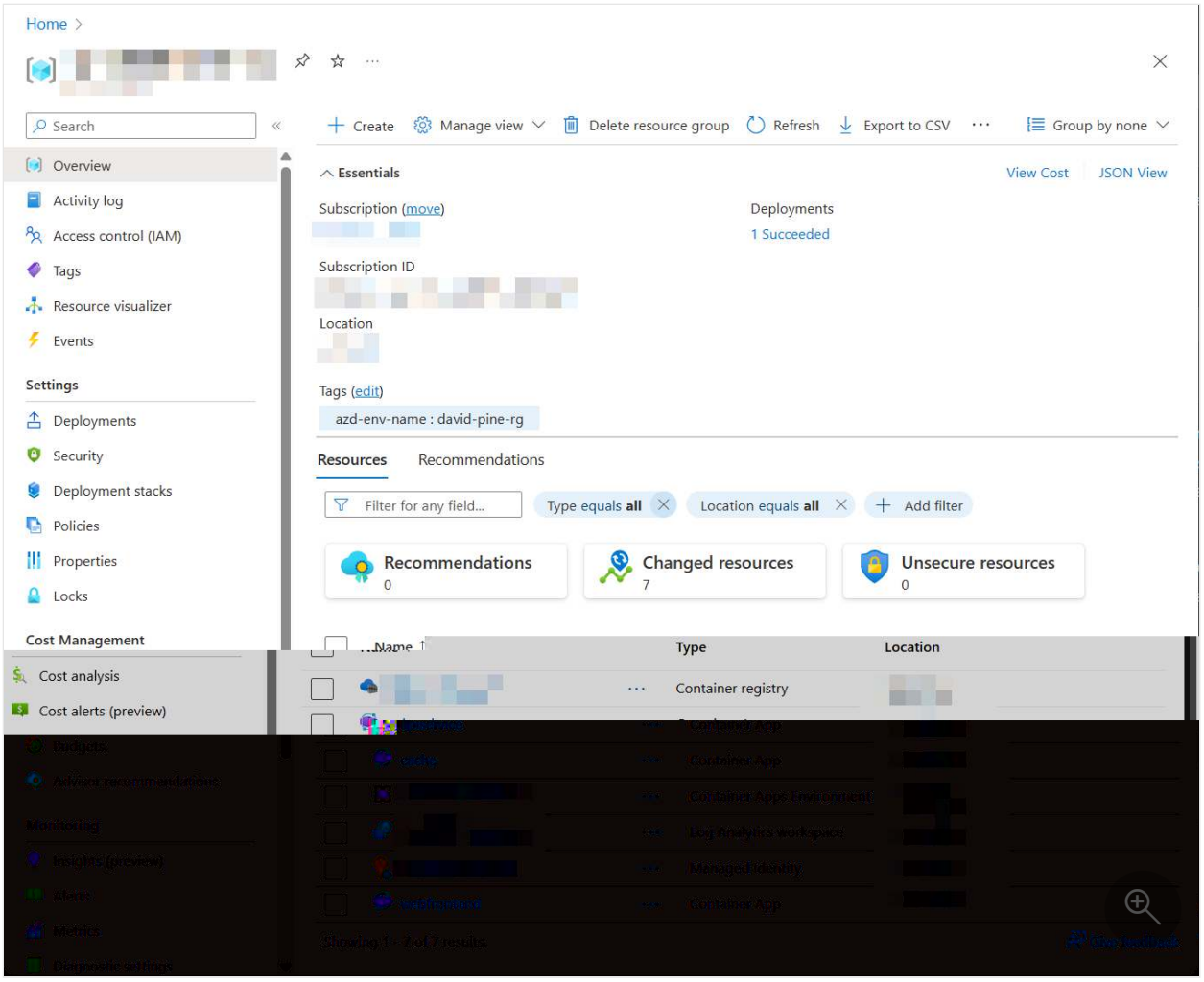
deployment progress.

6. When the publish completes, Visual Studio displays the resource URLs at the bottom of the environment screen. Use these links to view the various deployed resources. Select the **webfrontend** URL to open a browser to the deployed app.

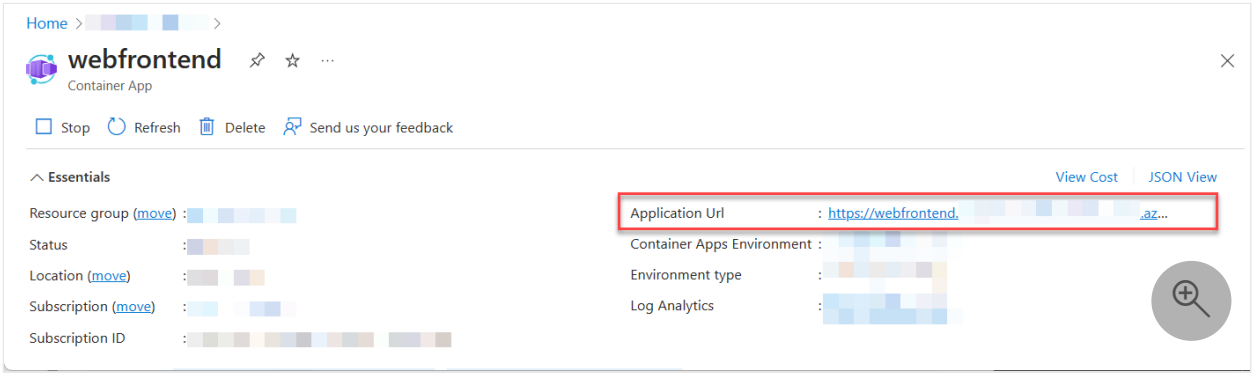


Test the deployed app

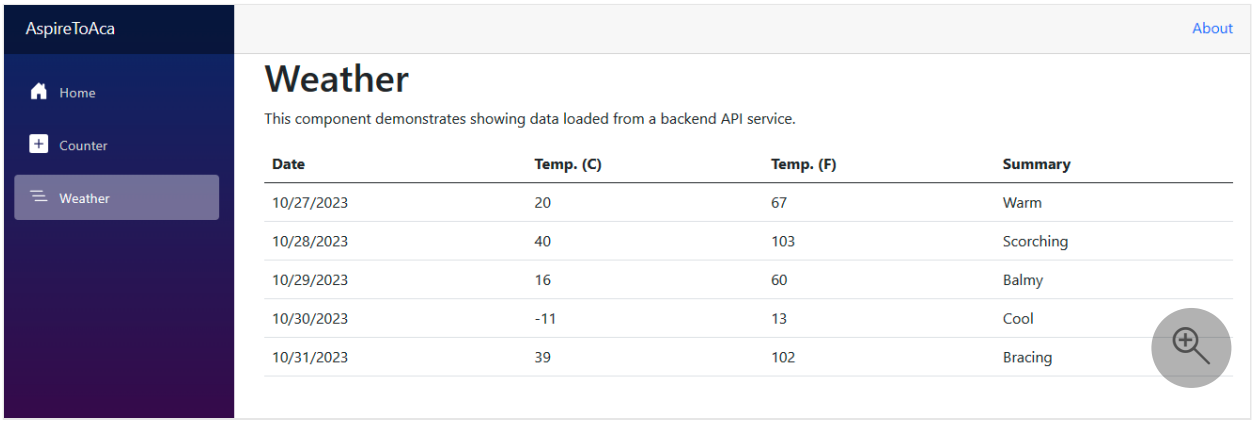
Now that the app has been provisioned and deployed, you can browse to the Azure portal. In the resource group where you deployed the app, you'll see the three container apps and other resources.



Click on the `web` Container App to open it up in the portal.



Click the **Application URL** link to open the front end in the browser.



When you click the "Weather" node in the navigation bar, the front end `web` container app makes a call to the `apiservice` container app to get data. The front end's output will be cached using the `redis` container app and the .NET Aspire Redis Output Caching integration. As you refresh the front end a few times, you'll notice that the weather data is cached. It will update after a few seconds.

Deploy the .NET Aspire Dashboard

You can deploy the .NET Aspire dashboard as part of your hosted app. This feature is now fully supported. When deploying, the azd output logs print an additional URL to the deployed dashboard.

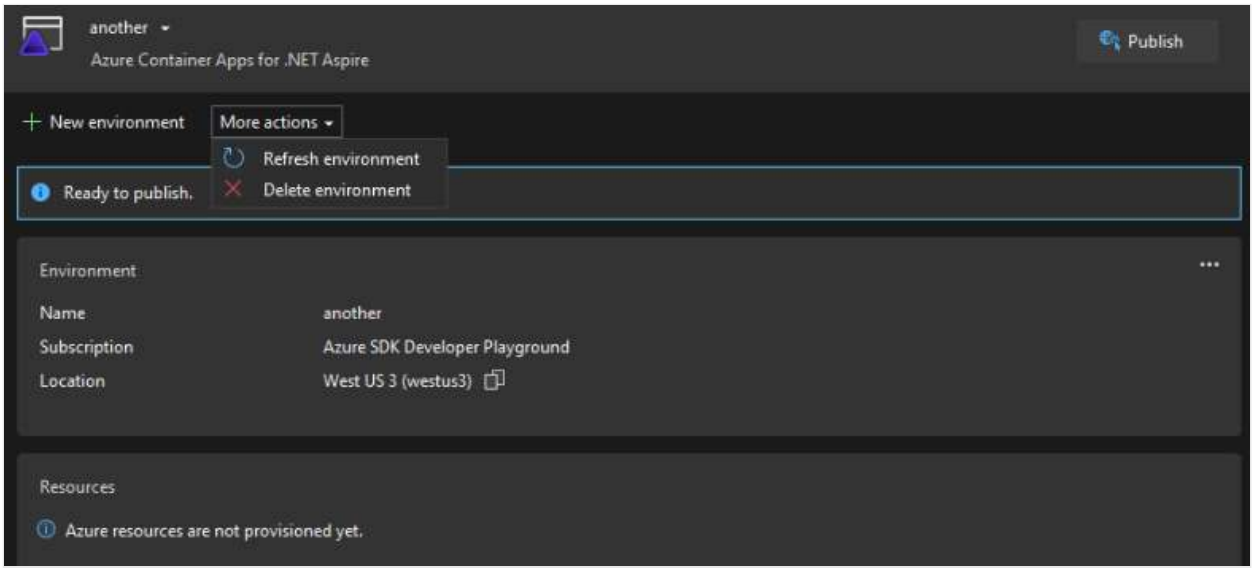
You can run `azd monitor` to automatically launch the dashboard.

Azure Developer CLI

```
azd monitor
```

Clean up resources

To delete the `azd` environment, the **More actions** dropdown and then choose **Delete environment**.



Deploy a .NET Aspire project to Azure Container Apps using the Azure Developer CLI (in-depth guide)

Article • 06/15/2024

The Azure Developer CLI (`azd`) has been extended to support deploying .NET Aspire projects. Use this guide to walk through the process of creating and deploying a .NET Aspire project to Azure Container Apps using the Azure Developer CLI. In this tutorial, you'll learn the following concepts:

- ✓ Explore how `azd` integration works with .NET Aspire projects
- ✓ Provision and deploy resources on Azure for a .NET Aspire project using `azd`
- ✓ Generate Bicep infrastructure and other template files using `azd`

Prerequisites

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0
- An OCI compliant container runtime, such as:
  - Docker Desktop or Podman. For more information, see Container runtime.
- An Integrated Developer Environment (IDE) or code editor, such as:
  - Visual Studio 2022 version 17.9 or higher (Optional)
  - Visual Studio Code (Optional)
    - C# Dev Kit: Extension (Optional)
  - JetBrains Rider with .NET Aspire plugin (Optional)

For more information, see .NET Aspire setup and tooling, and .NET Aspire SDK.

You will also need to have the Azure Developer CLI installed locally. Common install options include the following:

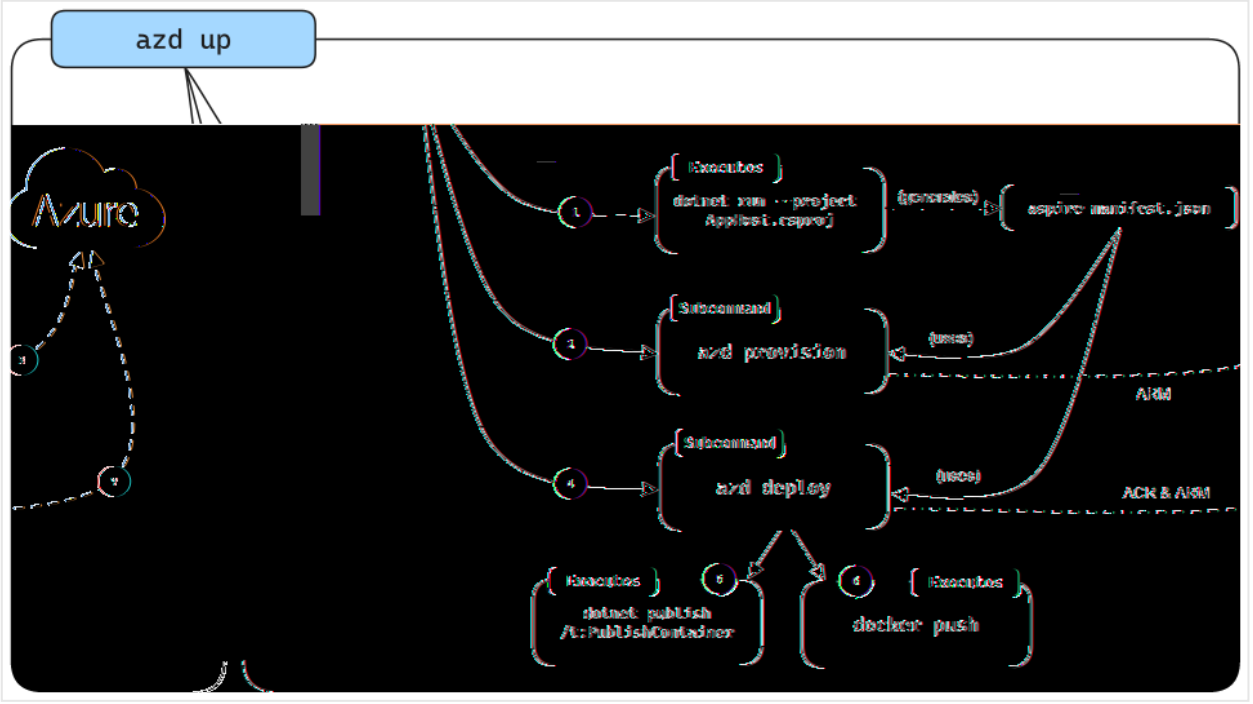
Windows

PowerShell

```
winget install microsoft.azd
```

How Azure Developer CLI integration works

The `azd init` workflow provides customized support for .NET Aspire projects. The following diagram illustrates how this flow works conceptually and how `azd` and .NET Aspire are integrated:



1. When `azd` targets a .NET Aspire project it starts the AppHost with a special command (`dotnet run --project AppHost.csproj --output-path manifest.json --publisher manifest`), which produces the Aspire manifest file.
2. The manifest file is interrogated by the `azd provision` sub-command logic to generate Bicep files in-memory only (by default).
3. After generating the Bicep files, a deployment is triggered using Azure's ARM APIs targeting the subscription and resource group provided earlier.
4. Once the underlying Azure resources are configured, the `azd deploy` sub-command logic is executed which uses the same Aspire manifest file.
5. As part of deployment `azd` makes a call to `dotnet publish` using .NET's built in container publishing support to generate container images.
6. Once `azd` has built the container images it pushes them to the ACR registry that was created during the provisioning phase.
7. Finally, once the container image is in ACR, `azd` updates the resource using ARM to start using the new version of the container image.

> **Note**

`azd` also enables you to output the generated Bicep to an `infra` folder in your project, which you can read more about in the Generating Bicep from .NET Aspire app model section.

Provision and deploy a .NET Aspire starter app

The steps in this section demonstrate how to create a .NET Aspire start app and handle provisioning and deploying the app resources to Azure using `azd`.

Create the .NET Aspire starter app

Create a new .NET Aspire project using the `dotnet new` command. You can also create the project using Visual Studio.

.NET CLI

```
dotnet new aspire-starter --use-redis-cache -o AspireSample
cd AspireSample
dotnet run --project AspireSample.AppHost\AspireSample.AppHost.csproj
```

The previous commands create a new .NET Aspire project based on the `aspire-starter` template which includes a dependency on Redis cache. It runs the .NET Aspire project which verifies that everything is working correctly.

Initialize the template

1. Open a new terminal window and `cd` into the directory of your .NET Aspire solution.

2. Execute the `azd init` command to initialize your project with `azd`, which will inspect the local directory structure and determine the type of app.

   Azure Developer CLI

   ```
   azd init
   ```

   For more information on the `azd init` command, see azd init.

3. Select **Use code in the current directory** when `azd` prompts you with two app initialization options.

Output

```
? How do you want to initialize your app?  [Use arrows to move, type to
filter]
> Use code in the current directory
  Select a template
```

4. After scanning the directory, `azd` prompts you to confirm that it found the correct .NET Aspire *AppHost* project. Select the **Confirm and continue initializing my app** option.

Output

```
Detected services:

  .NET (Aspire)
  Detected in:
D:\source\repos\AspireSample\AspireSample.AppHost\AspireSample.AppHost.
csproj

azd will generate the files necessary to host your app on Azure using
Azure Container Apps.

? Select an option  [Use arrows to move, type to filter]
> Confirm and continue initializing my app
  Cancel and exit
```

5. Enter an environment name, which is used to name provisioned resources in Azure and managing different environments such as `dev` and `prod`.

Output

```
Generating files to run your app on Azure:

  (✓) Done: Generating ./azure.yaml
  (✓) Done: Generating ./next-steps.md

SUCCESS: Your app is ready for the cloud!
You can provision and deploy your app to Azure by running the azd up
command in this directory. For more information on configuring your
app, see ./next-steps.md
```

`azd` generates a number of files and places them into the working directory. These files are:

- *azure.yaml*: Describes the services of the app, such as .NET Aspire AppHost project, and maps them to Azure resources.

- *.azure/config.json*: Configuration file that informs `azd` what the current active environment is.
- *.azure/aspireazddev/.env*: Contains environment specific overrides.

The *azure.yaml* file has the following contents:

yml

```yml
# yaml-language-server:
$schema=https://raw.githubusercontent.com/Azure/azure-
dev/main/schemas/v1.0/azure.yaml.json

name: AspireSample
services:
  app:
    language: dotnet
    project: .\AspireSample.AppHost\AspireSample.AppHost.csproj
    host: containerapp
```

Resource naming

When you create new Azure resources, it's important to follow the naming requirements. For Azure Container Apps, the name must be 2-32 characters long and consist of lowercase letters, numbers, and hyphens. The name must start with a letter and end with an alphanumeric character.

For more information, see Naming rules and restrictions for Azure resources.

Initial deployment

1. In order to deploy the .NET Aspire project, authenticate to Azure AD to call the Azure resource management APIs.

   Azure Developer CLI

   ```
   azd auth login
   ```

   The previous command will launch a browser to authenticate the command-line session.

2. Once authenticated, run the following command from the *AppHost* project directory to provision and deploy the application.

   Azure Developer CLI

```
azd up
```

> **Important**
>
> To push container images to the Azure Container Registry (ACR), you need to have `Microsoft.Authorization/roleAssignments/write` access. This can be achieved by enabling an **Admin user** on the registry. Open the Azure Portal, navigate to the ACR resource / Settings / Access keys, and then select the **Admin user** checkbox. For more information, see Enable admin user.

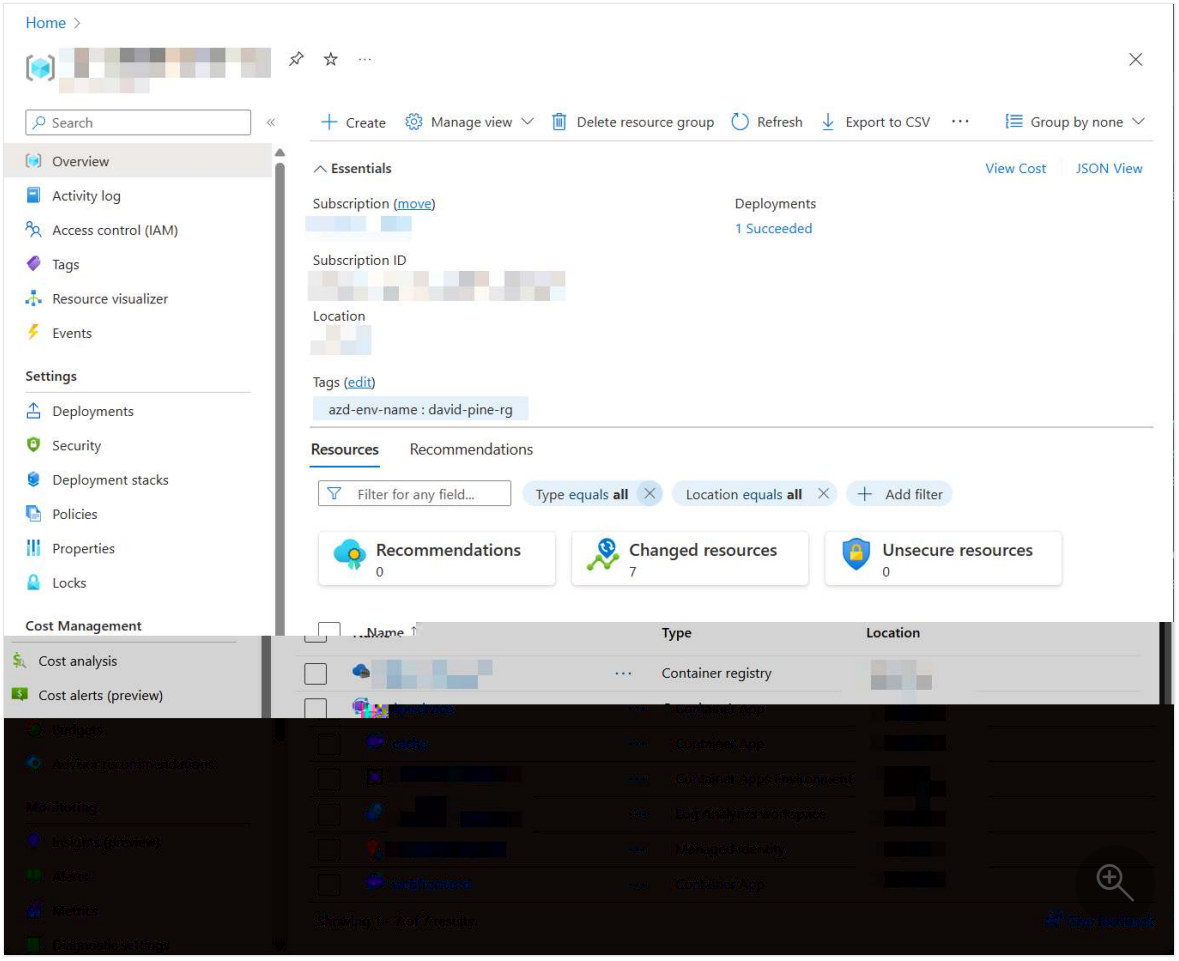
3. When prompted, select the subscription and location the resources should be deployed to. Once these options are selected the .NET Aspire project will be deployed.

Output

```
By default, a service can only be reached from inside the Azure
Container Apps environment it is running in. Selecting a service here
will also allow it to be reached from the Internet.
? Select which services to expose to the Internet webfrontend
? Select an Azure Subscription to use:  1. <YOUR SUBSCRIPTION>
? Select an Azure location to use: 1. <YOUR LOCATION>

Packaging services (azd package)


Provisioning Azure resources (azd provision)
Provisioning Azure resources can take some time.

Subscription: <YOUR SUBSCRIPTION>
Location: <YOUR LOCATION>

  You can view detailed progress in the Azure Portal:
  <LINK TO DEPLOYMENT>

  (✓) Done: Resource group: <YOUR RESOURCE GROUP>
  (✓) Done: Container Registry: <ID>
  (✓) Done: Log Analytics workspace: <ID>
  (✓) Done: Container Apps Environment: <ID>

SUCCESS: Your application was provisioned in Azure in 1 minute 13
seconds.
You can view the resources created under the resource group <YOUR
RESOURCE GROUP> in Azure Portal:
<LINK TO RESOURCE GROUP OVERVIEW>

Deploying services (azd deploy)
```

```
   (✓) Done: Deploying service apiservice
   - Endpoint: <YOUR UNIQUE apiservice APP>.azurecontainerapps.io/

   (✓) Done: Deploying service webfrontend
   - Endpoint: <YOUR UNIQUE webfrontend APP>.azurecontainerapps.io/

Aspire Dashboard: <LINK TO DEPLOYED .NET ASPIRE DASHBOARD>

SUCCESS: Your up workflow to provision and deploy to Azure completed in
3 minutes 50 seconds.
```

The final line of output from the `azd` command is a link to the Azure Portal that shows all of the Azure resources that were deployed:

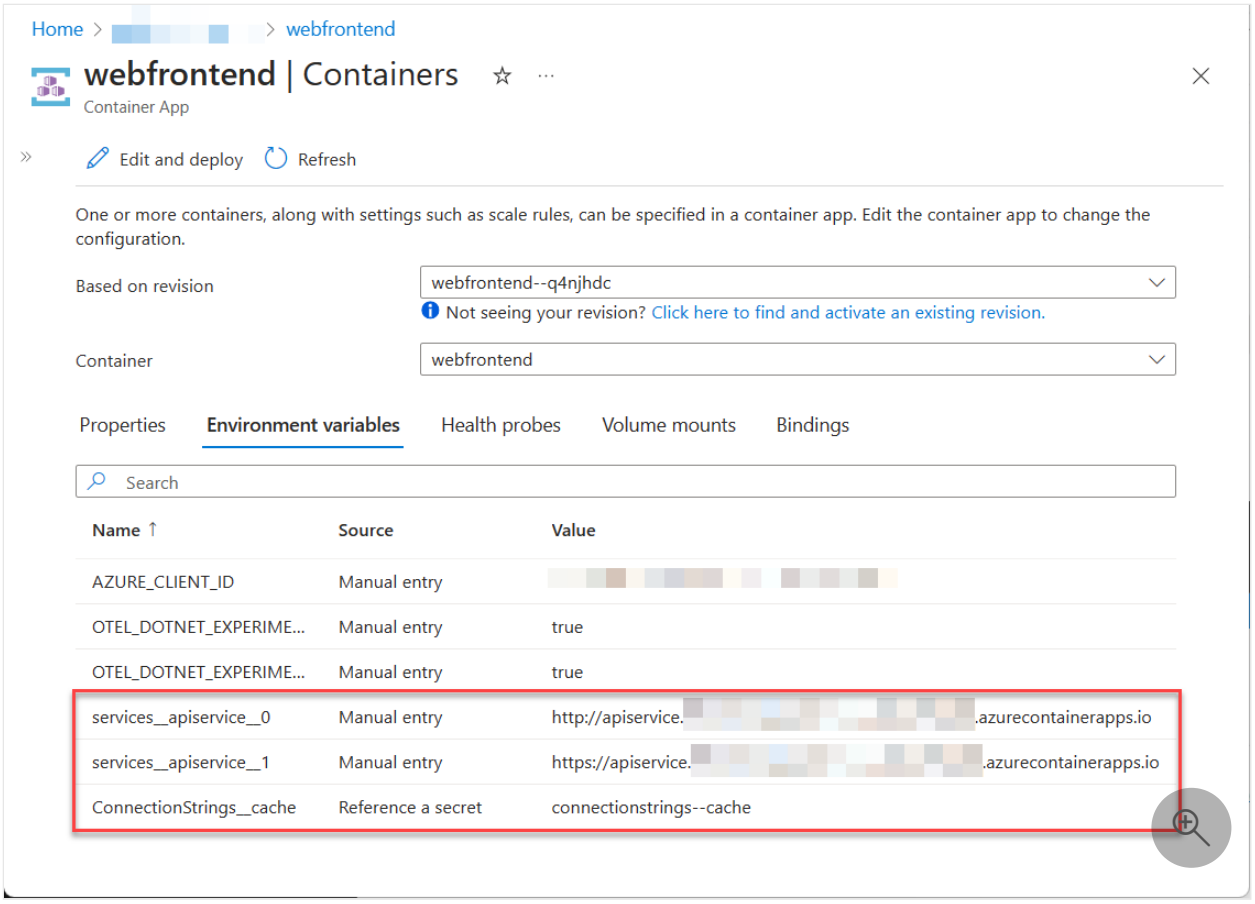


Three containers are deployed within this application:

- `webfrontend`: Contains code from the web project in the starter template.
- `apiservice`: Contains code from the API service project in the starter template.
- `cache`: A Redis container image to supply a cache to the front-end.

Just like in local development, the configuration of connection strings has been handled automatically. In this case, `azd` was responsible for interpreting the application model and translating it to the appropriate deployment steps. As an example, consider the

connection string and service discovery variables that are injected into the `webfrontend` container so that it knows how to connect to the Redis cache and `apiservice`.



For more information on how .NET Aspire projects handle connection strings and service discovery, see .NET Aspire orchestration overview.

Deploy application updates

When the `azd up` command is executed the underlying Azure resources are *provisioned* and a container image is built and *deployed* to the container apps hosting the .NET Aspire project. Typically once development is underway and Azure resources are deployed it won't be necessary to provision Azure resources every time code is updated—this is especially true for the developer inner loop.

To speed up deployment of code changes, `azd` supports deploying code updates in the container image. This is done using the `azd deploy` command:

Azure Developer CLI

```
azd deploy
```

Output

```
Deploying services (azd deploy)

  (✓) Done: Deploying service apiservice
  - Endpoint: <YOUR UNIQUE apiservice APP>.azurecontainerapps.io/

  (✓) Done: Deploying service webfrontend
  - Endpoint: <YOUR UNIQUE webfrontend APP>.azurecontainerapps.io/

Aspire Dashboard: <LINK TO DEPLOYED .NET ASPIRE DASHBOARD>
```

It's not necessary to deploy all services each time. `azd` understands the .NET Aspire project model, it's possible to deploy just one of the services specified using the following command:

Azure Developer CLI

```
azd deploy webfrontend
```

For more information, see Azure Developer CLI reference: azd deploy.

Deploy infrastructure updates

Whenever the dependency structure within a .NET Aspire project changes, `azd` must re-provision the underlying Azure resources. The `azd provision` command is used to apply these changes to the infrastructure.

To see this in action, update the *Program.cs* file in the AppHost project to the following:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache");

// Add the locations database.
var locationsdb = builder.AddPostgres("db").AddDatabase("locations");

// Add the locations database reference to the API service.
var apiservice = builder.AddProject<Projects.AspireSample_ApiService>
("apiservice")
    .WithReference(locationsdb);

builder.AddProject<Projects.AspireSample_Web>("webfrontend")
    .WithReference(cache)
    .WithReference(apiservice);
```

```
builder.Build().Run();
```

Save the file and issue the following command:

Azure Developer CLI

```
azd provision
```

The `azd provision` command updates the infrastructure by creating a container app to host the Postgres database. The `azd provision` command didn't update the connection strings for the `apiservice` container. In order to have connection strings updated to point to the newly provisioned Postgres database the `azd deploy` command needs to be invoked again. When in doubt, use `azd up` to both provision and deploy.

Clean up resources

Remember to clean up the Azure resources that you've created during this walkthrough. Because `azd knows the resource group in which it created the resources it can be used to spin down the environment using the following command:

Azure Developer CLI

```
azd down
```

The previous command may take some time to execute, but when completed the resource group and all its resources should be deleted.

Output

```
Deleting all resources and deployed code on Azure (azd down)
Local application code is not deleted when running 'azd down'.

  Resource group(s) to be deleted:

    • <YOUR RESOURCE GROUP>: <LINK TO RESOURCE GROUP OVERVIEW>

? Total resources to delete: 7, are you sure you want to continue? Yes
Deleting your resources can take some time.

  (✓) Done: Deleting resource group: <YOUR RESOURCE GROUP>

SUCCESS: Your application was removed from Azure in 9 minutes 59 seconds.
```

Generate Bicep from .NET Aspire project model

Although development teams are free to use `azd up` (or `azd provision` and `azd deploy`) commands for their deployments both for development and production purposes, some teams may choose to generate Bicep files that they can review and manage as part of version control (this also allows these Bicep files to be referenced as part of a larger more complex Azure deployment).

`azd` includes the ability to output the Bicep it uses for provisioning via following command:

Azure Developer CLI

```
azd config set alpha.infraSynth on
azd infra synth
```

After this command is executed in the starter template example used in this guide, the following files are created in the *AppHost* project directory:

- *infra/main.bicep*: Represents the main entry point for the deployment.
- *infra/main.parameters.json*: Used as the parameters for main Bicep (maps to environment variables defined in *.azure* folder).
- *infra/resources.bicep*: Defines the Azure resources required to support the .NET Aspire project model.
- *AspireSample.Web/manifests/containerApp.tmpl.yaml*: The container app definition for `webfrontend`.
- *AspireSample.ApiService/manifests/containerApp.tmpl.yaml*: The container app definition for `apiservice`.

The *infra\resources.bicep* file doesn't contain any definition of the container apps themselves (with the exception of container apps which are dependencies such as Redis and Postgres):

Bicep

```
@description('The location used for all deployed resources')
param location string = resourceGroup().location

@description('Tags that will be applied to all resources')
param tags object = {}

var resourceToken = uniqueString(resourceGroup().id)

resource managedIdentity
'Microsoft.ManagedIdentity/userAssignedIdentities@2023-01-31' = {
```

```
  name: 'mi-${resourceToken}'
  location: location
  tags: tags
}

resource containerRegistry 'Microsoft.ContainerRegistry/registries@2023-07-
01' = {
  name: replace('acr-${resourceToken}', '-', '')
  location: location
  sku: {
    name: 'Basic'
  }
  tags: tags
}

resource caeMiRoleAssignment 'Microsoft.Authorization/roleAssignments@2022-
04-01' = {
  name: guid(containerRegistry.id, managedIdentity.id,
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '7f951dda-
4ed3-4680-a7ca-43fe172d538d'))
  scope: containerRegistry
  properties: {
    principalId: managedIdentity.properties.principalId
    principalType: 'ServicePrincipal'
    roleDefinitionId:
subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '7f951dda-
4ed3-4680-a7ca-43fe172d538d')
  }
}

resource logAnalyticsWorkspace
'Microsoft.OperationalInsights/workspaces@2022-10-01' = {
  name: 'law-${resourceToken}'
  location: location
  properties: {
    sku: {
      name: 'PerGB2018'
    }
  }
  tags: tags
}

resource containerAppEnvironment 'Microsoft.App/managedEnvironments@2023-05-
01' = {
  name: 'cae-${resourceToken}'
  location: location
  properties: {
    appLogsConfiguration: {
      destination: 'log-analytics'
      logAnalyticsConfiguration: {
        customerId: logAnalyticsWorkspace.properties.customerId
        sharedKey: logAnalyticsWorkspace.listKeys().primarySharedKey
      }
    }
  }
```

```
  tags: tags
}

resource cache 'Microsoft.App/containerApps@2023-05-02-preview' = {
  name: 'cache'
  location: location
  properties: {
    environmentId: containerAppEnvironment.id
    configuration: {
      service: {
        type: 'redis'
      }
    }
    template: {
      containers: [
        {
          image: 'redis'
          name: 'redis'
        }
      ]
    }
  }
  tags: union(tags, {'aspire-resource-name': 'cache'})
}

resource locations 'Microsoft.App/containerApps@2023-05-02-preview' = {
  name: 'locations'
  location: location
  properties: {
    environmentId: containerAppEnvironment.id
    configuration: {
      service: {
        type: 'postgres'
      }
    }
    template: {
      containers: [
        {
          image: 'postgres'
          name: 'postgres'
        }
      ]
    }
  }
  tags: union(tags, {'aspire-resource-name': 'locations'})
}
output MANAGED_IDENTITY_CLIENT_ID string =
managedIdentity.properties.clientId
output AZURE_CONTAINER_REGISTRY_ENDPOINT string =
containerRegistry.properties.loginServer
output AZURE_CONTAINER_REGISTRY_MANAGED_IDENTITY_ID string =
managedIdentity.id
output AZURE_CONTAINER_APPS_ENVIRONMENT_ID string =
containerAppEnvironment.id
```

```
output AZURE_CONTAINER_APPS_ENVIRONMENT_DEFAULT_DOMAIN string =
containerAppEnvironment.properties.defaultDomain
```

For more information on using Bicep to automate deployments to Azure see, What is Bicep?

The definition of the container apps from the .NET service projects is contained within the *containerApp/tmpl.yaml* files in the `manifests` directory in each project respectively. Here is an example from the `webfrontend` project:

yml

```yaml
location: {{ .Env.AZURE_LOCATION }}
identity:
  type: UserAssigned
  userAssignedIdentities:
    ? "{{ .Env.AZURE_CONTAINER_REGISTRY_MANAGED_IDENTITY_ID }}"
    : {}
properties:
  environmentId: {{ .Env.AZURE_CONTAINER_APPS_ENVIRONMENT_ID }}
  configuration:
    activeRevisionsMode: single
    ingress:
      external: true
      targetPort: 8080
      transport: http
      allowInsecure: false
    registries:
    - server: {{ .Env.AZURE_CONTAINER_REGISTRY_ENDPOINT }}
      identity: {{ .Env.AZURE_CONTAINER_REGISTRY_MANAGED_IDENTITY_ID }}
  template:
    containers:
    - image: {{ .Env.SERVICE_WEBFRONTEND_IMAGE_NAME }}
      name: webfrontend
      env:
      - name: AZURE_CLIENT_ID
        value: {{ .Env.MANAGED_IDENTITY_CLIENT_ID }}
      - name: ConnectionStrings__cache
        value: {{ connectionString "cache" }}
      - name: OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EVENT_LOG_ATTRIBUTES
        value: "true"
      - name: OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EXCEPTION_LOG_ATTRIBUTES
        value: "true"
      - name: services__apiservice__0
        value: http://apiservice.internal.{{
.Env.AZURE_CONTAINER_APPS_ENVIRONMENT_DEFAULT_DOMAIN }}
      - name: services__apiservice__1
        value: https://apiservice.internal.{{
.Env.AZURE_CONTAINER_APPS_ENVIRONMENT_DEFAULT_DOMAIN }}
tags:
```

```
    azd-service-name: webfrontend
    aspire-resource-name: webfrontend
```

After executing the `azd infra synth` command, when `azd provision` and `azd deploy` are called they use the Bicep and supporting generated files.

> **ⓘ Important**
>
> If `azd infra synth` is called again, it replaces any modified files with freshly generated ones and prompts you for confirmation before doing so.

Isolated environments for debugging

Because `azd` makes it easy to provision new environments, it's possible for each team member to have an isolated cloud-hosted environment for debugging code in a setting that closely matches production. When doing this each team member should create their own environment using the following command:

Azure Developer CLI

```
azd env new
```

This will prompt the user for subscription and resource group information again and subsequent `azd up`, `azd provision`, and `azd deploy` invocations will use this new environment by default. The `--environment` switch can be applied to these commands to switch between environments.

Clean up resources

Run the following Azure CLI command to delete the resource group when you no longer need the Azure resources you created. Deleting the resource group also deletes the resources contained inside of it.

Azure CLI

```
az group delete --name <your-resource-group-name>
```

For more information, see Clean up resources in Azure.

Tutorial: Deploy a .NET Aspire project using the Azure Developer CLI

Article • 01/26/2025

The Azure Developer CLI (`azd`) enables you to deploy .NET Aspire projects using GitHub Actions or Azure Devops pipelines by automatically configuring the required authentication and environment settings. This article walks you through the process of creating and deploying a .NET Aspire project on Azure Container Apps using `azd`. You learn the following concepts:

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0
- An OCI compliant container runtime, such as:

```
```

Create a .NET Aspire solution

As a starting point, this article assumes that you've created a .NET Aspire solution from the **.NET Aspire Starter Application** template. For more information, see Quickstart: Build your first .NET Aspire app.

Initialize the template

1. Open a new terminal window and `cd` into the directory of your .NET Aspire solution.

2. Execute the `azd init` command to initialize your project with `azd`, which will inspect the local directory structure and determine the type of app.

   Azure Developer CLI

   ```
   azd init
   ```

   For more information on the `azd init` command, see azd init.

3. Select **Use code in the current directory** when `azd` prompts you with two app initialization options.

   Output

   ```
   ? How do you want to initialize your app?  [Use arrows to move, type to
   filter]
   > Use code in the current directory
     Select a template
   ```

4. After scanning the directory, `azd` prompts you to confirm that it found the correct .NET Aspire *AppHost* project. Select the **Confirm and continue initializing my app** option.

   Output

   ```
   Detected services:

     .NET (Aspire)
     Detected in:
   D:\source\repos\AspireSample\AspireSample.AppHost\AspireSample.AppHost.
   csproj

   azd will generate the files necessary to host your app on Azure using
   Azure Container Apps.
   ```

```
? Select an option  [Use arrows to move, type to filter]
> Confirm and continue initializing my app
  Cancel and exit
```

5. Enter an environment name, which is used to name provisioned resources in Azure and managing different environments such as `dev` and `prod`.

Output

```
Generating files to run your app on Azure:

  (✓) Done: Generating ./azure.yaml
  (✓) Done: Generating ./next-steps.md

SUCCESS: Your app is ready for the cloud!
You can provision and deploy your app to Azure by running the azd up
command in this directory. For more information on configuring your
app, see ./next-steps.md
```

`azd` generates a number of files and places them into the working directory. These files are:

- *azure.yaml*: Describes the services of the app, such as .NET Aspire AppHost project, and maps them to Azure resources.
- *.azure/config.json*: Configuration file that informs `azd` what the current active environment is.
- *.azure/aspireazddev/.env*: Contains environment specific overrides.

Create the GitHub repository and pipeline

The Azure Developer CLI enables you to automatically create CI/CD pipelines with the correct configurations and permissions to provision and deploy resources to Azure. `azd` can also create a GitHub repository for your app if it doesn't exist already.

1. Run the `azd pipeline config` command to configure your deployment pipeline and securely connect it to Azure:

Azure Developer CLI

```
azd pipeline config
```

2. Select the subscription to provision and deploy the app resources to.

3. Select the Azure location to use for the resources.

4. When prompted whether to create a new Git repository in the directory, enter `y` and press `Enter`.

   > **Note**
   >
   > Creating a GitHub repository required you being logged into GitHub. There are a few selections that vary based on your preferences. After logging in, you will be prompted to create a new repository in the current directory.

5. Select **Create a new private GitHub repository** to configure the git remote.

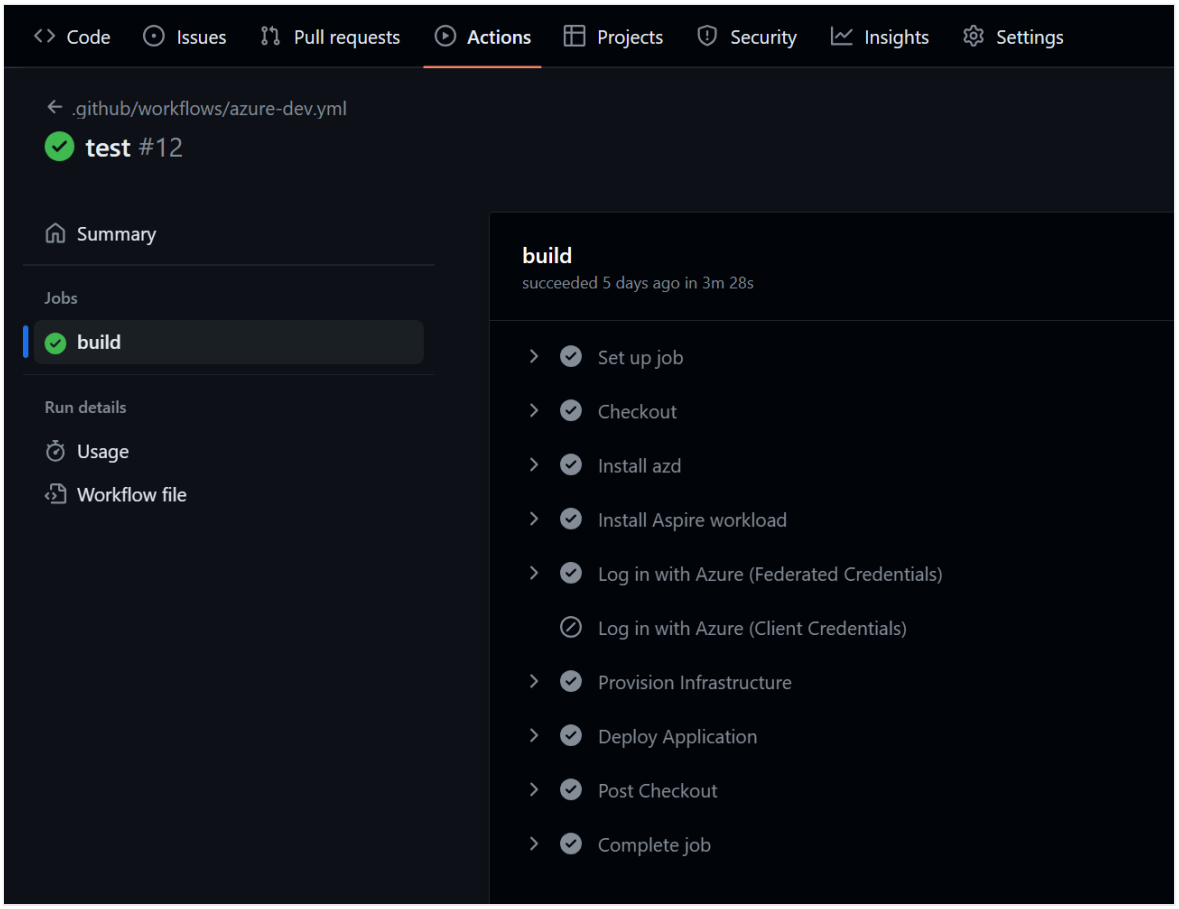
6. Enter a name of your choice for the new GitHub repository or press enter to use the default name. `azd` creates a new repository in GitHub and configures it with the necessary secrets required to authenticate to Azure.

7. Enter `y` to proceed when `azd` prompts you to commit and push your local changes to start the configured pipeline.

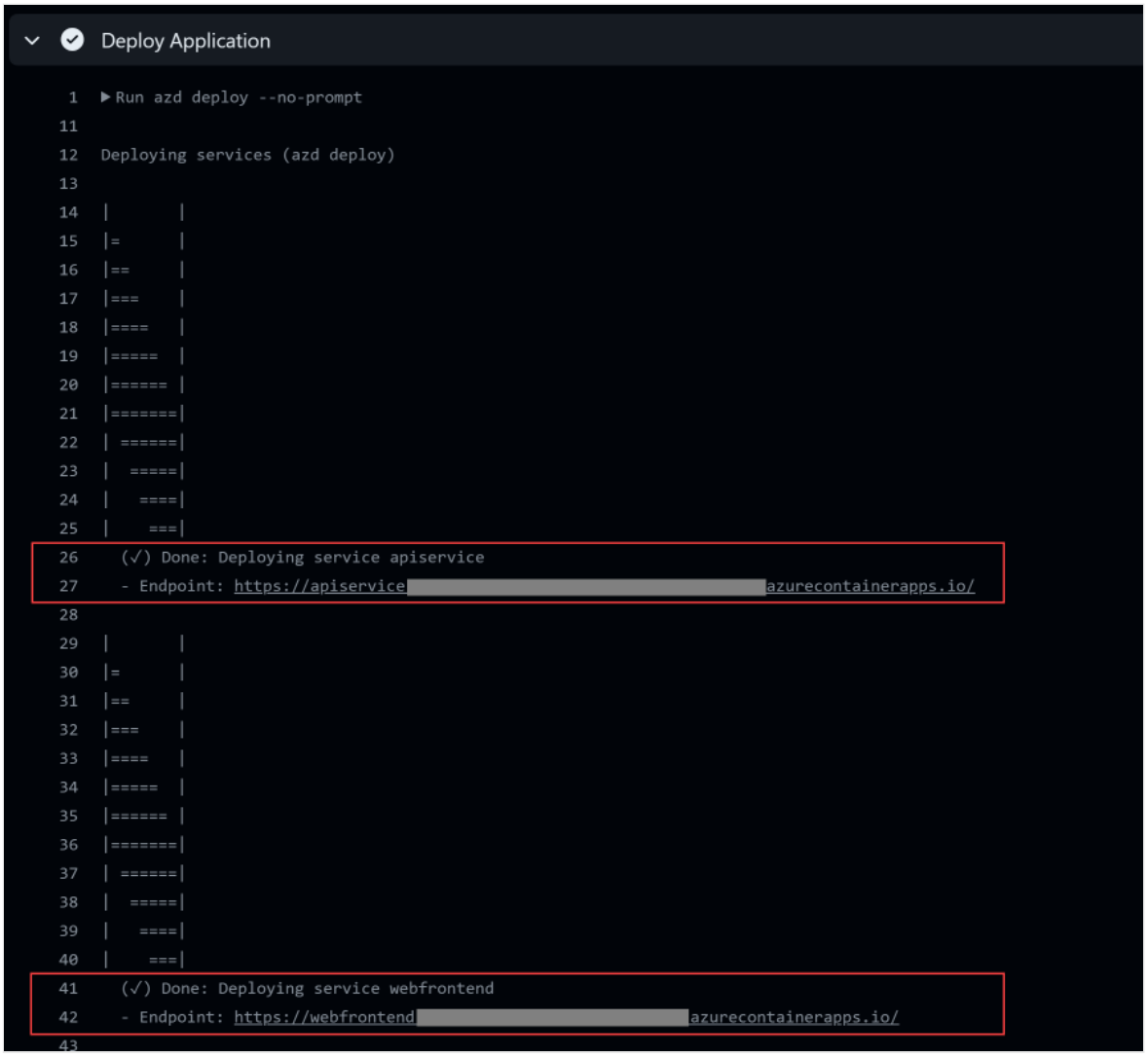
Explore the GitHub Actions workflow and deployment

1. Navigate to your new GitHub repository using the link output by `azd`.

2. Select the **Actions** tab to view the repository workflows. You should see the new workflow either running or already completed. Select the workflow to view the job steps and details in the logs of the run. For example, you can expand steps such as **Install .NET Aspire Workload** or **Deploy application** to see the details of the completed action.



3. Select **Deploy Application** to expand the logs for that step. You should see two endpoint urls printed out for the `apiservice` and `webfrontend`. Select either of these links to open them in another browser tab and explore the deployed application.



Congratulations! You successfully deployed a .NET Aspire project using the Azure Developer CLI and GitHub Actions.

Clean up resources

Run the following Azure CLI command to delete the resource group when you no longer need the Azure resources you created. Deleting the resource group also deletes the resources contained inside of it.

Azure CLI

```
az group delete --name <your-resource-group-name>
```

For more information, see Clean up resources in Azure.

Use Application Insights for .NET Aspire telemetry

Article • 04/12/2024

Azure Application Insights, a feature of Azure Monitor, excels in Application Performance Management (APM) for live web applications. .NET Aspire projects are designed to use OpenTelemetry for application telemetry. OpenTelemetry supports an extension model to support sending data to different APMs. .NET Aspire uses OTLP by default for telemetry export, which is used by the dashboard during development. Azure Monitor doesn't (yet) support OTLP, so the applications need to be modified to use the Azure Monitor exporter, and configured with the connection string.

To use Application insights, you specify its configuration in the app host project *and* use the Azure Monitor distro in the service defaults project.

Choosing how Application Insights is provisioned

.NET Aspire has the capability to provision cloud resources as part of cloud deployment, including Application Insights. In your .NET Aspire project, you can decide if you want .NET Aspire to provision an Application Insights resource when deploying to Azure. You can also select to use an existing Application Insights resource by providing its connection string. The connection information is managed by the resource configuration in the app host project.

Provisioning Application insights during Azure deployment

With this option, an instance of Application Insights will be created for you when the application is deployed using the Azure Developer CLI (`azd`).

To use automatic provisioning, you specify a dependency in the app host project, and reference it in each project/resource that needs to send telemetry to Application Insights. The steps include:

- Add a Nuget package reference to Aspire.Hosting.Azure.ApplicationInsights in the app host project.

- Update the app host code to use the Application Insights resource, and reference it from each project:

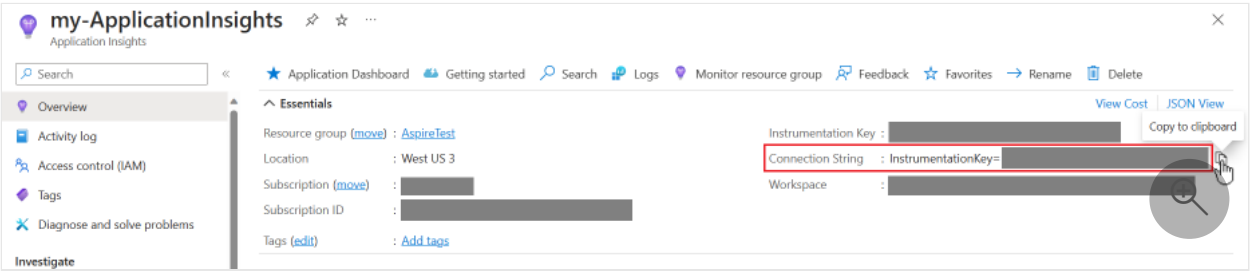
C#

```
var builder = DistributedApplication.CreateBuilder(args);

// Automatically provision an Application Insights resource
var insights = builder.AddAzureApplicationInsights("MyApplicationInsights");

// Reference the resource from each project
var apiService = builder.AddProject<Projects.ApiService>("apiservice")
    .WithReference(insights);

builder.AddProject<Projects.Web>("webfrontend")
    .WithReference(apiService)
    .WithReference(insights);

builder.Build().Run();
```

Follow the steps in Deploy a .NET Aspire project to Azure Container Apps using the Azure Developer CLI (in-depth guide) to deploy the application to Azure Container Apps. `azd` will create an Application Insights resource as part of the same resource group, and configure the connection string for each container.

Manual provisioning of Application Insights resource

Application Insights uses a connection string to tell the OpenTelemetry exporter where to send the telemetry data. The connection string is specific to the instance of Application Insights you want to send the telemetry to. It can be found in the Overview page for the application insights instance.



If you wish to use an instance of Application Insights that you have provisioned manually, then you should use the `AddConnectionString` API in the app host project to tell the projects/containers where to send the telemetry data. The Azure Monitor distro expects the environment variable to be `APPLICATIONINSIGHTS_CONNECTION_STRING`, so that needs to be explicitly set when defining the connection string.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var insights = builder.AddConnectionString(
    "myInsightsResource",
    "APPLICATIONINSIGHTS_CONNECTION_STRING");

var apiService = builder.AddProject<Projects.ApiService>("apiservice")
    .WithReference(insights);

builder.AddProject<Projects.Web>("webfrontend")
    .WithReference(apiService)
    .WithReference(insights);

builder.Build().Run();
```

Resource usage during development

When running the .NET Aspire project locally, the preceding code reads the connection string from configuration. As this is a secret, you should store the value in app secrets. Right click on the app host project and choose **Manage Secrets** from the context menu to open the secrets file for the app host project. In the file add the key and your specific connection string, the example below is for illustration purposes.

JSON

```
{
  "ConnectionStrings": {
    "myInsightsResource": "InstrumentationKey=12345678-abcd-1234-abcd-
1234abcd5678;IngestionEndpoint=https://westus3-
1.in.applicationinsights.azure.com"
  }
}
```

> **Note**
>
> The `name` specified in the app host code needs to match a key inside the `ConnectionStrings` section in the settings file.

Resource usage during deployment

When deploying an Aspire application with Azure Developer CLI (azd), it will recognize the connection string resource and prompt for a value. This enables a different resource

to be used for the deployment from the value used for local development.

Mixed deployment

If you wish to use a different deployment mechanism per execution context, use the appropriate API conditionally. For example, the following code uses a pre-supplied connection at development time, and an automatically provisioned resource at deployment time.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var insights = builder.ExecutionContext.IsPublishMode
    ? builder.AddAzureApplicationInsights("myInsightsResource")
    : builder.AddConnectionString("myInsightsResource",
"APPLICATIONINSIGHTS_CONNECTION_STRING");

var apiService = builder.AddProject<Projects.ApiService>("apiservice")
    .WithReference(insights);

builder.AddProject<Projects.Web>("webfrontend")
    .WithReference(apiService)
    .WithReference(insights);

builder.Build().Run();
```

> **Tip**
>
> The preceding code requires you to supply the connection string information in app secrets for development time usage, and will be prompted for the connection string by `azd` at deployment time.

Use the Azure Monitor distro

To make exporting to Azure Monitor simpler, this example uses the Azure Monitor Exporter Repo. This is a wrapper package around the Azure Monitor OpenTelemetry Exporter package that makes it easier to export to Azure Monitor with a set of common defaults.

Add the following package to the `ServiceDefaults` project, so that it will be included in each of the .NET Aspire services. For more information, see .NET Aspire service defaults.

XML

```
<PackageReference Include="Azure.Monitor.OpenTelemetry.AspNetCore"
                  Version="*" />
```

Add a using statement to the top of the project.

C#

```
using Azure.Monitor.OpenTelemetry.AspNetCore;
```

Uncomment the line in `AddOpenTelemetryExporters` to use the Azure Monitor exporter:

C#

```
private static IHostApplicationBuilder AddOpenTelemetryExporters(
    this IHostApplicationBuilder builder)
{
    // Omitted for brevity...

    // Uncomment the following lines to enable the Azure Monitor exporter
    // (requires the Azure.Monitor.OpenTelemetry.AspNetCore package)
    if
(!string.IsNullOrEmpty(builder.Configuration["APPLICATIONINSIGHTS_CONNECTION
_STRING"]))
    {
        builder.Services.AddOpenTelemetry().UseAzureMonitor();
    }
    return builder;
}
```

It's possible to further customize the Azure Monitor exporter, including customizing the resource name and changing the sampling. For more information, see Customize the Azure Monitor exporter. Using the parameterless version of `UseAzureMonitor()`, will pickup the connection string from the `APPLICATIONINSIGHTS_CONNECTION_STRING` environment variable, we configured via the app host project.

Tutorial: Deploy a .NET Aspire project with a SQL Server Database to Azure

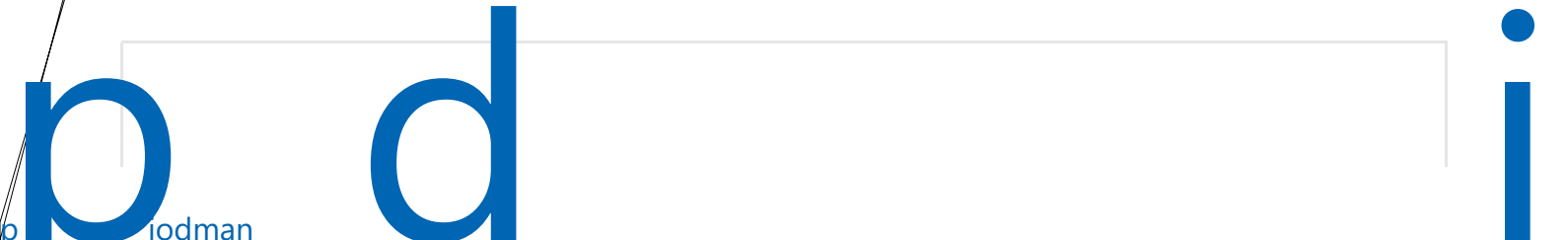
Article • 04/04/2025

In this tutorial, you learn to configure an ASP.NET Core app with a SQL Server Database for deployment to Azure. .NET Aspire provides multiple SQL Server integration configurations that provision different database services in Azure. You'll learn how to:

Prerequisites

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0
- An OCI compliant container runtime, such as:
  - Docker Desktop or Podman. For more information, see Container runtime.
- An Integrated Developer Environment (IDE) or code editor, such as:



2. In the dialog window, search for *Aspire* and select **.NET Aspire Starter App**. Choose **Next**.
3. On the **Configure your new project** screen:

    - Enter a **Solution name** of **AspireSql**.
    - Leave the rest of the values at their defaults and select **Next**.

4. On the **Additional information** screen:

    - In the **Framework** list, verify that **.NET 9.0** is selected.
    - In the **.NET Aspire version** list, verify that **9.1** is selected.
    - Choose **Create**.

Visual Studio creates a new ASP.NET Core solution that is structured to use .NET Aspire. The solution consists of the following projects:

- **AspireSql.ApiService**: An API project that depends on service defaults.
- **AspireSql.AppHost**: An orchestrator project designed to connect and configure the different projects and services of your app. The orchestrator should be set as the startup project.
- **AspireSql.ServiceDefaults**: A shared class library to hold configurations that can be reused across the projects in your solution.
- **AspireSql.Web**: A Blazor project that depends on service defaults.

Configure the app for SQL Server deployment

.NET Aspire provides two built-in configuration options to streamline SQL Server deployment on Azure:

- Provision a containerized SQL Server database using Azure Container Apps
- Provision an Azure SQL Database instance

Add the .NET Aspire integration to the app

Add the appropriate .NET Aspire integration to the *AspireSql.AppHost* project for your desired hosting service.

Azure SQL Database

Open a command prompt and add the Aspire.Hosting.Azure.Sql NuGet package to the *AspireSql.AppHost* project:

.NET CLI

```
cd AspireSql.AppHost
dotnet add package Aspire.Hosting.Azure.Sql
```

Configure the AppHost project

Configure the *AspireSql.AppHost* project for your desired SQL database service.

Azure SQL Database

Replace the contents of the *Program.cs* file in the *AspireSql.AppHost* project with the following code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var apiService = builder.AddProject<Projects.AspireSql_ApiService>
("apiservice");

// Provisions an Azure SQL Database when published
var sqlServer = builder.AddAzureSqlServer("sqlserver")
                       .AddDatabase("sqldb");

builder.AddProject<Projects.AspireSql_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(apiService)
    .WaitFor(apiService);

builder.Build().Run();
```

The preceding code adds an Azure SQL Server resource to your app and configures a connection to a database called `sqldb`. The AddAzureSqlServer method ensures that tools such as the Azure Developer CLI or Visual Studio create an Azure SQL Database resource during the deployment process.

Deploy the app

Tools such as the Azure Developer CLI (`azd`) support .NET Aspire SQL Server integration configurations to streamline deployments. `azd` consumes these settings and provisions properly configured resources for you.

Initialize the template

1. Open a new terminal window and `cd` into the directory of your .NET Aspire solution.

2. Execute the `azd init` command to initialize your project with `azd`, which will inspect the local directory structure and determine the type of app.

   Azure Developer CLI

   ```
   azd init
   ```

   For more information on the `azd init` command, see azd init.

3. Select **Use code in the current directory** when `azd` prompts you with three app initialization options.

   Output

   ```
   ? How do you want to initialize your app?  [Use arrows to move, type to
   filter]
   > Use code in the current directory
     Select a template
     Create a minimal project
   ```

4. After scanning the directory, `azd` prompts you to confirm that it found the correct .NET Aspire *AppHost* project. Select the **Confirm and continue initializing my app** option.

   Output

   ```
   Detected services:

     .NET (Aspire)
     Detected in:
   D:\source\repos\AspireSample\AspireSample.AppHost\AspireSample.AppHost.csproj

   azd will generate the files necessary to host your app on Azure using Azure
   Container Apps.

   ? Select an option  [Use arrows to move, type to filter]
   > Confirm and continue initializing my app
     Cancel and exit
   ```

5. Enter an environment name, which is used to name provisioned resources in Azure and managing different environments such as `dev` and `prod`.

   Output

```
Generating files to run your app on Azure:

  (✓) Done: Generating ./azure.yaml
  (✓) Done: Generating ./next-steps.md

SUCCESS: Your app is ready for the cloud!
You can provision and deploy your app to Azure by running the azd up command
in this directory. For more information on configuring your app, see ./next-
steps.md
```

`azd` generates a number of files and places them into the working directory. These files are:

- *azure.yaml*: Describes the services of the app, such as .NET Aspire AppHost project, and maps them to Azure resources.
- *.azure/config.json*: Configuration file that informs `azd` what the current active environment is.
- *.azure/aspireazddev/.env*: Contains environment specific overrides.

Deploy the template

1. Once an `azd` template is initialized, the provisioning and deployment process can be executed as a single command from the *AppHost* project directory using azd up:

   Azure Developer CLI

   ```
   azd up
   ```

2. Select the subscription you'd like to deploy to from the list of available options:

   Output

   ```
   Select an Azure Subscription to use:  [Use arrows to move, type to filter]
     1. SampleSubscription01 (xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx)
     2. SampleSubscription02 (xxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx)
   ```

3. Select the desired Azure location to use from the list of available options:

   Output

   ```
   Select an Azure location to use:  [Use arrows to move, type to filter]
     42. (US) Central US (centralus)
     43. (US) East US (eastus)
   > 44. (US) East US 2 (eastus2)
   ```

```
   46. (US) North Central US (northcentralus)
   47. (US) South Central US (southcentralus)
```

After you make your selections, `azd` executes the provisioning and deployment process.

Output

```
By default, a service can only be reached from inside the Azure Container Apps
environment it is running in. Selecting a service here will also allow it to be
reached from the Internet.
? Select which services to expose to the Internet webfrontend
? Select an Azure Subscription to use:  1. <YOUR SUBSCRIPTION>
? Select an Azure location to use: 1. <YOUR LOCATION>

Packaging services (azd package)


Provisioning Azure resources (azd provision)
Provisioning Azure resources can take some time.

Subscription: <YOUR SUBSCRIPTION>
Location: <YOUR LOCATION>

  You can view detailed progress in the Azure Portal:
  <LINK TO DEPLOYMENT>

  (✓) Done: Resource group: <YOUR RESOURCE GROUP>
  (✓) Done: Container Registry: <ID>
  (✓) Done: Log Analytics workspace: <ID>
  (✓) Done: Container Apps Environment: <ID>

SUCCESS: Your application was provisioned in Azure in 1 minute 13 seconds.
You can view the resources created under the resource group <YOUR RESOURCE GROUP>
in Azure Portal:
<LINK TO RESOURCE GROUP OVERVIEW>

Deploying services (azd deploy)

  (✓) Done: Deploying service apiservice
  - Endpoint: <YOUR UNIQUE apiservice APP>.azurecontainerapps.io/

  (✓) Done: Deploying service webfrontend
  - Endpoint: <YOUR UNIQUE webfrontend APP>.azurecontainerapps.io/

Aspire Dashboard: <LINK TO DEPLOYED .NET ASPIRE DASHBOARD>

SUCCESS: Your up workflow to provision and deploy to Azure completed in 3 minutes
50 seconds.
```

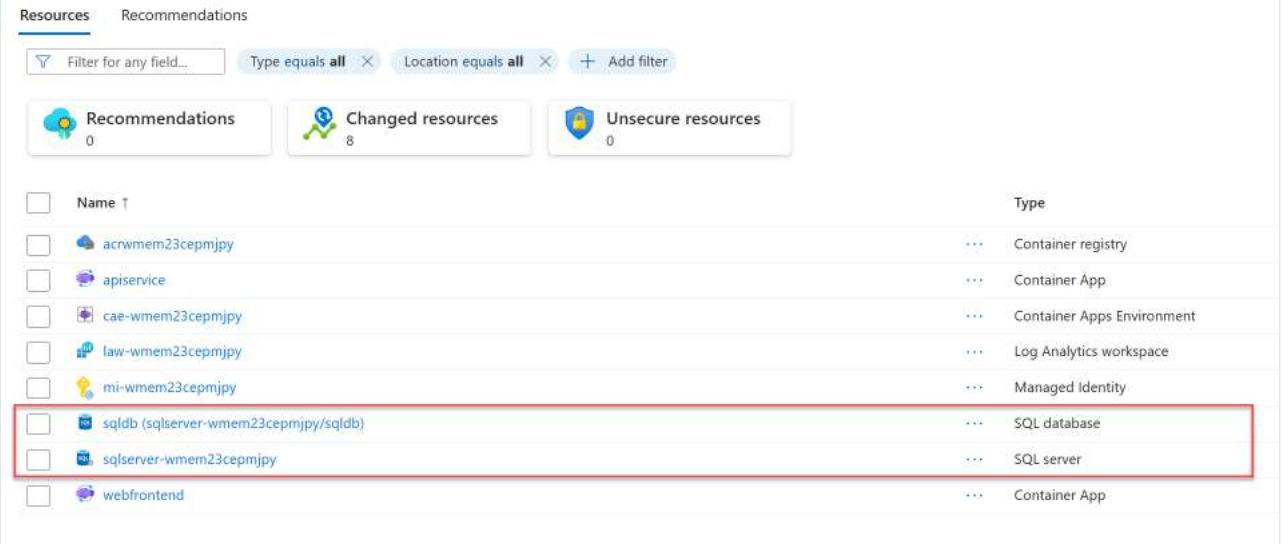
The `azd up` command acts as wrapper for the following individual `azd` commands to provision and deploy your resources in a single step:

1. azd package: The app projects and their dependencies are packaged into containers.
2. azd provision: The Azure resources the app will need are provisioned.
3. azd deploy: The projects are pushed as containers into an Azure Container Registry instance, and then used to create new revisions of Azure Container Apps in which the code will be hosted.

When the `azd up` stages complete, your app will be available on Azure, and you can open the Azure portal to explore the resources. `azd` also outputs URLs to access the deployed apps directly.

Azure SQL Database

The deployment process provisioned an Azure SQL Database resource due to the **.AppHost** configuration you provided.



Clean up resources

Run the following Azure CLI command to delete the resource group when you no longer need the Azure resources you created. Deleting the resource group also deletes the resources contained inside of it.

Azure CLI

```
az group delete --name <your-resource-group-name>
```

For more information, see Clean up resources in Azure.

See also

- Deploy a .NET Aspire project to Azure Container Apps
- Deploy a .NET Aspire project to Azure Container Apps using the Azure Developer CLI (in-depth guide)
- Tutorial: Deploy a .NET Aspire project using the Azure Developer CLI

Tutorial: Deploy a .NET Aspire project with a Redis Cache to Azure

Article • 11/12/2024

In this tutorial, you learn to configure a .NET Aspire project with a Redis Cache for deployment to Azure. .NET Aspire provides multiple caching integration configurations that provision different Redis services in Azure. You'll learn how to:

- ✓ Configure the app to provision an Azure Cache for Redis
- ✓ Configure the app to provision a containerized Redis Cache

> **Note**
>
> This document focuses specifically on .NET Aspire configurations to provision and deploy Redis Cache resources in Azure. For more information and to learn more about the full .NET Aspire deployment process, see the **Azure Container Apps deployment** tutorial.

Prerequisites

To work with .NET Aspire, you need the following installed locally:

- .NET 8.0 or .NET 9.0
- An OCI compliant container runtime, such as:
  - Docker Desktop or Podman. For more information, see Container runtime.
- An Integrated Developer Environment (IDE) or code editor, such as:
  - Visual Studio 2022 version 17.9 or higher (Optional)
  - Visual Studio Code (Optional)
    - C# Dev Kit: Extension (Optional)
  - JetBrains Rider with .NET Aspire plugin (Optional)

For more information, see .NET Aspire setup and tooling, and .NET Aspire SDK.

Create the sample solution

Follow the Tutorial: Implement caching with .NET Aspire integrations to create the sample project.

Configure the app for Redis cache deployment

.NET Aspire provides two built-in configuration options to streamline Redis Cache deployment on Azure:

- Provision a containerized Redis Cache using Azure Container Apps
- Provision an Azure Cache for Redis instance

Add the .NET Aspire integration to the app

Add the appropriate .NET Aspire integration to the *AspireRedis.AppHost* project for your desired hosting service.

Azure Cache for Redis

Add the Aspire.Hosting.Azure.Redis NuGet package to the *AspireRedis.AppHost* project:

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Redis
```

Configure the AppHost project

Configure the *AspireRedis.AppHost* project for your desired Redis service.

Azure Cache for Redis

Replace the contents of the *Program.cs* file in the *AspireRedis.AppHost* project with the following code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddAzureRedis("cache");

var apiService = builder.AddProject<Projects.AspireRedis_ApiService>
("apiservice")
                        .WithReference(cache);

builder.AddProject<Projects.AspireRedis_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(cache)
```

```
    .WaitFor(cache)
    .WithReference(apiService)
    .WaitFor(apiService);

builder.Build().Run();
```

The preceding code adds an Azure Cache for Redis resource to your app and configures a connection called `cache`. The `AddAzureRedis` method ensures that tools such as the Azure Developer CLI or Visual Studio create an Azure Cache for Redis resource during the deployment process.

Tools such as the Azure Developer CLI (`azd`) support .NET Aspire Redis integration configurations to streamline deployments. `azd` consumes these settings and provisions properly configured resources for you.

1. Open a terminal window in the root of your .NET Aspire project.
2. Run the `azd init` command to initialize the project with `azd`.

   Azure Developer CLI

   ```
   
   ```

3. When prompted for an environment name, enter *docs-aspireredis*.
4. Run the `azd up` command to begin the deployment process:

   Azure Developer CLI

   ```
   
   ```

5. Select the Azure subscription that should host your app resources.

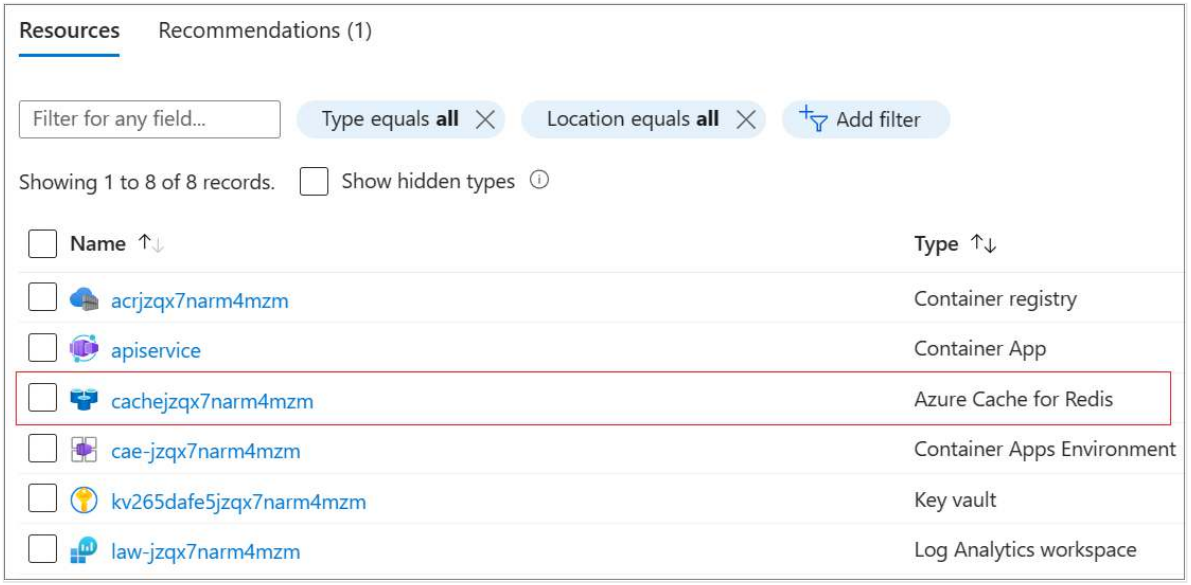
6. Select the Azure location to use.

   The Azure Developer CLI provisions and deploys your app resources. The process may take a few minutes to complete.

7. When the deployment finishes, click the resource group link in the output to view the created resources in the Azure portal.

Azure Cache for Redis

The deployment process provisioned an Azure Cache for Redis resource due to the **.AppHost** configuration you provided.



Clean up resources

Run the following Azure CLI command to delete the resource group when you no longer need the Azure resources you created. Deleting the resource group also deletes the resources contained inside of it.

Azure CLI

```
az group delete --name <your-resource-group-name>
```

For more information, see Clean up resources in Azure.

See also

- .NET Aspire deployment via Azure Container Apps

- .NET Aspire Azure Container Apps deployment deep dive
- Deploy a .NET Aspire project using GitHub Actions

.NET Aspire manifest format for deployment tool builders

Article • 03/29/2024

In this article, you learn about the .NET Aspire manifest format. This article serves as a reference guide for deployment tool builders, aiding in the creation of tooling to deploy .NET Aspire projects on specific hosting platforms, whether on-premises or in the cloud.

.NET Aspire simplifies the local development experience by helping to manage interdependencies between application integrations. To help simplify the deployment of applications, .NET Aspire projects can generate a manifest of all the resources defined as a JSON formatted file.

Generate a manifest

A valid .NET Aspire project is required to generate a manifest. To get started, create a .NET Aspire project using the `aspire-starter` .NET template:

.NET CLI

```
dotnet new aspire-starter --use-redis-cache `
    -o AspireApp && `
    cd AspireApp
```

Manifest generation is achieved by running `dotnet build` with a special target:

.NET CLI

```
dotnet run --project AspireApp.AppHost\AspireApp.AppHost.csproj `
    --publisher manifest `
    --output-path ../aspire-manifest.json
```

> **Tip**
>
> The `--output-path` supports relative paths. The previous command uses `../aspire-manifest.json` to place the manifest file in the root of the project directory.

For more information, see dotnet run. The previous command produces the following output:

Output

```
Building...
info: Aspire.Hosting.Publishing.ManifestPublisher[0]
      Published manifest to: .\AspireApp.AppHost\aspire-manifest.json
```

The file generated is the .NET Aspire manifest and is used by tools to support deploying into target cloud environments.

> **Note**
>
> You can also generate a manifest as part of the launch profile. Consider the following *launchSettings.json*:
>
> JSON
>
> ```
> {
>   "$schema": "http://json.schemastore.org/launchsettings.json",
>   "profiles": {
>     "generate-manifest": {
>       "commandName": "Project",
>       "launchBrowser": false,
>       "dotnetRunMessages": true,
>       "commandLineArgs": "--publisher manifest --output-path aspire-
> manifest.json"
>     }
>   }
> }
> ```

Basic manifest format

Publishing the manifest from the default starter template for .NET Aspire produces the following JSON output:

JSON

```
{
  "resources": {
    "cache": {
      "type": "container.v0",
      "connectionString": "{cache.bindings.tcp.host}:
{cache.bindings.tcp.port}",
      "image": "redis:7.2.4",
      "bindings": {
        "tcp": {
          "scheme": "tcp",
```

```
          "protocol": "tcp",
          "transport": "tcp",
          "containerPort": 6379
        }
      }
    },
    "apiservice": {
      "type": "project.v0",
      "path": "../AspireApp.ApiService/AspireApp.ApiService.csproj",
      "env": {
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EXCEPTION_LOG_ATTRIBUTES":
"true",
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EVENT_LOG_ATTRIBUTES": "true"
      },
      "bindings": {
        "http": {
          "scheme": "http",
          "protocol": "tcp",
          "transport": "http"
        },
        "https": {
          "scheme": "https",
          "protocol": "tcp",
          "transport": "http"
        }
      }
    },
    "webfrontend": {
      "type": "project.v0",
      "path": "../AspireApp.Web/AspireApp.Web.csproj",
      "env": {
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EXCEPTION_LOG_ATTRIBUTES":
"true",
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EVENT_LOG_ATTRIBUTES": "true",
        "ConnectionStrings__cache": "{cache.connectionString}",
        "services__apiservice__0": "{apiservice.bindings.http.url}",
        "services__apiservice__1": "{apiservice.bindings.https.url}"
      },
      "bindings": {
        "http": {
          "scheme": "http",
          "protocol": "tcp",
          "transport": "http"
        },
        "https": {
          "scheme": "https",
          "protocol": "tcp",
          "transport": "http"
        }
      }
    }
  }
}
```

The manifest format JSON consists of a single object called `resources`, which contains a property for each resource specified in *Program.cs* (the `name` argument for each name is used as the property for each of the child resource objects in JSON).

Connection string and binding references

In the previous example, there are two project resources and one Redis cache resource. The *webfrontend* depends on both the *apiservice* (project) and *cache* (Redis) resources.

This dependency is known because the environment variables for the *webfrontend* contain placeholders that reference the two other resources:

JSON

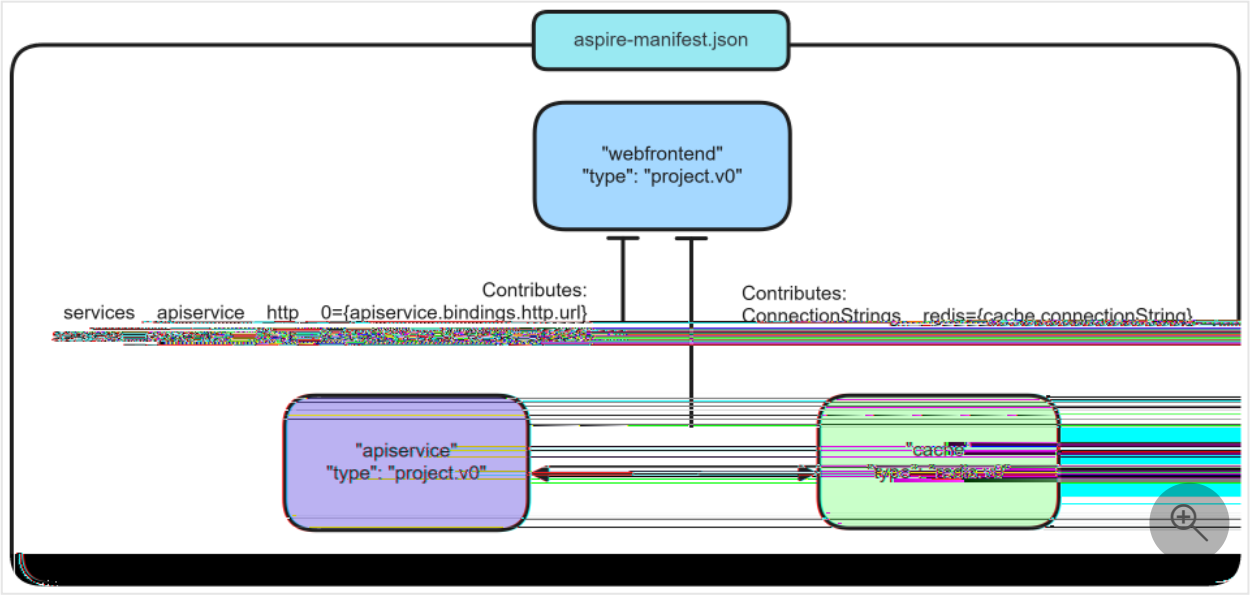
```
"env": {
  // ... other environment variables omitted for clarity
  "ConnectionStrings__cache": "{cache.connectionString}",
  "services__apiservice__0": "{apiservice.bindings.http.url}",
  "services__apiservice__1": "{apiservice.bindings.https.url}"
},
```

The `apiservice` resource is referenced by `webfrontend` using the call `WithReference(apiservice)` in the app host *Program.cs* file and `redis` is referenced using the call `WithReference(cache)`:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache");

var apiService = builder.AddProject<Projects.AspireApp_ApiService>
("apiservice");

builder.AddProject<Projects.AspireApp_Web>("webfrontend")
    .WithReference(cache)
    .WithReference(apiService);

builder.Build().Run();
```

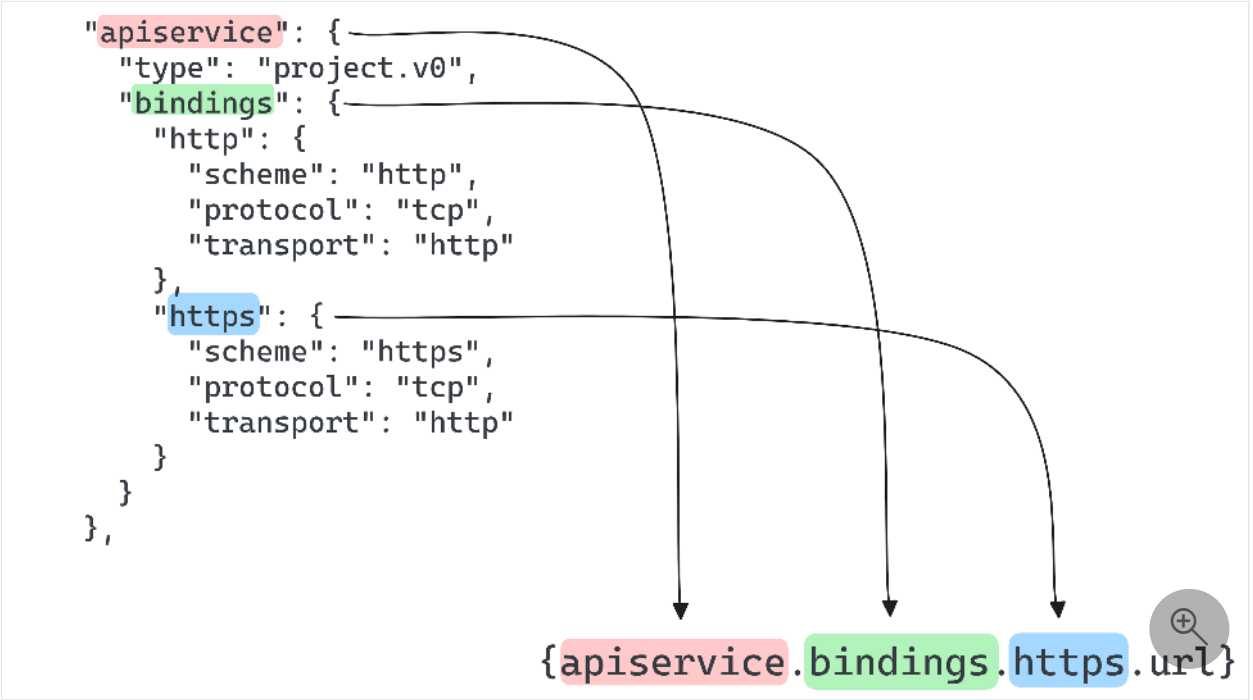
References between project resource types result in service discovery variables being injected into the referencing project. References to well known reference types such as Redis result in connection strings being injected.



For more information on how resources in the app model and references between them work, see, .NET Aspire orchestration overview.

Placeholder string structure

Placeholder strings reference the structure of the .NET Aspire manifest:



The final segment of the placeholder string (`url` in this case) is generated by the tool processing the manifest. There are several suffixes that could be used on the placeholder string:

- `connectionString`: For well-known resource types such as Redis. Deployment tools translate the resource in the most appropriate infrastructure for the target cloud environment and then produce a .NET Aspire compatible connection string for the

consuming application to use. On `container.v0` resources the `connectionString` field may be present and specified explicitly. This is to support scenarios where a container resource type is referenced using the WithReference extension but is desired to be hosted explicitly as a container.

- `url`: For service-to-service references where a well-formed URL is required. The deployment tool produces the `url` based on the scheme, protocol, and transport defined in the manifest and the underlying compute/networking topology that was deployed.
- `host`: The host segment of the URL.
- `port`: The port segment of the URL.

Resource types

Each resource has a `type` field. When a deployment tool reads the manifest, it should read the type to verify whether it can correctly process the manifest. During the .NET Aspire preview period, all resource types have a `v0` suffix to indicate that they're subject to change. As .NET Aspire approaches release a `v1` suffix will be used to signify that the structure of the manifest for that resource type should be considered stable (subsequent updates increment the version number accordingly).

Common resource fields

The `type` field is the only field that is common across all resource types, however, the `project.v0`, `container.v0`, and `executable.v0` resource types also share the `env` and `bindings` fields.

> ⓘ Note
>
> The `executable.v0` resource type isn't fully implemented in the manifest due to its lack of utility in deployment scenarios. For more information on containerizing executables, see Dockerfile resource types.

The `env` field type is a basic key/value mapping where the values might contain *placeholder strings*.

Bindings are specified in the `bindings` field with each binding contained within its own field under the `bindings` JSON object. The fields omitted by the .NET Aspire manifest in the `bindings` node include:

- `scheme`: One of the following values `tcp`, `udp`, `http`, or `https`.
- `protocol`: One of the following values `tcp` or `udp`
- `transport`: Same as `scheme`, but used to disambiguate between `http` and `http2`.
- `containerPort`: Optional, if omitted defaults to port 80.

The `inputs` field

Some resources generate an `inputs` field. This field is used to specify input parameters for the resource. The `inputs` field is a JSON object where each property is an input parameter that's used in placeholder structure resolution. Resources that have a `connectionString`, for example, might use the `inputs` field to specify a `password` for the connection string:

JSON

```
"connectionString": "Host={<resourceName>.bindings.tcp.host};Port=
{<resourceName>.bindings.tcp.port};Username=admin;Password=
{<resourceName>.inputs.password};"
```

The connection string placeholder references the `password` input parameter from the `inputs` field:

JSON

```
"inputs": {
  "password": {
    "type": "string",
    "secret": true,
    "default": {
      "generate": {
        "minLength": 10
      }
    }
  }
}
```

The preceding JSON snippet shows the `inputs` field for a resource that has a `connectionString` field. The `password` input parameter is a string type and is marked as a secret. The `default` field is used to specify a default value for the input parameter. In this case, the default value is generated using the `generate` field, with random string of a minimum length.

Built-in resources

The following table is a list of resource types that are explicitly generated by .NET Aspire and extensions developed by the .NET Aspire team:

Cloud-agnostic resource types

These resources are available in the Aspire.Hosting NuGet package.

Expand table

App model usage	Manifest resource type	Heading link
AddContainer	`container.v0`	Container resource type
`PublishAsDockerFile`	`dockerfile.v0`	Dockerfile resource types
AddDatabase	`value.v0`	MongoDB Server resource types
AddMongoDB	`container.v0`	MongoDB resource types
AddDatabase	`value.v0`	MySQL Server resource types
AddMySql	`container.v0`	MySQL resource types
AddDatabase	`value.v0`	Postgres resource types
AddPostgres	`container.v0`	Postgres resource types
AddProject	`project.v0`	Project resource type
AddRabbitMQ	`container.v0`	RabbitMQ resource types
AddRedis	`container.v0`	Redis resource type
AddDatabase	`value.v0`	SQL Server resource types
AddSqlServer	`container.v0`	SQL Server resource types

Project resource type

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);
var apiservice = builder.AddProject<Projects.AspireApp_ApiService>
("apiservice");
```

Example manifest:

JSON

```
"apiservice": {
  "type": "project.v0",
  "path": "../AspireApp.ApiService/AspireApp.ApiService.csproj",
  "env": {
    "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EXCEPTION_LOG_ATTRIBUTES": "true",
    "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EVENT_LOG_ATTRIBUTES": "true"
  },
  "bindings": {
    "http": {
      "scheme": "http",
      "protocol": "tcp",
      "transport": "http"
    },
    "https": {
      "scheme": "https",
      "protocol": "tcp",
      "transport": "http"
    }
  }
}
```

Container resource type

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddContainer("mycontainer", "myimage")
       .WithEnvironment("LOG_LEVEL", "WARN")
       .WithHttpEndpoint(3000);
```

Example manifest:

JSON

```
{
  "resources": {
    "mycontainer": {
      "type": "container.v0",
      "image": "myimage:latest",
      "env": {
        "LOG_LEVEL": "WARN"
      },
      "bindings": {
        "http": {
          "scheme": "http",
```

```
          "protocol": "tcp",
          "transport": "http",
          "containerPort": 3000
        }
      }
    }
  }
}
```

Dockerfile resource types

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddNodeApp("nodeapp", "../nodeapp/app.js")
       .WithHttpEndpoint(hostPort: 5031, env: "PORT")
       .PublishAsDockerFile();
```

> **Tip**
>
> The `PublishAsDockerFile` call is required to generate the Dockerfile resource type in the manifest, and this extension method is only available on the ExecutableResource type.

Example manifest:

JSON

```
{
  "resources": {
    "nodeapp": {
      "type": "dockerfile.v0",
      "path": "../nodeapp/Dockerfile",
      "context": "../nodeapp",
      "env": {
        "NODE_ENV": "development",
        "PORT": "{nodeapp.bindings.http.port}"
      },
      "bindings": {
        "http": {
          "scheme": "http",
          "protocol": "tcp",
          "transport": "http",
          "containerPort": 5031
```

```
        }
      }
    }
  }
}
```

Postgres resource types

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddPostgres("postgres1")
       .AddDatabase("shipping");
```

Example manifest:

JSON

```
{
  "resources": {
    "postgres1": {
      "type": "container.v0",
      "connectionString": "Host={postgres1.bindings.tcp.host};Port=
{postgres1.bindings.tcp.port};Username=postgres;Password=
{postgres1.inputs.password}",
      "image": "postgres:16.2",
      "env": {
        "POSTGRES_HOST_AUTH_METHOD": "scram-sha-256",
        "POSTGRES_INITDB_ARGS": "--auth-host=scram-sha-256 --auth-
local=scram-sha-256",
        "POSTGRES_PASSWORD": "{postgres1.inputs.password}"
      },
      "bindings": {
        "tcp": {
          "scheme": "tcp",
          "protocol": "tcp",
          "transport": "tcp",
          "containerPort": 5432
        }
      },
      "inputs": {
        "password": {
          "type": "string",
          "secret": true,
          "default": {
            "generate": {
              "minLength": 10
```

```
            }
          }
        }
      }
    },
    "shipping": {
      "type": "value.v0",
      "connectionString": "{postgres1.connectionString};Database=shipping"
    }
  }
}
```

RabbitMQ resource types

RabbitMQ is modeled as a container resource `container.v0`. The following sample shows how they're added to the app model.

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddRabbitMQ("rabbitmq1");
```

The previous code produces the following manifest:

JSON

```
{
  "resources": {
    "rabbitmq1": {
      "type": "container.v0",
      "connectionString": "amqp://guest:
{rabbitmq1.inputs.password}@{rabbitmq1.bindings.tcp.host}:
{rabbitmq1.bindings.tcp.port}",
      "image": "rabbitmq:3",
      "env": {
        "RABBITMQ_DEFAULT_USER": "guest",
        "RABBITMQ_DEFAULT_PASS": "{rabbitmq1.inputs.password}"
      },
      "bindings": {
        "tcp": {
          "scheme": "tcp",
          "protocol": "tcp",
          "transport": "tcp",
          "containerPort": 5672
        }
      },
      "inputs": {
        "password": {
          "type": "string",
```

```
          "secret": true,
          "default": {
            "generate": {
              "minLength": 10
            }
          }
        }
      }
    }
  }
}
```

Redis resource type

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddRedis("redis1");
```

Example manifest:

JSON

```
{
  "resources": {
    "redis1": {
      "type": "container.v0",
      "connectionString": "{redis1.bindings.tcp.host}:
{redis1.bindings.tcp.port}",
      "image": "redis:7.2.4",
      "bindings": {
        "tcp": {
          "scheme": "tcp",
          "protocol": "tcp",
          "transport": "tcp",
          "containerPort": 6379
        }
      }
    }
  }
}
```

SQL Server resource types

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddSqlServer("sql1")
       .AddDatabase("shipping");
```

Example manifest:

JSON

```
{
  "resources": {
    "sql1": {
      "type": "container.v0",
      "connectionString": "Server={sql1.bindings.tcp.host},
{sql1.bindings.tcp.port};User ID=sa;Password=
{sql1.inputs.password};TrustServerCertificate=true",
      "image": "mcr.microsoft.com/mssql/server:2022-latest",
      "env": {
        "ACCEPT_EULA": "Y",
        "MSSQL_SA_PASSWORD": "{sql1.inputs.password}"
      },
      "bindings": {
        "tcp": {
          "scheme": "tcp",
          "protocol": "tcp",
          "transport": "tcp",
          "containerPort": 1433
        }
      },
      "inputs": {
        "password": {
          "type": "string",
          "secret": true,
          "default": {
            "generate": {
              "minLength": 10
            }
          }
        }
      }
    },
    "shipping": {
      "type": "value.v0",
      "connectionString": "{sql1.connectionString};Database=shipping"
    }
  }
}
```

MongoDB resource types

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddMongoDB("mongodb1")
       .AddDatabase("shipping");
```

Example manifest:

JSON

```
{
  "resources": {
    "mongodb1": {
      "type": "container.v0",
      "connectionString": "mongodb://{mongodb1.bindings.tcp.host}:
{mongodb1.bindings.tcp.port}",
      "image": "mongo:7.0.5",
      "bindings": {
        "tcp": {
          "scheme": "tcp",
          "protocol": "tcp",
          "transport": "tcp",
          "containerPort": 27017
        }
      }
    },
    "shipping": {
      "type": "value.v0",
      "connectionString": "{mongodb1.connectionString}/shipping"
    }
  }
}
```

MySQL resource types

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddMySql("mysql1")
       .AddDatabase("shipping");
```

Example manifest:

JSON

```
{
  "resources": {
    "mysql1": {
      "type": "container.v0",
      "connectionString": "Server={mysql1.bindings.tcp.host};Port=
{mysql1.bindings.tcp.port};User ID=root;Password={mysql1.inputs.password}",
      "image": "mysql:8.3.0",
      "env": {
        "MYSQL_ROOT_PASSWORD": "{mysql1.inputs.password}"
      },
      "bindings": {
        "tcp": {
          "scheme": "tcp",
          "protocol": "tcp",
          "transport": "tcp",
          "containerPort": 3306
        }
      },
      "inputs": {
        "password": {
          "type": "string",
          "secret": true,
          "default": {
            "generate": {
              "minLength": 10
            }
          }
        }
      }
    },
    "shipping": {
      "type": "value.v0",
      "connectionString": "{mysql1.connectionString};Database=shipping"
    }
  }
}
```

Azure-specific resource types

The following resources are available in the Aspire.Hosting.Azure NuGet package.

Expand table

App Model usage	Manifest resource type	Heading link
AddAzureAppConfiguration	`azure.bicep.v0`	Azure App Configuration resource types
AddAzureKeyVault	`azure.bicep.v0`	Azure Key Vault resource type
`AddAzureRedis`	`azure.bicep.v0`	Azure Redis resource types
AddAzureServiceBus	`azure.bicep.v0`	Azure Service Bus resource type
`AddAzureSqlServer(...)`	`azure.bicep.v0`	Azure SQL resource types
`AddAzureSqlServer(...).AddDatabase(...)`	`value.v0`	Azure SQL resource types
`AddAzurePostgresFlexibleServer(...)`	`azure.bicep.v0`	Azure Postgres resource types
`AddAzurePostgresFlexibleServer(...).AddDatabase(...)`	`value.v0`	Azure Postgres resource types
AddAzureStorage	`azure.storage.v0`	Azure Storage resource types
AddBlobs	`value.v0`	Azure Storage resource types
AddQueues	`value.v0`	Azure Storage resource types
AddTables	`value.v0`	Azure Storage resource types

Azure Key Vault resource type

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddAzureKeyVault("keyvault1");
```

Example manifest:

JSON

```
{
  "resources": {
    "keyvault1": {
      "type": "azure.bicep.v0",
      "connectionString": "{keyvault1.outputs.vaultUri}",
      "path": "aspire.hosting.azure.bicep.keyvault.bicep",
      "params": {
        "principalId": "",
        "principalType": "",
        "vaultName": "keyvault1"
      }
    }
  }
}
```

Azure Service Bus resource type

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddAzureServiceBus("sb1")
       .AddTopic("topic1", [])
       .AddTopic("topic2", [])
       .AddQueue("queue1")
       .AddQueue("queue2");
```

Example manifest:

JSON

```
{
  "resources": {
    "sb1": {
      "type": "azure.bicep.v0",
      "connectionString": "{sb1.outputs.serviceBusEndpoint}",
      "path": "aspire.hosting.azure.bicep.servicebus.bicep",
      "params": {
        "serviceBusNamespaceName": "sb1",
        "principalId": "",
        "principalType": "",
        "queues": [
          "queue1",
          "queue2"
```

```
        ],
        "topics": [
          {
            "name": "topic1",
            "subscriptions": []
          },
          {
            "name": "topic2",
            "subscriptions": []
          }
        ]
      }
    }
  }
}
```

Azure Storage resource types

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("images");

storage.AddBlobs("blobs");
storage.AddQueues("queues");
storage.AddTables("tables");
```

Example manifest:

JSON

```
{
  "resources": {
    "images": {
      "type": "azure.bicep.v0",
      "path": "aspire.hosting.azure.bicep.storage.bicep",
      "params": {
        "principalId": "",
        "principalType": "",
        "storageName": "images"
      }
    },
    "blobs": {
      "type": "value.v0",
      "connectionString": "{images.outputs.blobEndpoint}"
    },
    "queues": {
```

```
      "type": "value.v0",
      "connectionString": "{images.outputs.queueEndpoint}"
    },
    "tables": {
      "type": "value.v0",
      "connectionString": "{images.outputs.tableEndpoint}"
    }
  }
}
```

Azure Redis resource type

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddAzureRedis("azredis1");
```

Example manifest:

JSON

```
{
  "resources": {
    "azredis": {
      "type": "azure.bicep.v0",
      "connectionString": "{azredis.outputs.connectionString}",
      "path": "azredis.module.bicep",
      "params": {
        "principalId": "",
        "principalName": ""
      }
    }
  }
}
```

Azure App Configuration resource type

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddAzureAppConfiguration("appconfig1");
```

Example manifest:

JSON

```
{
  "resources": {
    "appconfig1": {
      "type": "azure.bicep.v0",
      "connectionString": "{appconfig1.outputs.appConfigEndpoint}",
      "path": "aspire.hosting.azure.bicep.appconfig.bicep",
      "params": {
        "configName": "appconfig1",
        "principalId": "",
        "principalType": ""
      }
    }
  }
}
```

Azure SQL resource types

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddAzureSqlServer("sql")
       .AddDatabase("inventory");
```

Example manifest:

JSON

```
{
  "resources": {
    "sql": {
      "type": "azure.bicep.v0",
      "connectionString": "Server=tcp:
{sql.outputs.sqlServerFqdn},1433;Encrypt=True;Authentication=\u0022Active
Directory Default\u0022",
      "path": "sql.module.bicep",
      "params": {
        "principalId": "",
        "principalName": ""
      }
    },
    "inventory": {
      "type": "value.v0",
      "connectionString": "{sql.connectionString};Database=inventory"
```

```
    }
  }
}
```

Azure Postgres resource types

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddAzurePostgresFlexibleServer("postgres")
       .AddDatabase("db");
```

Example manifest:

JSON

```
{
  "resources": {
    "postgres": {
      "type": "azure.bicep.v0",
      "connectionString": "{postgres.outputs.connectionString}",
      "path": "postgres.module.bicep",
      "params": {
        "principalId": "",
        "principalType": "",
        "principalName": ""
      }
    },
    "db": {
      "type": "value.v0",
      "connectionString": "{postgres.connectionString};Database=db"
    }
  }
}
```

Resource types supported in the Azure Developer CLI

The Azure Developer CLI (azd) is a tool that can be used to deploy .NET Aspire projects to Azure Container Apps. With the `azure.bicep.v0` resource type, cloud-agnostic resource container types can be mapped to Azure-specific resources. The following table lists the resource types that are supported in the Azure Developer CLI:

Expand table

Name	Cloud-agnostic API	Azure API
Redis	AddRedis	`AddAzureRedis`
Postgres	AddPostgres	`AddAzurePostgresFlexibleServer`
SQL Server	AddSqlServer	`AddAzureSqlServer`

When resources as configured as Azure resources, the `azure.bicep.v0` resource type is generated in the manifest. For more information, see Deploy a .NET Aspire project to Azure Container Apps using the Azure Developer CLI (in-depth guide).

See also

- .NET Aspire overview
- .NET Aspire orchestration overview
- .NET Aspire integrations overview
- Service discovery in .NET Aspire

Allow unsecure transport in .NET Aspire

Article • 06/15/2024

Starting with .NET Aspire preview 5, the app host will crash if an `applicationUrl` is configured with an unsecure transport (non-TLS `http`) protocol. This is a security feature to prevent accidental exposure of sensitive data. However, there are scenarios where you might need to allow unsecure transport. This article explains how to allow unsecure transport in .NET Aspire projects.

Symptoms

When you run a .NET Aspire project with an `applicationUrl` configured with an unsecure transport protocol, you might see the following error message:

Output

```
The 'applicationUrl' setting must be an https address unless the
'ASPIRE_ALLOW_UNSECURED_TRANSPORT' environment variable is set to true.

This configuration is commonly set in the launch profile.
```

How to allow unsecure transport

To allow an unsecure transport in .NET Aspire, set the `ASPIRE_ALLOW_UNSECURED_TRANSPORT` environment variable to `true`. This environment variable is used to control the behavior of the app host when an `applicationUrl` is configured with an insecure transport protocol:

Windows

PowerShell

```
$env:ASPIRE_ALLOW_UNSECURED_TRANSPORT = "true"
```

Alternatively, you can control this via the launch profile as it exposes the ability to configure environment variables per profile. To do this, consider the following example settings in the `launchSettings.json` file:

JSON

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:15015;http://localhost:15016",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "DOTNET_ENVIRONMENT": "Development",
        "DOTNET_DASHBOARD_OTLP_ENDPOINT_URL": "https://localhost:16099",
        "DOTNET_RESOURCE_SERVICE_ENDPOINT_URL": "https://localhost:17037"
      }
    },
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:15016",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "DOTNET_ENVIRONMENT": "Development",
        "DOTNET_DASHBOARD_OTLP_ENDPOINT_URL": "http://localhost:16099",
        "DOTNET_RESOURCE_SERVICE_ENDPOINT_URL": "http://localhost:17038",
        "ASPIRE_ALLOW_UNSECURED_TRANSPORT": "true"
      }
    }
  }
}
```

The preceding example shows two profiles, `https` and `http`. The `https` profile is configured with a secure transport protocol, while the `http` profile is configured with an insecure transport protocol. The `ASPIRE_ALLOW_UNSECURED_TRANSPORT` environment variable is set to `true` in the `http` profile to allow unsecure transport.

Troubleshoot untrusted localhost certificate in .NET Aspire

Article • 10/16/2024

This article provides guidance on how to troubleshoot issues that you might encounter when working with untrusted localhost certificates in .NET Aspire.

Symptoms

Several .NET Aspire templates include ASP.NET Core projects that are configured to use HTTPS by default. If this is the first time you're running the project, and you're using Visual Studio, you're prompted to install a localhost certificate.

- There are situations in which you trust/install the development certificate, but you don't close all your browser windows. In these cases, your browser might indicate that the certificate isn't trusted.
- There are also situations where you don't trust the certificate at all. In these cases, your browser might indicate that the certificate isn't trusted.

Additionally, there are warning messages from Kestrel written to the console that indicate that the certificate is not trusted.

Possible solutions

1. **Close all browser windows** and *try again*.
2. If you're still experiencing the issue, then attempt to resolve this by trusting the self-signed development certificate with the .NET CLI. To trust the certificate, run the following commands. First, remove the existing certificates.

   > **Note**
   >
   > This will remove all existing development certificates on the local machine.

   .NET CLI

   ```
   dotnet dev-certs https --clean
   ```

To trust the certificate:

.NET CLI

```
dotnet dev-certs https --trust
```

For more troubleshooting, see Troubleshoot certificate problems such as certificate not trusted.

See also

- Trust the ASP.NET Core HTTPS development certificate on Windows and macOS
- Trust HTTPS certificate on Linux
- .NET CLI: dotnet dev-certs
- Trust localhost certificate on Linux

Troubleshoot installing the .NET Aspire workload

Article • 05/21/2024

This article provides guidance on how to troubleshoot issues that you might encounter when installing the .NET Aspire workload from the .NET CLI.

Symptoms

When you install the .NET Aspire workload, you might encounter an installation error. The error message might indicate that the installation failed, or that the workload couldn't be installed. The error message might also indicate that a package source is unavailable, or that a package source isn't found often similar to:

Output

```
Workload update failed: One or more errors ocurred: (Version X.Y.00Z of
package A.B.C is not found in NuGet feeds.
```

One common issue is that your SDK is aware of some workload manifest or workload pack versions that are not present in any of the feeds configured when you are trying to run the dotnet workload commands. This can happen if the SDK, during its daily check for updates, finds a new version of a workload manifest in a feed that isn't used when running `dotnet workload` commands. This discrepancy can cause errors during installation.

A less common issue, even when using the correct feeds, is that a workload manifest may have a dependency on a workload pack that is not published on the same feed. This can also lead to errors during installation as the required pack cannot be found.

Possible solution

Ensure that any recursive *Nuget.config* files are configured to specify the correct package sources and NuGet feeds. For example, if you have a *Nuget.config* file in your user profile directory, ensure that it doesn't specify a package source that is no longer available.

If you encounter errors related to the SDK being aware of workload manifest or workload pack versions not present in your configured feeds, you may need to adjust

your feeds or find the feed where the new version of the manifest or required pack is located.

In the case where a workload manifest has a dependency on a workload pack not published on the same feed, you will need to find and add the feed where that pack is located to your NuGet configuration.

> **ⓘ Important**
>
> Some development environments may depend on private feeds that provide newer versions of the workload manifest or workload pack. In these situations, you may want to disable the daily SDK check for updates to avoid encountering errors during installation.
>
> To disable the daily SDK check for updates, set the `DOTNET_CLI_WORKLOAD_UPDATE_NOTIFY_DISABLE` environment variable to `true`.

See also

- .NET SDK: Diagnosing issues with .NET SDK Workloads
- .NET CLI: dotnet workload install
- NuGet: nuget.config reference

The specified name is already in use

Article • 06/15/2024

When deploying to Azure initial deployments may fail with an error similar to the following:

> "The specified name is already in use"

This article describes several techniques to avoid this problem.

Symptoms

When deploying a .NET Aspire project to Azure, the resources in the app model are transformed into Azure resources. Some Azure resources have globally scoped names, such as Azure App Configuration, where all instances are in the `[name].azconfig.io` global namespace.

The value of `[name]` is derived from the .NET Aspire resource name, along with random characters based on the resource group name. However, the generated string may exceed the allowable length for the resource name in App Configuration. As a result, some characters are truncated to ensure compliance.

When a conflict occurs in the global namespace, the resource fails to deploy because the combination of `[name]+[truncated hash]` isn't unique enough.

Possible solutions

One workaround is to avoid using common names like `appconfig` or `storage` for resources. Instead, choose a more meaningful and specific name. This reduces the potential for conflict, but does not completely eliminate it. In such cases, you can use callback methods to set a specific name and avoid using the computed string altogether.

Consider the following example:

C#

```
var appConfig = builder.AddAzureAppConfiguration(
    "appConfig",
    (resource, construct, store) =>
{
```

```
    store.AssignProperty(p => p.Name, "'noncalculatedname'");
});
```

Container runtime appears to be unhealthy

Article • 07/08/2024

.NET Aspire requires Docker (or Podman) to be running and healthy. This topic describes a possible symptom you may see if Docker isn't in a healthy state.

Symptoms

When starting the AppHost the dashboard doesn't show up and an exception stack trace similar to this example is displayed in the console:

Output

```
info: Aspire.Hosting.DistributedApplication[0]
      Aspire version: 8.1.0-dev
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application starting.
info: Aspire.Hosting.DistributedApplication[0]
      Application host directory is:
D:\aspire\playground\PostgresEndToEnd\PostgresEndToEnd.AppHost
fail: Microsoft.Extensions.Hosting.Internal.Host[11]
      Hosting failed to start
      Aspire.Hosting.DistributedApplicationException: Container runtime
'docker' was found but appears to be unhealthy. The error from the container
runtime check was error during connect: this error may indicate that the
docker daemon is not running: Get
"http://%2F%2F.%2Fpipe%2Fdocker_engine/v1.45/containers/json?limit=1": open
//./pipe/docker_engine: The system cannot find the file specified..
```

Possible solutions

Confirm that Docker is installed and running:

- On Windows, check that in the system tray the Docker icon is present and marked as "Running".
- On Linux, check that `docker ps -a` returns success.

Connection string is missing

Article • 08/29/2024

In .NET Aspire, code identifies resources with an arbitrary string, such as "database". Code that is consuming the resource elsewhere must use the same string or it will fail to correctly configure their relationships.

Symptoms

When your app accesses a service that needs one of the integrations in your app, it may fail with an exception similar to the following:

> "InvalidOperationException: ConnectionString is missing."

Possible solutions

Verify that the name of the resource, for instance a database resource, is the same in the AppHost and the Service that fails.

For example, if the AppHost defines a PostgreSQL resource with the name `db1` like this:

C#

```
var db1 = builder.AddPostgres("pg1").AddDatabase("db1");
```

Then the service needs to resolve the resource with the same name `db1`.

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.AddNpgsqlDbContext<MyDb1Context>("db1");
```

Any other value than the one provided in the AppHost will result in the exception message described above.

Breaking changes in .NET Aspire

Article • 11/12/2024

Use this reference section to find breaking changes that might apply to you if you're upgrading your app to a newer version of .NET Aspire. You can navigate the table of contents either by .NET Aspire version or by technology area.

If you're looking for breaking changes for .NET, see Breaking changes in .NET.

GitHub issues and announcements

You can also view individual issues that detail the breaking changes introduced in .NET in the following GitHub repositories:

- For .NET Aspire issues tracking breaking changes, see dotnet/aspire.
- For .NET Aspire pull requests introducing breaking changes, see dotnet/aspire.

See also

- API removal in .NET
- .NET runtime compatibility

.NET Aspire diagnostics overview

Article • 02/25/2025

Several APIs of .NET Aspire are decorated with the ExperimentalAttribute. This attribute indicates that the API is experimental and may be removed or changed in future versions of .NET Aspire. The attribute is used to identify APIs that aren't yet stable and may not be suitable for production use.

AZPROVISION001

.NET Aspire provides various overloads for Azure Provisioning resource types (from the `Azure.Provisioning` package). The overloads are used to create resources with different configurations. The overloads are experimental and may be removed or changed in future versions of .NET Aspire.

To suppress this diagnostic with the `SuppressMessageAttribute`, add the following code to your project:

C#

```
using System.Diagnostics.CodeAnalysis;

[assembly: SuppressMessage("AZPROVISION001", "Justification")]
```

Alternatively, you can suppress this diagnostic with preprocessor directive by adding the following code to your project:

C#

```
#pragma warning disable AZPROVISION001
        // API that is causing the warning.
#pragma warning restore AZPROVISION001
```

ASPIREACADOMAINS001

.NET Aspire 9.0 introduces the ability to customize container app resources using the `PublishAsAzureContainerApp(...)` extension method. When using this method the Azure Developer CLI (`azd`) can no longer preserve custom domains. Instead use the `ConfigureCustomDomain` method to configure a custom domain within the .NET Aspire

app host. The `ConfigureCustomDomain(...)` extension method is experimental. To suppress the compiler error/warning use the following code:

To suppress this diagnostic with the `SuppressMessageAttribute`, add the following code to your project:

C#

```
using System.Diagnostics.CodeAnalysis;

[assembly: SuppressMessage("ASPIREACADOMAINS001", "Justification")]
```

Alternatively, you can suppress this diagnostic with preprocessor directive by adding the following code to your project:

C#

```
#pragma warning disable ASPIREACADOMAINS001
        // API that is causing the warning.
#pragma warning restore ASPIREACADOMAINS001
```

ASPIREHOSTINGPYTHON001

.NET Aspire provides a way to add Python executables or applications to the .NET Aspire app host. Since the shape of this API is expected to change in the future, it has been marked as *Experimental*. To suppress the compiler error/warning use the following code:

To suppress this diagnostic with the `SuppressMessageAttribute`, add the following code to your project file:

XML

```
<PropertyGroup>
  <NoWarn>$(NoWarn);ASPIREHOSTINGPYTHON001</NoWarn>
<PropertyGroup>
```

Alternatively, you can suppress this diagnostic with preprocessor directive by adding the following code to your project:

C#

```
#pragma warning disable ASPIREHOSTINGPYTHON001
        // API that is causing the warning.
#pragma warning restore ASPIREHOSTINGPYTHON001
```

ASPIRECOSMOSDB001

.NET Aspire provides a way to use the CosmosDB Linux-based (preview) emulator. Since this emulator is in preview and the shape of this API is expected to change in the future, it has been marked as *Experimental*. To suppress the compiler error/warning use the following code:

To suppress this diagnostic with the `SuppressMessageAttribute`, add the following code to your project file:

XML

```xml
<PropertyGroup>
  <NoWarn>$(NoWarn);ASPIRECOSMOSDB001</NoWarn>
<PropertyGroup>
```

Alternatively, you can suppress this diagnostic with preprocessor directive by adding the following code to your project:

C#

```csharp
#pragma warning disable ASPIRECOSMOSDB001
        // API that is causing the warning.
#pragma warning restore ASPIRECOSMOSDB001
```

Frequently asked questions about .NET Aspire

FAQ

This article lists frequently asked questions about .NET Aspire. For a more comprehensive overview, see .NET Aspire overview.

Why choose .NET Aspire over Docker Compose for orchestration?

Docker Compose is excellent but is unproductive when all you want to do is run several projects or executables. Docker Compose requires developers to build container images and to run apps inside of containers. That's a barrier when you just want to run your front end, back end, workers, and a database. With .NET Aspire, you don't need to learn anything beyond what you already know.

Configuration through declarative code is better than through YAML. Docker Compose gets complex once you attempt to do any form of abstraction or composition (for example, see the old eshopOnContainers app). In addition, there are environment variable replacements (and includes) and no types or IntelliSense, and it's hard to reason about what exactly is running. Debugging is also difficult. .NET Aspire produces a better experience that's easy to get started and scales up to an orchestrator like Compose using a real programming language.

How to add projects to .NET Aspire?

You can manually add projects to your .NET Aspire solution by using the `builder.AddProject("<name>", "<path/to/project.csproj>")` API.

How to deploy .NET Aspire without target cloud-provider tooling?

.NET Aspire doesn't constrain deployment of any existing project or solution. .NET Aspire exposes a deployment manifest that's used by tool authors to produce artifacts for deployment to any cloud provider. However, unfortunately, not all cloud providers offer tooling for deployments based on this manifest. The manifest is a simple JSON file that describes the resources of your app and the dependencies between them. The

manifest is used by the Azure Developer CLI to deploy to Azure. Likewise, Aspir8 uses the manifest to deploy to Kubernetes. You can use the manifest to deploy to any cloud provider that supports the resources you're using.

If .NET Aspire is mainly for local development, why are client integrations in my project?

.NET Aspire's client integrations simplify your development process. They're lightweight wrappers around popular libraries like StackExchange.Redis and Npgsql, adding features such as telemetry, health checks, and resilience.

Even if you use .NET Aspire only for local development, these integrations provide reasonable defaults, seamless dependency injection, and consistent APIs.

You're not locked into .NET Aspire's ecosystem. These integrations are just libraries, and you can configure them as you would with the underlying libraries, using environment variables or your preferred methods.

Can .NET Aspire apps be built without Azure dependencies and deployed elsewhere?

Yes, you can build .NET Aspire apps without using any Azure-proprietary dependencies. While .NET Aspire does offer a first-party solution to deploying to Azure, it's not a requirement. .NET Aspire is a cloud-native stack that can be used to build applications that run anywhere. All Azure-specific offerings are explicitly called out as such.

Why use .NET Aspire service discovery over Docker Compose with Kubernetes?

.NET Aspire service discovery APIs are an abstraction that works with various providers (like Kubernetes and Consul). One of the big advantages is that it works locally and is backed by .NET's `IConfiguration` abstraction. This means you can implement service discovery across your compute fabric in a way that doesn't result in code changes. If you have multiple Kubernetes clusters or services on Azure App Service or Azure Functions,

you don't have to fundamentally change your application code to make it work locally, either in a single cluster or across multiple clusters. That's the benefit of the abstraction.

Why use .NET Aspire if OpenTelemetry is available in .NET?

.NET Aspire takes a big bet on .NET's integration with OpenTelemetry. The .NET Aspire dashboard is a standard OTLP server that visualizes various telemetry data. Leaning on these open standards makes it easy to build these things without breaking compatibility with the broader ecosystem.

Why use .NET Aspire if Grafana, Jaeger, and Prometheus work with .NET?

.NET Aspire isn't a replacement for these tools, but rather a complementary technology. .NET Aspire is a set of libraries and tools that make it easy to build applications that are observable. For more information, see the Metrics example in the .NET Aspire sample repository that shows Grafana and Prometheus.

Why create another framework when existing ones work well?

.NET Aspire isn't a framework, it's an opinionated stack. Perhaps the most controversial parts of it are the `DistributedApplication` APIs that you can use to build up the orchestration model in any .NET-based language. While everything is possible today, it's not easy. Using the Unix philosophy, the entire cloud-native ecosystem is built around tying various pieces of CNCF software together to build a stack. .NET Aspire tries to do the same thing using learnings from the cloud-native space and picks some opinions (in ways that use the same building blocks). One novel thing about how .NET Aspire builds various pieces of the stack is that it doesn't restrict the access or compatibility of other applications, frameworks, or services. As people play with it more, they realize how composable and extensible it is.

How does .NET Aspire differ from Microsoft Orleans?

Microsoft Orleans and .NET Aspire are complementary technologies.

Orleans is a distributed actor-based framework. .NET Aspire is a cloud-ready stack for building observable, production-ready, distributed applications. It includes local orchestration capabilities to simplify the developer inner loop and reusable opinionated components for integrating with commonly used application dependencies. An Orleans-based solution will still have external dependencies such as data stores and caches for which .NET Aspire can be used for orchestration purposes.

For more information, see Use Orleans with .NET Aspire and the corresponding Orleans voting app sample.

How does .NET Aspire differ from Dapr?

Dapr and .NET Aspire are complementary technologies.

Where Dapr abstracts some of the underlying cloud platform, .NET Aspire provides opinionated configuration around the underlying cloud technologies without abstracting them. A .NET-based application that uses Dapr can use .NET Aspire to orchestrate the local developer inner loop and streamline deployment. .NET Aspire includes extensions that support the launching of Dapr side-car processes during the inner loop.

For more information, see Use Dapr with .NET Aspire and the corresponding Dapr sample app in the .NET Aspire sample repository.

How does .NET Aspire differ from Project Tye?

Project Tye was an experiment which explored the launching and orchestration of micro-services and support deployment into orchestrators such as Kubernetes. .NET Aspire is a superset of Tye which includes the orchestration and deployment capabilities along with opinionated components for integrating common cloud-native dependencies. .NET Aspire can be considered the evolution of the Project Tye experiment.

How are .NET Aspire and Azure SDK for .NET related?

.NET Aspire provides components that rely on the Azure SDK for .NET, to expose common functionality for storage (Azure Blob Storage, Azure Storage Queues, and

Azure Table Storage), databases (Azure Cosmos DB and Azure Cosmos DB with Entity Framework Core), messaging, and security.

How are .NET Aspire and Kubernetes related?

.NET Aspire makes it easy to develop distributed applications that can be orchestrated on your local development environment as executables and containers. Kubernetes is a technology that orchestrates and manages containers across multiple machines. .NET Aspire projects can produce a manifest that tool authors can use to produce artifacts for deployment to Kubernetes. In essence, Kubernetes is a deployment target for .NET Aspire projects.

Are worker services supported in .NET Aspire?

Yes, worker services are fully supported and there are docs and samples available to help you get started. Worker services are a great way to run background tasks, scheduled tasks, or long-running tasks in .NET Aspire. For more information, see Database migrations with Entity Framework Core sample app.

Are Azure Functions supported in .NET Aspire?

Yes, .NET Aspire has preview support for integrating Azure Functions into your app.

Does .NET Aspire support running web apps locally on IIS or IIS Express?

No. .NET Aspire doesn't support running web apps on IIS or IIS Express.

Does .NET Aspire support deploying apps to IIS?

No. .NET Aspire doesn't support deploying apps to IIS. However, it doesn't prevent you from deploying your apps to IIS in the same way that you always have.

How to fix integrations and Service Discovery issues when deploying .NET Aspire apps to IIS?

.NET Aspire integrations require specific configuration that must be provided manually. The same is true for Service Discovery, ideally, you should deploy to something other than IIS.

What is the purpose of the Community Toolkit project?

The goal of the project is to be a centralized home for extensions and integrations for .NET Aspire, helping to provide consistency in the way that integrations are built and maintained, as well as easier discoverability for users.

How is the Community Toolkit project different from the official .NET Aspire project?

The .NET Aspire Community Toolkit is a community-driven project that's maintained by the community and isn't officially supported by the .NET Aspire team. The toolkit is a collection of integrations and extensions that are built on top of the .NET Aspire project.

How can I contribute to the Community Toolkit project?

Anyone can contribute to the .NET Aspire Community Toolkit and before you get started, be sure to read the Contributing Guide to learn how to contribute to the project.

Should I propose a new integration on the Community Toolkit or the `dotnet/aspire` repo?

If you have an idea for a new integration, you should propose it on the .NET Aspire Community Toolkit repository, rather than dotnet/aspire, as the official .NET Aspire project is focused on the core functionality of the .NET Aspire project.

If you've proposed an integration on the `dotnet/aspire` repository, you can still propose it in the Community Toolkit, but link to the existing issue on the `dotnet/aspire` repository to provide context.

How can I find Community Toolkit integrations?

Integrations from the .NET Aspire Community Toolkit appear in the **Add Aspire Integration** dialog in Visual Studio under the namespace `CommunityToolkit.Aspire.*`.

Next steps

To learn more about networking and functions:

- .NET Aspire overview
- Build your first .NET Aspire project
- .NET Aspire components