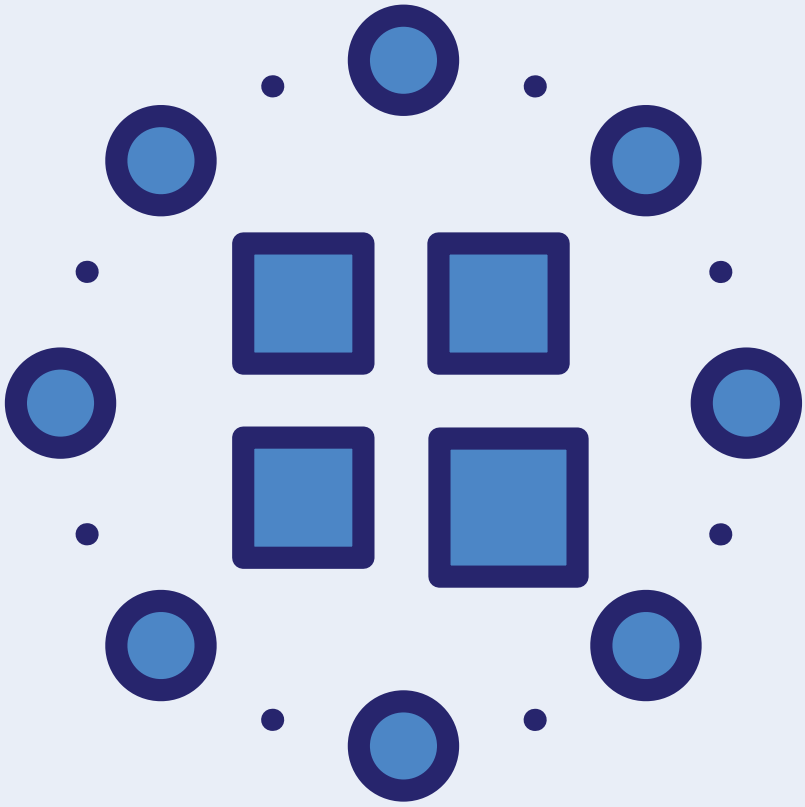
Building AI-Powered Distributed Architectures with Aspire

Build cloud-native microservices with PostgreSQL, Redis, RabbitMQ, Keycloak, Ollama, and GenAI



Mehmet Ozkaya

First Edition

Building AI-Powered Distributed Architectures with Aspire

Build cloud-native microservices with PostgreSQL, Redis, RabbitMQ, Keycloak, Ollama, and GenAI

Mehmet Ozkaya

‹packt›

Building AI-Powered Distributed Architectures with Aspire

Copyright © 2026 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.
The author's original ideas were further refined by a professional publishing team. Cutting-edge AI was also used to shape content into this form. Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.
Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Gebin George
Relationship Lead: Dhruv Kataria
Project Manager: Prajakta Naik
Content Engineer: Mark D'Souza
Technical Editor: Rahul Limbachiya
Indexer: Pratik Shirodkar
Production Designer: Salma Patel
Growth Lead: Dipali Malwatkar

First published: October 2026
Production reference: 1300926

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK

ISBN 978-1-80849-277-8

packtpub.com

Contributors

About the author

Mehmet Ozkaya is a software and solutions architect with more than 15 years of experience in the software development industry, including roles at global companies. His work focuses on AI agents, generative AI, microservices, and cloud-native architectures across .NET, AWS, and Azure. He develops practical agentic AI, generative AI, and microservices projects on GitHub and draws on this hands-on experience to create technical training. Mehmet has taught more than 200,000 students across 20 Udemy courses and regularly shares his development work and insights through GitHub and Medium.

About the reviewer

Avinash Maddineni is a lead data engineer at a Fortune 10 healthcare company, where he builds agentic AI tooling and governed data platforms that support reliable, audit-ready enterprise systems. With more than 14 years of experience across travel, finance, healthcare, and utilities, he has designed large-scale data and AI solutions spanning ETL, data warehousing, LLMs, and agent-based workflows. He is also the founder of PureStrokeAI, an AI-powered tennis sensor startup, and co-founder of StemSenseAI. His work focuses on making enterprise AI practical, trustworthy, and scalable by combining real-world data with strong governance and disciplined engineering practices.

Table of Contents

Preface xiii

How this book is structured 0

Free benefits with your book xxi

Chapter 1: Introducing the AI-Powered eShop Architecture with Aspire 1

Understanding the eShop application 2

Applying cloud-native principles 4

Understanding Aspire's role 4

Following the architecture as it evolves 5

Understanding the solution organization 8

Summary 9

Chapter 2: Cloud-Native Distributed Architectures and Backing Services 11

Defining cloud-native distributed architectures 12

Backing services in cloud-native applications 14

Summary 18

Chapter 3: Developing Distributed Applications with Aspire 21

Introducing Aspire 22

Core concepts of Aspire 24

Orchestration • 25

Integrations • 25

Service discovery • 26

Orchestration in Aspire 26

Integrations in Aspire 31

Hosting integrations for resource provisioning • 31

Client integrations for resource consumption • 31

Hosting integrations – Resource provisioning via AppHost integrations • 32

Client integrations – Extending Host Builder for Resources • 33

Hosting and client integration workflow • 33
Service discovery in Aspire **34**
Summary **38**

Chapter 4: Building Your First Aspire Application 41

Creating an Aspire starter application with Visual Studio **42**
Examining the Aspire project structure and responsibilities **47**
AspireSample.AppHost • 48
AspireSample.ServiceDefaults • 48
AspireSample.ApiService • 49
AspireSample.Web • 49
Understanding how the Aspire projects work together • 49
Inspecting AppHost configuration and orchestration **50**
Implementing shared configuration with the Service Defaults project **54**
Managing inter-service communication and service discovery **58**
Adding Aspire to existing .NET applications **62**
Summary **70**

Chapter 5: Building an eShop Distributed Microservices Architecture with Aspire 71

Planning and setting up the eShop distributed architecture **72**
Setting up the eShop solution structure **75**
Setting up the AppHost and ServiceDefaults projects **76**
Planning the eShop distributed microservices development **78**
Summary **79**

Chapter 6: Developing the Catalog Microservice with PostgreSQL and Aspire 81

Domain analysis of the Catalog microservice **82**
Technical analysis of the Catalog microservice **84**
N-layer architecture pattern • 84
Fundamental patterns and principles • 85
Project folder structure • 86
Setting up the Catalog microservice with Aspire orchestration **88**
Creating the Catalog Web API project • 88

Registering the Catalog project in AppHost • 91
Creating the Product domain model • 92
Configuring PostgreSQL as the Catalog backing service 93
Adding PostgreSQL to the AppHost 96
Connecting the Catalog microservice to PostgreSQL • 98
Running and inspecting the backing service • 99
Configuring the PostgreSQL client integration 101
Creating the Catalog data layer 102
Creating the Entity Framework Core migration 105
Applying migrations and seeding data • 107
Verifying migrations, seeding, and persistence • 111
Implementing the Product service layer 115
Creating the Catalog service layer • 115
Registering ProductService with dependency injection • 118
Creating the Product endpoints 119
Retrieving all products • 120
Retrieving a product by ID • 121
Creating a product • 121
Updating a product • 122
Deleting a product • 122
Registering the Product endpoints 123
Testing the endpoints 127
Preparing Catalog.http and starting the application • 127
Testing and debugging product retrieval • 129
Testing create, update, and delete operations • 130
Summary 132

Chapter 7: Developing the Basket Microservice with Redis and Aspire 133

Domain analysis of the Basket microservice 134
Domain models • 135
Application use cases • 135
REST API endpoints • 136
Data storage • 136
Technical analysis of the Basket microservice 136
Fundamental patterns and principles • 137

Project folder structure • 137
Creating the Basket microservice project with Aspire ... **139**
Creating the Basket domain models ... **141**
Integrating Redis as a distributed cache with Aspire ... **142**
Connecting the Basket microservice to Redis using StackExchange.Redis • 143
Redis client integration • 146
Developing the Basket service class with caching logic ... **148**
Developing the Basket API endpoints with ASP.NET Core Minimal APIs ... **150**
Registering the Basket endpoints ... **152**
Testing the Basket microservice operations with Redis ... **153**
Summary ... **159**

Chapter 8: Synchronous Communication Between Catalog and Basket Using Aspire Service Discovery **161**

Architecture of synchronous communication between Catalog and Basket microservices ... **162**
Leveraging Aspire Service Discovery for microservice communication ... **163**
Sharing the Product entity between Catalog and Basket microservices ... **165**
Implementing the Catalog API client for HTTP communication ... **167**
Registering the Catalog API client ... **168**
Integrating Catalog into BasketService ... **169**
Testing synchronous communication and end-to-end data flow ... **170**
Summary ... **173**

Chapter 9: Asynchronous Communication with RabbitMQ and MassTransit in Aspire **175**

Architecture of asynchronous communication between microservices ... **176**
Modeling the product-price-change event flow ... **177**
Adding RabbitMQ to the Aspire environment ... **179**
Connecting Catalog and Basket to RabbitMQ ... **180**
Defining shared integration events ... **182**
Configuring MassTransit for Catalog and Basket ... **185**
Publishing product-price-change events from Catalog ... **189**
Consuming product-price-change events in Basket ... **192**
Testing the asynchronous communication flow ... **194**
Summary ... **203**

Chapter 10: Securing the Basket Microservice with Keycloak Authentication 205

Understanding Keycloak identity and access management ... 206
Integrating Keycloak Hosting with Aspire ... 207
Configuring Keycloak for the eShop application ... 212
Creating a Keycloak realm • 212
Registering the OpenID Connect client • 213
Creating a test user • 215
Requesting an access token • 217
Securing the Basket microservice with Keycloak ... 219
Integrating Keycloak with the Basket microservice • 220
Protecting the Basket endpoints • 221
Testing and observing the secured Basket API ... 223
Testing the secured Basket endpoints • 223
Observing authentication in the Aspire dashboard • 226
Summary ... 226

Chapter 11: Developing the Client Blazor Web Application 227

Creating the Blazor Web Application and Aspire Integration ... 228
Integrating the Catalog microservice with the Blazor application ... 232
Registering the Catalog API client with Aspire service discovery ... 234
Developing the Products Razor page ... 236
Testing the Products page ... 239
Enabling output caching with Redis ... 243
Applying and testing output caching on the Products page ... 245
Summary ... 247

Chapter 12: Deploying the eShop Aspire Application to Azure Container Apps 249

Overview of Azure Container Apps ... 250
Managing data volumes and backing services in Azure Container Apps ... 252
Deploying our Aspire application to Azure Container Apps ... 255
Prerequisites and setup verification • 255
Initializing the Aspire application for Azure deployment • 256

Provisioning and deploying the application • 256
Verifying the deployment in the Azure portal ... 258
Testing the deployment ... 259
Cleaning up Azure resources with the azd down command ... 262
Summary ... 263

Chapter 13: Building Generative AI Features with Microsoft.Extensions.AI, Microsoft Agent Framework, and Ollama 265

Fundamentals and capabilities of large language models ... 267
Running LLMs locally with Ollama ... 267
Using AI services as cloud-native backing services ... 268
Ollama integration in Aspire ... 268
Downloading and managing LLMs in Ollama containers ... 271
Consuming Ollama from the Catalog microservice ... 274
Using Microsoft.Extensions.AI for provider-independent AI integration ... 275
Registering the Ollama chat client ... 276
Developing the AI-powered product support service ... 277
Exposing and testing the AI support endpoint ... 280
Integrating AI-powered support into the Blazor application ... 282
Testing the Blazor AI support feature ... 285
Summary ... 288

Chapter 14: Building Semantic Product Search with Vector Embeddings 289

Understanding semantic product search, vector databases, and embeddings ... 290
Comparing traditional and vector databases • 291
Understanding vectors and vector embeddings • 292
Configuring the embedding model with Ollama ... 294
Adding vector-search integrations to the Catalog microservice ... 297
Creating the ProductVector model ... 299
Implementing traditional and semantic product search ... 301
Adding traditional keyword search • 301
Implementing the semantic search • 302
Registering the embedding generator and vector store • 303
Implementing semantic search in ProductAIService • 304
Exposing and testing the search endpoints ... 306

Adding the Catalog search endpoints • 306
Testing the Catalog search endpoints • 308
Integrating search into the Blazor frontend 310
Testing the complete search experience 313
Summary 315

Chapter 15: Unlock Your Exclusive Benefits 317

Unlock this Book's Free Benefits in 3 Easy Steps 318

Other Books You May Enjoy 322

Subscribe to Deep Engineering 324

Index 325

Preface

Modern distributed applications bring together far more than application code. They depend on databases, caches, message brokers, identity providers, service discovery, observability, container platforms, cloud infrastructure, and, increasingly, generative AI services. The challenge is not simply to build each component, but to make the whole system work together in a way that remains understandable, testable, secure, and deployable.

This book takes a practical, project-driven approach to that challenge. You will build one AI-powered eShop application from its architectural foundations through to a working distributed system. Aspire provides the application model and orchestration experience that ties the project together, while the eShop itself grows chapter by chapter with Catalog and Basket microservices, a Blazor WebApp, PostgreSQL, Redis, RabbitMQ, Keycloak, Azure Container Apps, and AI capabilities powered by Ollama and Microsoft AI abstractions.

Following the same application throughout gives you a consistent context for learning. Instead of encountering databases, caching, messaging, security, cloud deployment, and AI as isolated examples, you will see why each capability is introduced, where it belongs in the architecture, and how it interacts with the components that are already in place. By the end of the book, you will have followed the evolution of the eShop from its initial architecture to cloud deployment, AI-powered customer support, and semantic product search.

Who this book is for

This book is for .NET developers, software engineers, and solution architects who want a hands-on path to building cloud-native distributed applications with Aspire. It is particularly useful if you already work with ASP.NET Core and want to understand how to coordinate multiple services, connect them to backing infrastructure, observe them as one system, deploy them to the cloud, and extend them with generative AI capabilities.

You should be comfortable reading and writing C# and have a basic understanding of ASP.NET Core web development and HTTP APIs. Prior experience with Aspire is not required: the early chapters establish the cloud-native concepts, introduce Aspire and its application model, and then build a starter distributed application before the main eShop implementation begins. Familiarity with containers, Redis, RabbitMQ, Keycloak, Azure, or LLM development is useful but not essential, because these technologies are introduced as they become relevant to the project.

How this book is structured

The book is designed to be read sequentially because each chapter extends the same solution. The early chapters establish the architecture and Aspire fundamentals; the middle chapters build and connect the eShop services; and the later chapters add security, a user-facing web application, cloud deployment, and AI features. If you prefer to jump to a particular stage, the section-by-section source repository lets you start from the corresponding project state rather than rebuilding every earlier step.

What this book covers

Chapter 1, *Introducing the AI-Powered eShop Architecture with Aspire*, introduces the eShop application you will build throughout the book. You will see how the eShop services work together, how Aspire supports orchestration and observability, and how the solution evolves throughout the book.

Chapter 2, *Cloud-Native Distributed Architectures and Backing Services*, establishes the implementation principles, covering microservices, containers, orchestration, scalability, CI/CD, observability, and backing services ranging from PostgreSQL, Redis, RabbitMQ, and Keycloak to LLMs and vector databases.

Chapter 3, *Developing Distributed Applications with Aspire*, introduces Aspire as a code-first application platform for distributed systems. You will explore its core concepts—application modeling, AppHost, orchestration, integrations, resources, references, and service discovery—and see how they reduce the manual configuration required to coordinate services and backing resources.

Chapter 4, *Building Your First Aspire Application*, moves from concepts to a working starter solution. You will create and run an Aspire application with a Blazor frontend, an API service, and Redis, inspect resources through the Aspire dashboard, understand AppHost and ServiceDefaults, observe service discovery in practice, and compare Aspire orchestration with an existing containerized setup.

Chapter 5, *Building an eShop Distributed Microservices Architecture with Aspire*, creates the foundation for the book's main project. You will plan the eShop architecture, organize the solution into `aspire`, `services`, and `frontend` folders, set up AppHost and ServiceDefaults, and establish the implementation roadmap for the Catalog and Basket services, communication, security, frontend, Azure deployment, and AI features.

Chapter 6, *Developing the Catalog Microservice with PostgreSQL and Aspire*, builds the Catalog microservice end to end. You will model the Product domain, configure PostgreSQL through Aspire, integrate Entity Framework Core, create migrations and seed data, implement the `Product` service layer, expose CRUD operations through minimal APIs, and test the endpoints while using Aspire logs and traces for troubleshooting.

Chapter 7, *Developing the Basket Microservice with Redis and Aspire*, builds the Basket microservice and its shopping-cart domain. You will configure Redis as a distributed cache, connect the Basket microservice through Aspire and `IDistributedCache`, implement the service and minimal API endpoints, and verify the stored shopping-cart data using Redis tooling.

Chapter 8, *Synchronous Communication Between Catalog and Basket Using Aspire Service Discovery*, connects the Catalog and Basket microservices over HTTP. Using Aspire service discovery, the Basket microservice locates the Catalog microservice by its logical service name, calls it through a typed API client, retrieves authoritative product names and prices, and enriches basket items before storing them in Redis.

Chapter 9, *Asynchronous Communication with RabbitMQ and MassTransit in Aspire*, adds event-driven communication. The Catalog microservice publishes a `ProductPriceChangedIntegrationEvent` through RabbitMQ, MassTransit provides the `publish`/`consume` integration, and the Basket microservice consumes the event to update matching items already stored in Redis without requiring a direct request from the Basket microservice.

Chapter 10, *Securing the Basket Microservice with Keycloak Authentication*, introduces identity and access management with Keycloak. You will configure Keycloak in Aspire, create the required realm, client, and test user, obtain and inspect tokens, configure JWT bearer authentication in the Basket microservice, protect its endpoints, and test authorized and unauthorized requests.

Chapter 11, *Developing the Client Blazor Web Application*, adds the user-facing application. You will create and orchestrate the Blazor WebApp, connect it to the Catalog and Basket microservices through Aspire service discovery, build a `Products` page, and add Redis-backed output caching to reduce unnecessary Catalog calls and improve responsiveness.

Chapter 12, *Deploying the eShop Aspire Application to Azure Container Apps*, takes the distributed application from local orchestration to Azure. You will prepare the Aspire application for Azure Container Apps, use the Azure Developer CLI to initialize, provision, and deploy the environment, handle differences in backing-service persistence, verify the deployed resources and request flows, and clean up the environment with azd down.

Chapter 13, *Building Generative AI Features with Microsoft.Extensions.AI, Microsoft Agent Framework, and Ollama*, extends the eShop with LLM-powered customer support. You will run Llama 3.2 locally through Ollama, integrate it as an Aspire-managed AI backing service, use `Microsoft.Extensions.AI` abstractions such as `IChatClient` to keep application logic provider-independent, expose a support endpoint in the Catalog microservice, and connect the capability to the Blazor frontend.

Chapter 14, *Building Semantic Product Search with Vector Embeddings*, builds on the AI foundation to implement semantic product search. You will generate embeddings with an Ollama-hosted embedding model, represent products as vector records through `Microsoft.Extensions.VectorData`, implement both keyword and semantic search paths, expose dedicated Catalog endpoints, and add a Blazor interface that lets users switch between the two approaches.

To get the most out of this book

The examples are hands-on and are intended to be run as you progress through the chapters. The following setup covers the tools and services used across the book:

Requirement	How it is used
C# and ASP.NET Core knowledge	A working familiarity with C# and basic ASP.NET Core web/API development will help you follow the implementation.
.NET 10	The current project examples target .NET 10.
Visual Studio 2022 or later	Used throughout the book for creating projects, editing code, debugging, and running the Aspire solution.
Aspire 13-compatible tooling	Used to define the application model, orchestrate projects and backing services, provide service discovery, and inspect the distributed application.
Docker Desktop or a supported container runtime	Required for the containerized backing services and local Aspire development used throughout the hands-on chapters.

Requirement	How it is used
Internet access	Required to restore packages, pull container images, and download local AI models such as Llama 3.2 when you reach the generative AI chapters.
Azure subscription	Required for the Azure Container Apps deployment chapter. The subscription must allow you to provision resources such as resource groups and Container Apps.
Azure Developer CLI (azd)	Required for the Chapter 12 deployment workflow, including authentication, initialization, provisioning, deployment, and cleanup.
Git (recommended)	Useful for cloning and updating the example repositories and for moving between project states as you follow the book.

You do not need an Azure subscription for the local-development chapters. Azure becomes necessary when you reach the deployment workflow. Likewise, the AI chapters use an Ollama-hosted model that is downloaded when the Aspire application starts; the first download can take several minutes depending on your network connection, and the model data is then reused from persistent local storage.

Download the example code files

This book includes a complete downloadable code bundle containing all the example projects and files used throughout the chapters. We recommend downloading the bundle so you can follow along smoothly and experiment with the examples.

Use the bundle as a practical starting point. Modify it, extend it, and apply what you learn by creating your own variations as you progress through the chapters.

Get the code bundle

If you bought the book directly from Packt:

1. Go to `packtpub.com`
2. Click your **profile picture** and select **Your Orders**
3. Find this book and click **Download Code**

If you bought this book from Amazon or any other channel partner:

1. Go to `packtpub.com/unlock` or scan the following QR code:



1. Search for this book
2. Sign up or log in to your free Packt account
3. Upload your proof of purchase and download the code bundle locally

Usage note: You're free to use and modify this code for personal learning and non-commercial projects.

Download the color images

Your purchase includes a color, DRM-free PDF copy of this book, ideal for viewing color images, screenshots, and diagrams. Refer to *Free benefits with your book section* at the end of the *Preface* to unlock your PDF copy.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "We will then create the `ShoppingCart` and `ShoppingCartItem` domain models and organize the project into `Models`, `Services`, and `Endpoints` folders."

A block of code is set as follows:

```
# LlamaIndex: explicit next-agent declaration
fundamental_agent = FunctionAgent(
    name="FundamentalAgent",
    description="Get fundamental ratios for a ticker.",
    tools=[get_fundamentals], can_handoff_to=["ProfitabilityAgent"],
)
```

Any command-line input or output is written as follows:

```
Add-Migration InitialCreate -OutputDir Data/Migrations -Project Catalog
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "Next, organize your solution by creating three main folders. Right-click the solution (`eshop-distributed`) in **Solution Explorer**, select **Add** and then **New Solution Folder**."

> **NOTE**
>
> Warnings or important notes appear like this.

> **TIP**
>
> Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packt.com/`.

Share your thoughts

Now you've finished *Building AI-Powered Distributed Architectures with Aspire*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.



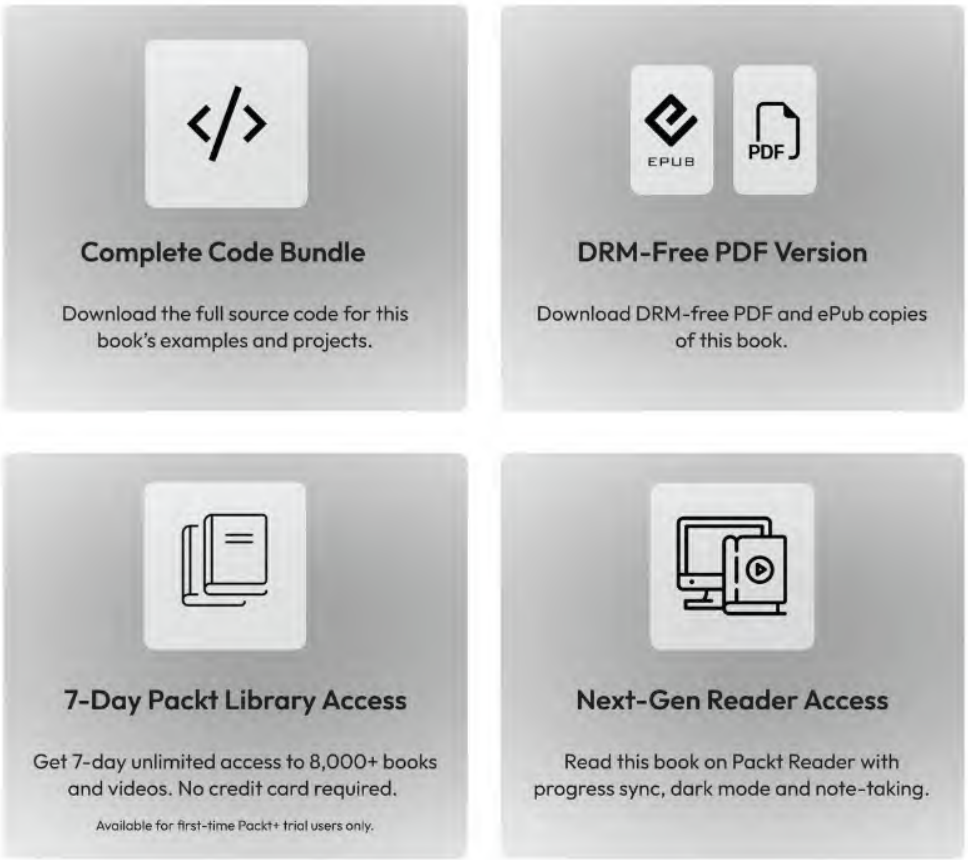
`https://packt.link/r/1808492765`

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Free benefits with your book

This book includes free benefits designed to support your learning and help you apply what you learn effectively. Activate them now for instant access (see the *How to unlock* section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:



How to unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.





Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

Join us on Discord!

Read this book alongside other users, developers, experts, and the author himself.

Ask questions, provide solutions to other readers, chat with the authors via Ask Me Anything sessions, and much more. Scan the QR or visit the link to join the community.



`https://packt.link/deep-engineering`

1

Introducing the AI-Powered eShop Architecture with Aspire

Modern applications rarely operate as a single, self-contained service. They depend on databases, caches, messaging systems, identity providers, external services, and, increasingly, **generative AI** capabilities. As these applications grow, the challenge is no longer simply writing individual services. It is designing, connecting, running, observing, securing, and deploying all of these components as one **distributed system**.

This book takes a practical approach to that challenge. You will build an AI-powered e-commerce application incrementally, using **Aspire** to orchestrate the distributed application and bring its services and infrastructure together. Rather than studying each technology in isolation, you will see how the pieces work together as part of one evolving system.

The application you will build is an eShop that allows customers to browse products, view product information, and manage items in a shopping basket. As the application evolves, you will extend these core capabilities with secure access, **event-driven updates**, cloud deployment, **semantic product search**, and AI-powered customer support.

Behind this user-facing experience is a distributed architecture centered on Catalog and Basket **microservices** and a Blazor web application. As the architecture develops, you will integrate PostgreSQL for product data, Redis for basket storage and caching, RabbitMQ for asynchronous communication, and Keycloak for **authentication and authorization**. You will also explore synchronous service-to-service communication, **observability**, **containerization**, and deployment to Azure Container Apps.

In the later stages of the book, you will extend the same application with generative AI capabilities. Using `Microsoft.Extensions.AI`, **Microsoft Agent Framework**, and **Ollama**, you will add customer-support capabilities and semantic product search. In this way, the book

moves from the foundations of distributed application development through communication, security, deployment, and finally AI-enabled application experiences.

The emphasis throughout is on building the system step by step. Each chapter adds another part of the architecture. This progression helps you understand each technology, its purpose, and its interaction with the components already in place. By the end of the book, you will have followed the evolution of a complete distributed application from its initial architecture through to cloud deployment and AI integration.

Before we begin building the individual services, however, we need a clear picture of the system we are going to create and the architectural principles that guide it. In this chapter, we will introduce the eShop application that we will develop throughout the book, examine its major components and their responsibilities, and see how Aspire brings these resources together as a distributed application.

We will also look at how the architecture evolves throughout the book, giving you a roadmap for the chapters that follow. By the end of the chapter, you will have a clear understanding of the target architecture, the **cloud-native principles** that shape it, Aspire's role in orchestration and observability, and how the solution is organized for the implementation that follows.

In this chapter, we'll be covering the following topics:

- Understanding the eShop application
- Applying cloud-native principles
- Understanding Aspire's role
- Following the architecture as it evolves
- Understanding the solution organization
- Building the application step by step

NOTE

To access the code for this book, follow the repository instructions provided in the **Download the example code files** section of the *Preface*.

Understanding the eShop application

The book follows one practical project from beginning to end: an eShop distributed application composed of independently focused services and the backing resources they need. At the user level, the application supports common e-commerce activities such as browsing the product catalog, viewing product details, and managing a shopping basket. As the project develops,

these core capabilities are extended with security, event-driven communication, cloud deployment, and AI-powered features.

Using the same application throughout gives you a consistent context for understanding service boundaries, communication patterns, persistence, security, observability, deployment, and AI integration.

At the heart of the system are the Catalog and Basket microservices, each responsible for a distinct part of the eShop domain.

The Catalog microservice owns product information. It stores product data in PostgreSQL and exposes operations for listing, retrieving, creating, updating, and deleting products. Because Catalog owns this data, other parts of the application treat it as the authoritative source for product details such as names and prices.

Working alongside Catalog, the Basket microservice manages shopping-cart data and uses **Redis** as a **distributed cache**. When basket operations need current product information, Basket can retrieve it from Catalog rather than relying on values supplied by an incoming request. This creates a clear separation of responsibility: Catalog owns product data, while Basket owns the customer's cart.

The services communicate in two distinct ways. **Synchronous communication** is used when one service needs an immediate response from another. For example, the Basket microservice calls the Catalog microservice over **HTTP** when it needs the latest product information before updating a basket. The calling service waits for that response before continuing.

Asynchronous communication, by contrast, is used when an update needs to be propagated without requiring an immediate response from the receiving service. **RabbitMQ** provides this messaging path. For example, when a product price changes in Catalog, Catalog publishes an **integration event** that Basket can consume and use to update the corresponding prices stored in baskets.

Communication is only one part of the overall application architecture. The eShop also needs to control access to protected operations and provide a user-facing interface for interacting with the system. **Keycloak** provides authentication and authorization, while the **Blazor WebApp** gives users a browser-based interface for browsing products and working with their baskets. Once the core distributed application is in place, later chapters extend it with generative AI capabilities. The generative AI capabilities include customer-support chat and semantic product search based on embeddings and vector storage.

Together, these components form the application you will build throughout the book. Before moving into the implementation, however, it is useful to establish the cloud-native principles that shape how you design, connect, and operate these services.

Applying cloud-native principles

Although the book is hands-on, the implementation follows a small set of cloud-native principles. Each service has a focused responsibility, keeps state in an appropriate **backing service**, and communicates with other components through well-defined interfaces. This structure makes individual parts of the system easier to change, test, observe, and scale without treating the application as one tightly coupled unit.

As the application evolves, you will repeatedly apply the following ideas:

- Keep service responsibilities clear so that data ownership and business behavior have an obvious home.
- Use backing services such as **PostgreSQL**, Redis, RabbitMQ, and Keycloak for capabilities that should remain outside individual application processes.
- Use **service discovery** and injected configuration instead of hardcoding service addresses and ports.
- Observe the distributed application through logs, traces, and resource health so that failures can be diagnosed across service boundaries.
- Automate environment setup and deployment so that the same application model can be reproduced consistently across development and cloud environments.

These principles become concrete in later chapters as you add each microservice and backing service and then connect them through Aspire.

Understanding Aspire's role

Aspire provides the application model and orchestration experience that brings the distributed system together during development. At the center of this setup is the **AppHost project**, which acts as the orchestration project for the application. It is where you declare the projects, containers, backing services, and relationships that make up the distributed system.

Rather than configuring every dependency separately, you describe how these resources relate to one another in AppHost, and Aspire provides the corresponding connection information and service discovery configuration at runtime.

This model is particularly useful as the application grows. Catalog can reference its PostgreSQL database, Basket can reference Redis and Catalog, and later both services can participate in RabbitMQ messaging. You can also express **startup dependencies** so that projects wait for required resources to become available before they begin serving requests.

Aspire also gives you a unified dashboard for inspecting the distributed application. From one place, you can see projects and containers, open their endpoints, examine console and structured logs, and follow traces across service boundaries. Later chapters use these capabilities repeatedly while testing Catalog, Basket, messaging, caching, security, and the web application.

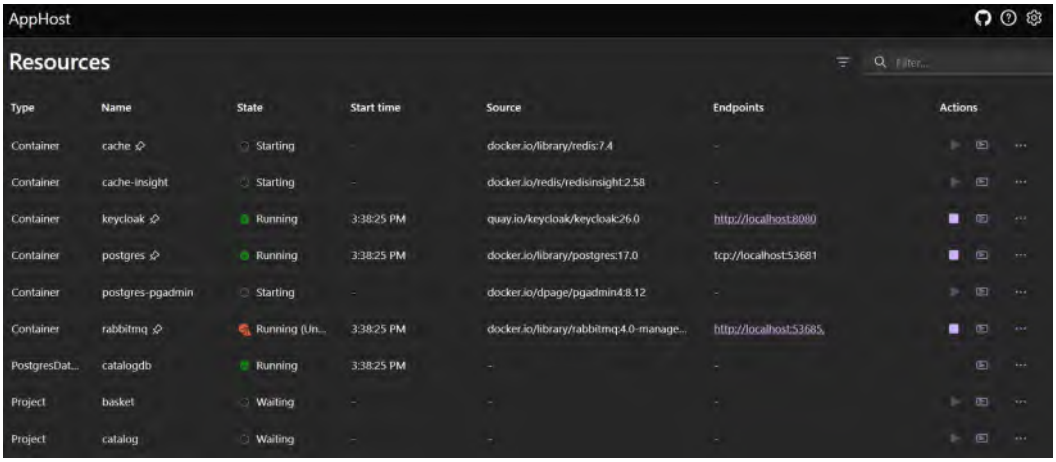


Figure 1.1 – Aspire dashboard showing the projects and backing resources that make up the completed eShop application

Together, the application model and dashboard let you to define the distributed system and observe it as it runs. With that foundation in place, we can now look at how the eShop architecture is introduced and expanded across the book.

Following the architecture as it evolves

The application is intentionally built in stages. Each stage introduces one or two new responsibilities while keeping the earlier pieces in place. This progression lets you understand why a component is needed before you add the next one.

The starting point is the Catalog microservice and its PostgreSQL backing service. RabbitMQ also appears in the wider architecture because it will later carry integration events between services.

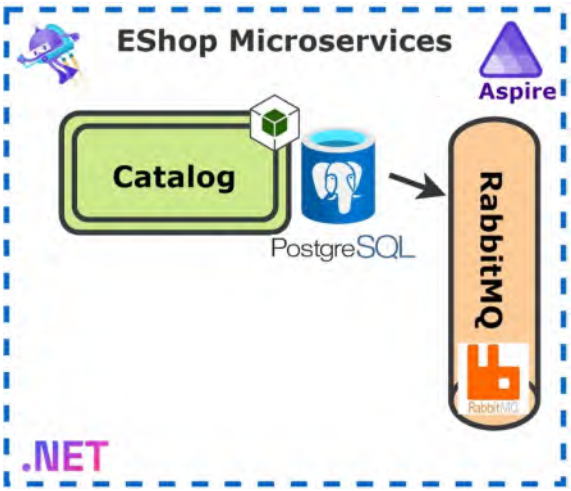


Figure 1.2 – Starting architecture with the Catalog microservice, PostgreSQL, and RabbitMQ

The next stage adds the Basket microservice and Redis (see following figure). In this stage, a second independently focused service is introduced, creating the need for service-to-service communication between Basket and Catalog.

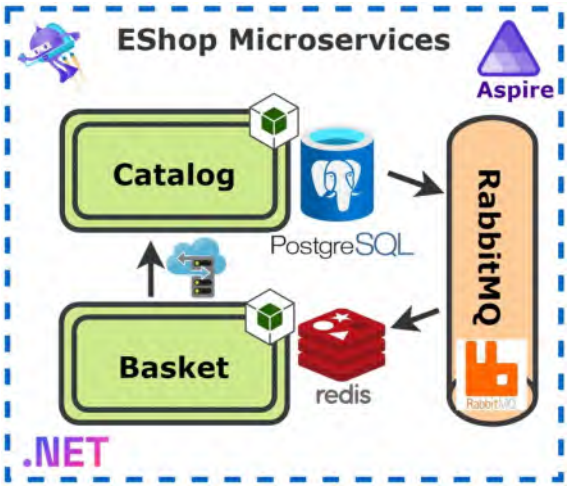


Figure 1.3 – Adding the Basket microservice and Redis to the distributed application

Authentication and authorization are then introduced with Keycloak so that protected Basket operations can require an authenticated user and validate **JWT** bearer tokens.

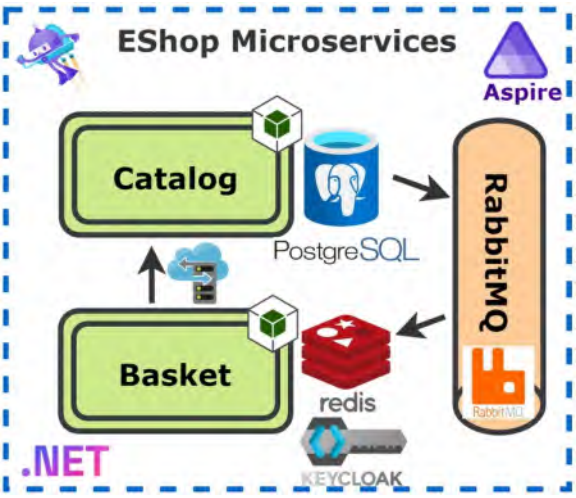


Figure 1.4 – Adding Keycloak for authentication and authorization

Finally, the Blazor WebApp is connected to the backend services and the application is prepared for cloud deployment.

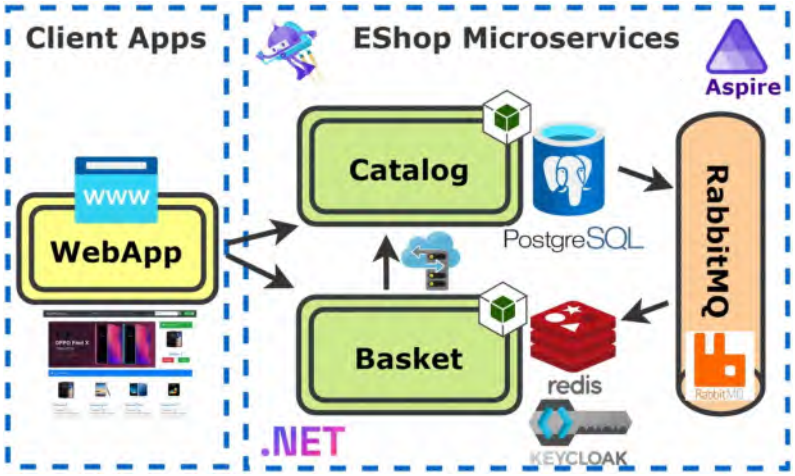


Figure 1.5 – The broader eShop architecture with the WebApp, Catalog, Basket, backing services, and Azure Container Apps

At this stage, the core distributed architecture is in place and can be extended with caching, observability, deployment automation, and AI-powered capabilities. Before moving into those implementations, it is useful to see how the solution is organized in code and where each part of the architecture lives.

Understanding the solution organization

The solution mirrors the architecture by separating orchestration, backend services, and the frontend into three distinct folders. This separation gives each part of the distributed application a clear home and makes the project easier to navigate as more capabilities are added. Let's look at each folder in turn, beginning with the orchestration layer:

- The `aspire` folder contains the AppHost and shared orchestration-related projects. AppHost holds the **application model**: the code that declares projects, backing services, references, and startup relationships. Shared service defaults provide common operational configuration that can be applied consistently across the application.
- The `services` folder contains the backend microservices. Catalog and Basket are developed independently, with each service owning its own API surface and persistence concerns. Later chapters connect these services through HTTP, RabbitMQ, security, and shared observability.
- The `frontend` folder contains the Blazor WebApp. It acts as the client-facing application and communicates with the backend services through their HTTP endpoints. Keeping the frontend separate from the microservices reinforces the same separation of concerns used throughout the architecture.

Together, these three folders map the logical architecture you have just seen to a concrete project structure.

The rest of the book turns this architecture into a working system incrementally. You will first establish the Aspire application and supporting project structure, then develop Catalog with PostgreSQL and Basket with Redis. From there, you will add synchronous service-to-service communication and asynchronous messaging with RabbitMQ and **MassTransit**. You will then configure authentication with Keycloak, add the Blazor frontend and caching behavior, and deploy to Azure Container Apps. The final stages extend the application with generative AI capabilities for customer support and semantic search.

Because each chapter builds on the same eShop solution, the application becomes progressively more complete rather than being replaced by unrelated examples. By the time you reach the later chapters, the services, infrastructure, security, observability, deployment, and AI features all form one connected distributed application.

Summary

In this chapter, you were introduced to the AI-powered eShop application that you will build throughout the book and established the architecture that will guide the implementation. We looked at the roles of the Catalog and Basket microservices, the Blazor WebApp, and the backing services that support them, including PostgreSQL, Redis, RabbitMQ, and Keycloak. We also distinguished between synchronous HTTP communication, where one service waits for an immediate response, and asynchronous messaging, where events can be propagated independently between services.

You then examined how Aspire brings these components together during development. You saw how the AppHost project defines the application model and the relationships between projects and backing resources, while Aspire provides service discovery, connection information, startup coordination, and a unified dashboard for observing the distributed application. You also saw how to organize the solution into separate `aspire`, `services`, and `frontend` folders and how the architecture will evolve incrementally as the book progresses.

With the overall application and its structure now established, the next step is to understand the architectural principles behind it. In *Chapter 2*, you will explore cloud-native distributed architectures, including microservices, containers, orchestration, scalability, **CI/CD**, observability, and the role of backing services. You will also see how traditional services such as databases, caches, message brokers, and identity providers can be extended with **AI**-powered services such as **LLMs** and vector databases to support intelligent application capabilities.

2

Cloud-Native Distributed Architectures and Backing Services

Modern applications are expected to scale on demand, remain available when individual components fail, and evolve rapidly without disrupting the entire system. Cloud-native distributed architectures make this possible by combining independently deployable microservices, containers, automated delivery pipelines, and dynamic orchestration. They also rely on backing services that provide essential capabilities such as data storage, caching, messaging, authentication, monitoring, and observability.

Unlike traditional applications that are simply moved to virtual machines in the cloud, cloud-native distributed architectures are designed from the ground up to take advantage of cloud elasticity, on-demand resources, automated scaling, dynamic orchestration, and modularity.

In this chapter, you will explore the principles and characteristics that distinguish cloud-native distributed systems from monolithic applications and systems that have simply been migrated to the cloud. You will examine how traditional backing services, including relational databases, Redis, RabbitMQ, and Keycloak, support distributed applications. You will then learn how AI-powered backing services, such as **LLMs** and vector databases, can enable semantic search, question answering, and **retrieval-augmented generation** (**RAG**).

By the end of the chapter, you will understand how established cloud-native patterns and emerging AI services work together to support scalable, resilient, and intelligent applications.

In this chapter, we'll be covering the following topics:

- Defining cloud-native distributed architectures
- Backing services in cloud-native applications

Defining cloud-native distributed architectures

Cloud-native distributed architectures are applications designed from the ground up to leverage cloud platforms and services, rather than those that are simply migrated to the cloud. They are built for the cloud to benefit from elasticity, on-demand resources, and automated scaling. These architectures embrace services and containers, breaking down large applications into smaller units packaged into containers for consistency. They rely on automation and CI/CD, using DevOps pipelines for continuous integration, delivery, and deployment.

In a distributed architecture, functionality spans multiple independent services that communicate over a network. This separation of concerns allows each service to be developed, deployed, and scaled independently. It also provides reliability, in that if one service fails, others can continue operating, reducing system-wide downtime. This also increases scalability, since services handling heavy loads can be scaled up or down without affecting others.

The main elements of cloud-native distributed architectures include microservices, containers, DevOps, CI/CD, and a variety of backing services.

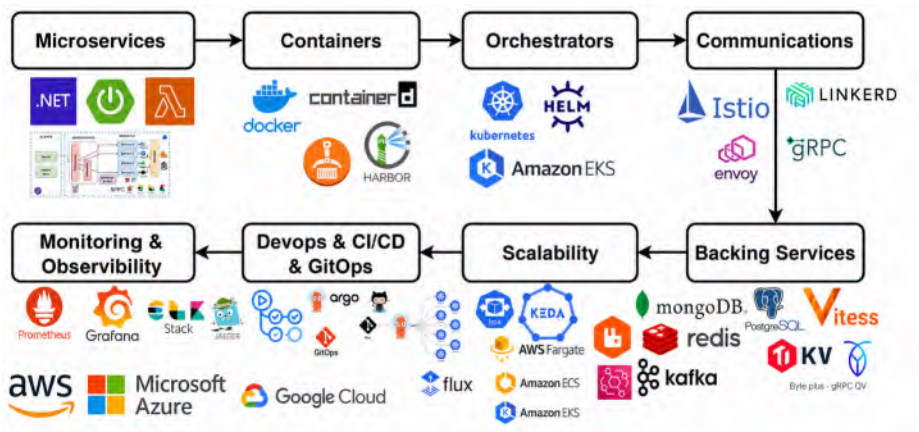


Figure 2.1 – Main elements of cloud-native distributed architectures: microservices, containers, DevOps, CI/CD, and a central backing services hub with API gateways, service meshes, logging, monitoring, tracing, search analytics, databases, distributed caching, message brokers, event streaming, object storage, authentication, and security identity management

Cloud-native distributed architectures are defined by several characteristics that work together to support scalability, resilience, and independent deployment. You can look at these characteristics in more detail, beginning with microservices.

The first is **microservices**: in a cloud-native environment, you break your application into logical units called microservices. Each microservice represents a **bounded context** and focuses on a specific business capability, such as product catalog management, shopping basket management, or order processing. Each microservice has its own data store, runs independently, and communicates with other microservices through well-defined **APIs** or event-driven messaging.

Another key characteristic is **containerization**. You package each microservice in a container, which helps ensure consistent environments across development, staging, and production. Containers bundle the application with its dependencies, making them lightweight and portable and providing a consistent experience across development, testing, and production.

Container orchestrators such as Kubernetes and managed platforms such as Azure Container Apps are essential for managing cloud-native applications. They provide capabilities such as automatic scaling, high availability, service discovery, and self-healing. They also help isolate workloads, so if one container crashes, it does not necessarily affect others, improving overall reliability.

Communication between microservices is another important consideration. Microservices communicate over the network using protocols such as **HTTP** or `gRPC`. In dynamic cloud environments, **service discovery** enables services to locate and communicate with one another without relying on hard-coded network addresses. In addition to communicating with each other, microservices also depend on **backing services** – external components that provide capabilities such as data storage, distributed caching, message brokering, and authentication.

Scalability is another key characteristic, with orchestrators and cloud providers offering autoscaling capabilities. **DevOps**, **CI/CD**, and **GitOps** practices bring development and operations together for seamless collaboration. *Continuous integration* automatically builds and tests your application whenever code changes are merged, while continuous delivery and deployment ensure that you can deploy new features and fixes rapidly and reliably.

Finally, **monitoring and observability** are vital in cloud-native distributed architectures. Metrics can be collected using tools such as Prometheus and visualized through Grafana, while centralized logging helps bring application logs together for analysis. **Distributed tracing**, using tools such as Jaeger and Zipkin, allows developers to track a single request as it travels across multiple microservices. The tracing process makes it easier to diagnose performance bottlenecks and failures.

These characteristics explain how distributed architectures allow independent scaling, deployment, and resilience across services.

Among these characteristics, backing services play a particularly important role because they provide the external capabilities that enable cloud-native applications to store data, exchange messages, cache frequently accessed information, and integrate with supporting infrastructure. Let's examine these services in more detail.

Backing services in cloud-native applications

In cloud-native distributed architectures, you rely on a variety of external services to support your application's core functionality. These backing services are essential components that your application depends on for operations such as data storage, messaging, caching, and authentication.

Backing services are external resources or components that your cloud-native application uses, but they are not part of the application's own codebase. Instead, you treat them as attached resources that you can swap or replace without changing your application's core logic. This decoupling of backing services from the application itself promotes flexibility, scalability, and easier maintenance.

For example, you might use **SQL** or **NoSQL** databases for persistent storage, distributed caching services such as Redis or Memcached to cache data, and message brokers such as RabbitMQ, Kafka, or AWS **SQS** to enable asynchronous communication. Event streaming platforms, object storage solutions, and authentication and authorization services such as Keycloak or AWS Cognito are also common backing services in cloud-native architectures.

The following figure shows a visual representation of these backing services in a cloud-native setup:

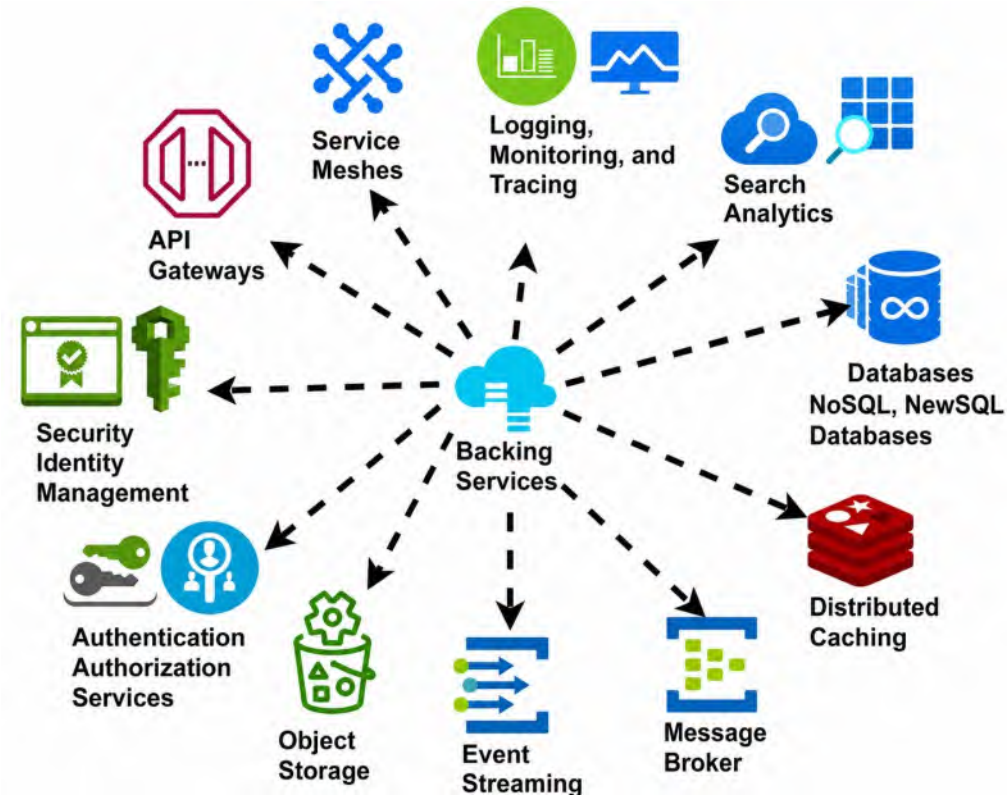


Figure 2.2 – Backing services for cloud-native architectures

Now, let's examine the backing services used by the **eShop** distributed application that you will be building. The Catalog microservice uses PostgreSQL as a backing service for database operations. For the Basket microservice, Redis is used as a backing service, functioning as a distributed cache to improve performance and scalability. The RabbitMQ message broker is used for asynchronous communication, facilitating messaging between microservices. Keycloak provides identity and access management, handling authentication and authorization to secure the Basket endpoints.

The following diagram shows how these backing services connect to the eShop microservices.

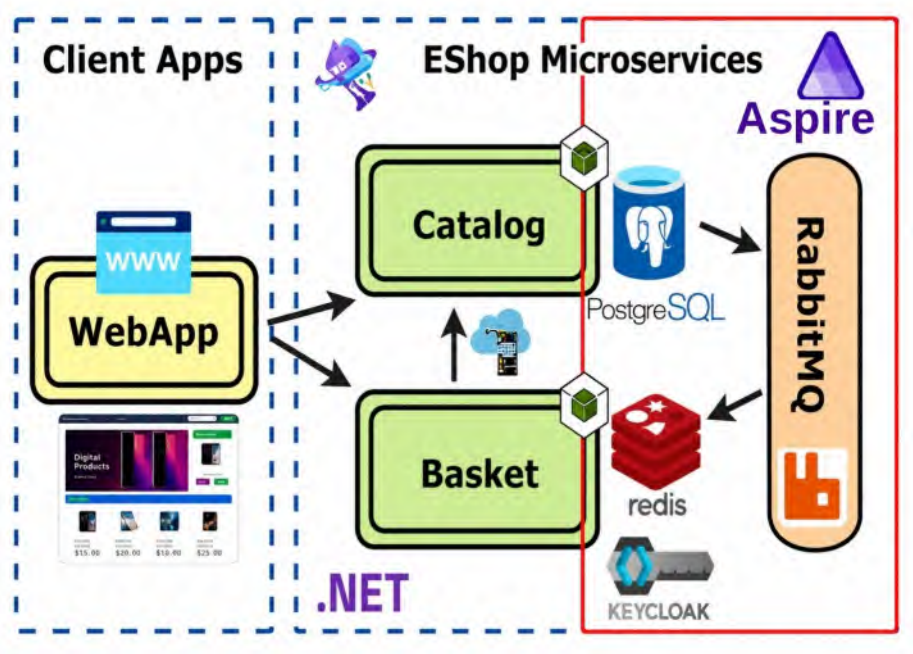


Figure 2.3 – eShop microservices backing services

As you design and scale enterprise applications, you usually rely on traditional backing services such as relational databases, distributed caches, and message brokers to provide essential functionality and scalability. These services form the backbone of your microservice architecture, supporting features such as data storage, messaging, and caching.

AI-powered services such as LLM inference services, **embedding services**, and **vector databases** extend this foundation with capabilities such as semantic search, **RAG**, recommendations, and intelligent automation. These AI services complement rather than replace traditional backing services. Used together, traditional and AI backing services allow cloud-native applications to combine reliable infrastructure with more intelligent and context-aware functionality.

The following figure contrasts these two categories, showing how AI services such as LLMs and vector databases can be integrated alongside established components.

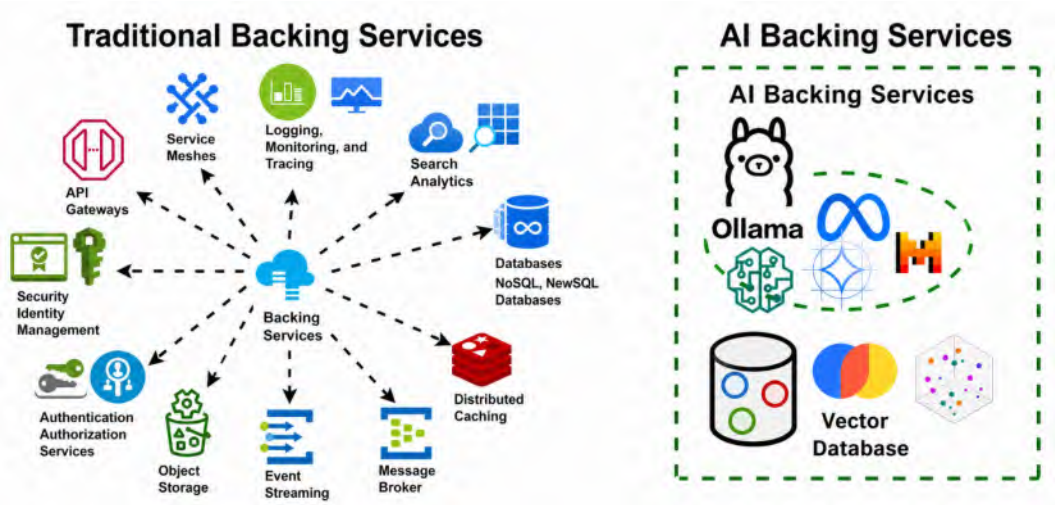


Figure 2.4 – Diagram comparing traditional backing services (databases, distributed caching, message brokers, event streaming, object storage, authentication/authorization, security identity management, API gateways, service meshes, logging/monitoring/tracing, search analytics) with AI backing services (Ollama, vector databases, Llama, Phi)

Two important types of AI backing services in this architecture are LLM inference services and vector databases. Let's look at how each extends the existing cloud-native foundation.

With the rise of AI-powered applications, cloud-native architectures increasingly incorporate AI backing services alongside traditional ones. One example is an **LLM inference service** such as Ollama, which enables you to run models like Llama and Phi locally on your own infrastructure. Hosting models locally provides greater privacy, lower latency, and independence from third-party APIs. Like other backing services, Ollama exposes HTTP or gRPC endpoints that microservices consume to perform AI tasks such as text generation, summarization, or question answering, making it a natural extension of a cloud-native distributed architecture.

For example, your application can use Llama and Phi models, which are advanced **language models** capable of understanding and generating human-like text responses. By running these models locally with Ollama, you maintain control over your data and inference process.

You also use vector databases as a backing service. A vector database is a specialized data store for **embeddings**, which are high-dimensional vector representations of text or images. In your application, you might store product descriptions transformed into numerical vectors, enabling semantic search and AI workflows. The vector database supports operations, such as cosine similarity. **Cosine similarity** is a common metric for measuring how close two vectors are and indicates semantic similarity between user queries and product descriptions.

You'll also be able to implement RAG workflows, where relevant data is pulled from the vector database to enhance LLM responses. This integration allows your application to provide advanced features such as Q&A support, semantic search, and more.

You connect these AI services to your existing infrastructure through integration points such as **Microsoft Agent Framework** and the `Microsoft.Extensions.AI` library. These tools allow your microservices to communicate with AI backing services over the network and consume them in much the same way as other external services.

By adopting AI as a backing service, you treat LLMs and vector databases the same way you treat databases and other traditional backing services. This approach allows you to make your application intelligent by embedding AI capabilities directly into your cloud-native distributed architecture.

Summary

In this chapter, we explored how cloud-native distributed architectures enable scalable and resilient applications by combining microservices, containers, dynamic orchestration, automation, and supporting backing services.

We began with the characteristics that distinguish cloud-native distributed systems from traditional monolithic applications or systems that have simply been migrated to the cloud. We examined how microservices, containerization, service discovery, scalability, CI/CD, and observability allow individual parts of a distributed application to be developed, deployed, scaled, and monitored independently.

We then looked at the role of backing services in cloud-native applications. Using the eShop architecture as our reference, we saw how **PostgreSQL** provides persistent storage for Catalog, **Redis** supports distributed caching for Basket, **RabbitMQ** enables asynchronous communication, and **Keycloak** provides authentication and authorization.

Finally, we extended the concept of backing services to AI-powered capabilities. We examined how LLM inference services such as Ollama and vector databases complement traditional infrastructure by enabling features such as semantic search, question answering, RAG, recommendations, and intelligent automation. We also saw how Microsoft Agent Framework

and `Microsoft.Extensions.AI` can provide integration points between microservices and these AI services.

With this foundation in place, we are ready to put these architectural concepts into practice on the .NET platform. In the next chapter, we will introduce **Aspire 13** and explore how it brings orchestration, integrations, and service discovery together to simplify the development of cloud-native distributed applications. We will see how Aspire models application resources and their dependencies, connects microservices to backing services, and reduces the manual configuration required to coordinate a distributed system.

3

Developing Distributed Applications with Aspire

In the previous chapter, we explored the foundations of cloud-native distributed architectures, including microservices, containers, service discovery, and the backing services that support capabilities such as data storage, caching, messaging, and authentication. With those concepts in place, you can now see how to apply them in practice on the .NET platform.

Building cloud-native distributed applications involves much more than writing individual microservices. Developers must provision supporting infrastructure, configure service-to-service communication, manage dependencies, connect applications to databases, caches, and message brokers, and ensure that the entire system can be monitored and run consistently across different environments. As the number of services grows, managing this infrastructure can quickly become as challenging as developing the application itself.

Aspire addresses these challenges by providing a unified, code-first development experience for distributed applications. Aspire 13 is a multi-language application platform with first-class support for .NET, Python, and JavaScript. In this book, we focus on its .NET capabilities. Aspire provides tooling and integrations that simplify orchestration, service discovery, resource configuration, and observability across a distributed application.

In this chapter, you will explore the core concepts that make Aspire an effective platform for distributed application development. You will see how Aspire orchestrates application resources, simplifies integration with the backing services introduced in the previous chapter, enables service discovery, and streamlines the development of containerized microservices. By the end of the chapter, you will understand how these capabilities work together to reduce infrastructure complexity and provide a foundation for building and coordinating distributed applications with .NET.

In this chapter, we'll be covering the following topics:

- Introducing Aspire
- Core concepts of Aspire
- Orchestration in Aspire
- Integrations in Aspire
- Service discovery in Aspire

Introducing Aspire

In this section, we'll introduce Aspire, a code-first orchestration and observability platform for distributed applications. Aspire 13 is a **multi-language** platform with first-class support for .NET, Python, and JavaScript. In this book, we use its .NET support to build and coordinate the eShop application.

Because this book focuses on .NET, the examples use ASP.NET Core and C# together with Aspire's AppHost, integrations, service discovery, and observability features.

Rather than extending ASP.NET Core itself, Aspire provides a code-first *application model* for defining services, containers, backing services, and their relationships. It then uses that model for local orchestration, configuration, service discovery, diagnostics, and deployment workflows.

Aspire offers a set of key features designed to accelerate and standardize your distributed application development. You benefit from pre-built templates and scaffolding, seamless integration with cloud-native platforms such as Azure Container Apps, and support for popular backing services such as Azure Service Bus.

The first key feature that Aspire offers is **pre-built templates and scaffolding**. Aspire provides project templates that jumpstart your microservice creation. This scaffolding ensures you follow a consistent pattern for folder structure, dependency injection, configuration, and logging.

Another important feature is that Aspire is built for **cloud-native architectures**. It integrates easily with Docker, container , orchestrators such as Kubernetes, and managed , platforms such as Azure Container Apps. It also simplifies the development of cloud-native microservices by supporting patterns such as service discovery, API integration, and event-driven communication.

Aspire also supports an **infrastructure-as-code approach** by allowing you to define application resources and their relationships directly in the AppHost project. For example, you can declare backing services such as PostgreSQL, Redis, or message brokers in code, and Aspire can provision and connect them when you run the application locally. This gives you a repeatable development and testing environment while reducing the manual effort required to configure and run multiple services individually.

You also have **integration with common backing services** out of the box. Aspire supports popular databases like PostgreSQL and MongoDB, distributed caches like Redis, and message brokers such as RabbitMQ and Azure Service Bus.

Another key feature is its **modular and extensible architecture**. Services in an Aspire solution are typically decoupled, so you can add new microservices without refactoring the entire system.

Finally, Aspire is **easy to adopt**. If you already know ASP.NET Core and the latest .NET versions, the learning curve is minimal, making it straightforward to start building cloud-native distributed applications.

The main reason for choosing Aspire is that it accelerates development. Standard templates and tooling help teams quickly stand up new **microservices**. It also incorporates best practices by default, providing built-in patterns for logging, configuration, and container orchestration that promote consistency and maintainability across services.

Aspire also comes with a cloud-native mindset, encouraging **event-driven communication**, **containerization**, and **horizontal scalability** from day one. Aspire has seamless integration, connecting readily to external services, databases, message brokers, and identity providers with minimal extra plumbing. In short, Aspire is all about faster development of cloud-native microservices, combining the power of the .NET platform with proven patterns and out-of-the-box integrations.

Now that we have an understanding of Aspire's features and benefits, we can now examine the core concepts that enable it to simplify distributed application development.

Core concepts of Aspire

Aspire introduces three main concepts that underpin its approach to distributed application development: **orchestration**, **integrations**, and **service discovery**.

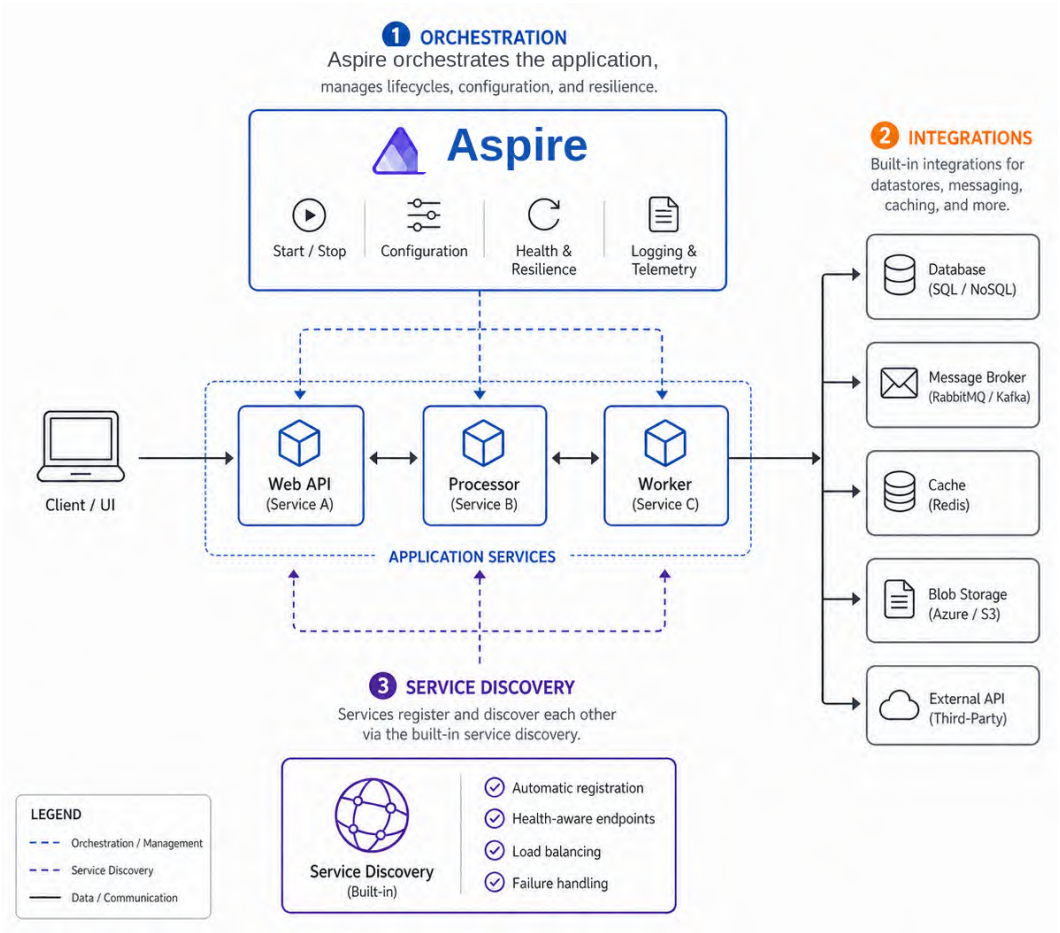


Figure 3.1 – Core concepts of Aspire

These concepts work together to coordinate resources, simplify service connectivity, and enable dynamic communication between services. You will explore how orchestration manages resource startup and dependencies during development. You will also see how integrations reduce boilerplate when adding common resources and how service discovery allows services to locate one another without hard-coded addresses.

Orchestration

Orchestration in Aspire is about coordinating how Aspire starts, configures, connects, and monitors application resources, configured, connected, and monitored during development. You do not need to maintain a large Docker Compose file simply to start the entire application locally. When you run the `AppHost`, Aspire starts the projects, containers, and backing services in the application model and wires their dependencies together.

Another key feature is *environment awareness*. Aspire uses an environment configuration, such as `local`, `test`, or `production`, to decide which services to start and how to configure them. This environment awareness ensures that your development team can run a fully containerized setup locally, and that configurations remain consistent across all stages of deployment.

You can also express service dependencies explicitly. If a microservice must wait for a database to be ready, `WaitFor` tells Aspire to delay starting the dependent resource until the database is ready. This explicit dependency declaration helps prevent startup failures caused by missing dependencies.

For production, the `AppHost` model can also be used by Aspire's publish and deploy workflows to generate or apply target-specific deployment artifacts. The target platform, such as Kubernetes or Azure Container Apps, handles runtime scaling, routing, and service life cycle.

Integrations

Integrations in Aspire provide built-in connectors for common cloud and on-premises services. You can add databases like PostgreSQL and MongoDB, caches like Redis, message brokers such as RabbitMQ and Azure Service Bus, and identity providers like Keycloak with minimal code and no extra Docker files. These pre-built integrations reduce the boilerplate required to configure each service in your application.

Aspire also includes cross-cutting , features such as default setups for logging, monitoring, and **telemetry**. This saves you time and promotes consistent patterns across your microservices. You do not need to start from scratch when implementing new capabilities, as Aspire handles much of the heavy lifting behind the scenes. Logging, telemetry, and monitoring integrations are part of Aspire's offering, ensuring a consistent approach across multiple microservices.

The modular and extensible design of Aspire means you only bring in the integrations you need. Adding services such as **AI**-based features or **vector databases** requires minimal adjustments, thanks to Aspire's pluggable architecture. This approach saves you time and promotes best practices from the start.

Service discovery

Service discovery in Aspire addresses dynamic service endpoints in distributed environments. When one resource references another with `WithReference`, Aspire injects service-discovery configuration into the consuming resource. Application code can then use a logical service name, and Aspire's endpoint resolver maps that name to the actual endpoint at runtime. This service-discovery configuration removes the need to hardcode **URLs** or maintain a separate service registry for the local application model.

By combining orchestration, integrations, and service discovery, Aspire coordinates application resources and dependencies. It simplifies adding and configuring backing services and enables services to locate one another without manual endpoint configuration. This synergy makes Aspire an effective choice for teams building cloud-native and event-driven distributed solutions.

In the upcoming sections, you will delve into the core concepts, starting with orchestration.

Orchestration in Aspire

Before diving into code, let's define the core concepts of Aspire orchestration:

- **App model:** This is the code-first model of your distributed application. It declares the resources your solution needs and their relationships.
- **AppHost**: This is the code-first project that defines the application model and drives local orchestration. In this book, the AppHost is written in C#.
- **Resource**: This is any item you manage, such as an ASP.NET Core web project, a Redis cache, or a Docker container.
- **Integration**: This is a way to quickly add a service such as Redis or Postgres through NuGet packages, so you don't have to write a lot of boilerplate code.
- **Reference**: This expresses a dependency, for example, a front-end project that references a Redis cache.

Understanding this terminology will help you read the code and conceptualize how Aspire orchestrates your distributed application.

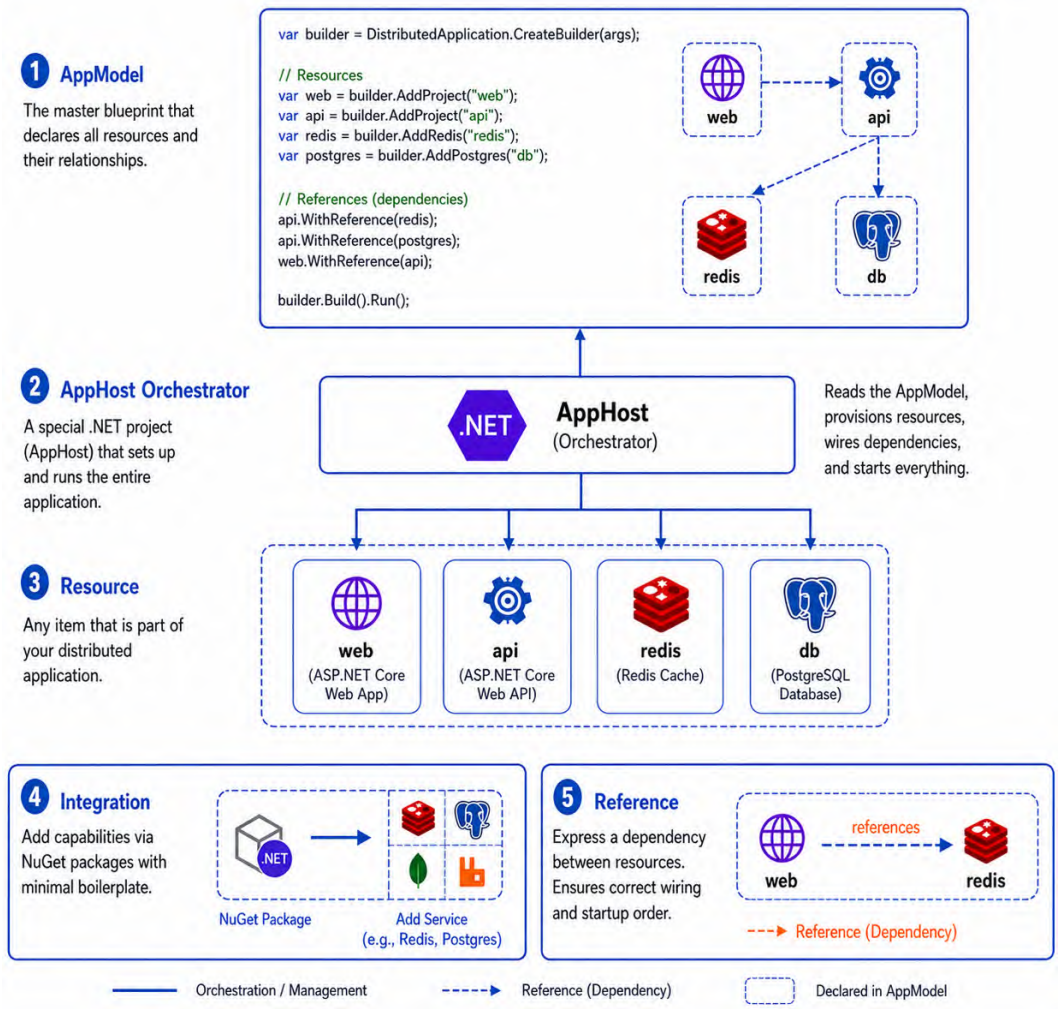


Figure 3.2 – Core concepts of Aspire orchestration

To begin orchestrating your distributed application in Aspire, you first define the app model. The app model describes resources, such as microservices and external services, and their relationships. In this chapter, you define this model in the C# AppHost project.

You start by creating a builder using the following code, which returns an `IDistributedApplicationBuilder` that you use to declare your resources:

```
var builder = DistributedApplication.CreateBuilder(args);
// Add resources
// Express dependencies
builder.Build().Run();
```

The app model is where you connect microservices to databases, caches, or other external services and express their dependencies. When you call `builder.Build().Run()`, Aspire builds the application model and starts the configured resources for local orchestration, applying the declared configuration and dependency relationships.

Then, we have the AppHost project, which acts as the **orchestrator** for your solution. This is where you define the application model in code, including the resources that make up the distributed application and the relationships between them. The following project file shows the configuration for the AppHost project:

```
<Project Sdk="Aspire.AppHost.Sdk/13.0.0">
  <!-- Aspire.Hosting.AppHost is included implicitly by Aspire.AppHost.Sdk. -->
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net10.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference
      Include="Aspire.Hosting.Redis"
      Version="13.0.0" />
  </ItemGroup>
</Project>
```

In Aspire 13, the AppHost project file declares the `Aspire.AppHost.Sdk` directly in the top-level `<Project> element`. The **SDK** tells Aspire that this project is an AppHost and automatically includes the `Aspire.Hosting.AppHost` package. The AppHost targets .NET 10 in this chapter, so the explicit `IsAspireHost` property and `Aspire.Hosting.AppHost` package reference used by earlier templates are no longer required. Hosting integrations used by the AppHost, such as `Aspire.Hosting.Redis`, remain explicit package references.

Let us go through a concrete example of how to define resources and dependencies inside your AppHost project's `Program.cs` file:

```
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();
```

You start by creating a distributed application builder, then add resources and set up dependencies between them.

This code sets up a Redis container resource named `cache`, an **API** service project resource named `apiservice`, and a frontend project resource named `webfrontend`. The code configures the frontend to expose **HTTP** endpoints, reference the cache and API service, and wait for both to be ready before starting.

You use the `WithReference` and `WaitFor` methods to express dependencies between resources. For example, the frontend project depends on the Redis cache and must wait until the cache is fully up before starting. Similarly, the frontend waits for the API service to be running before it starts.

After defining all resources and their relationships, calling `builder.Build().Run();` builds the application model and launches the configured resources in dependency-aware order.

You can visualize this setup as a diagram where the AppHost project orchestrates a container resource (`cache`), a project resource (`apiservice`), and another project resource (`webfrontend`), with arrows indicating dependencies and the flow of connection strings and service endpoints.

The following figure illustrates how the AppHost project manages these resources and their dependencies:

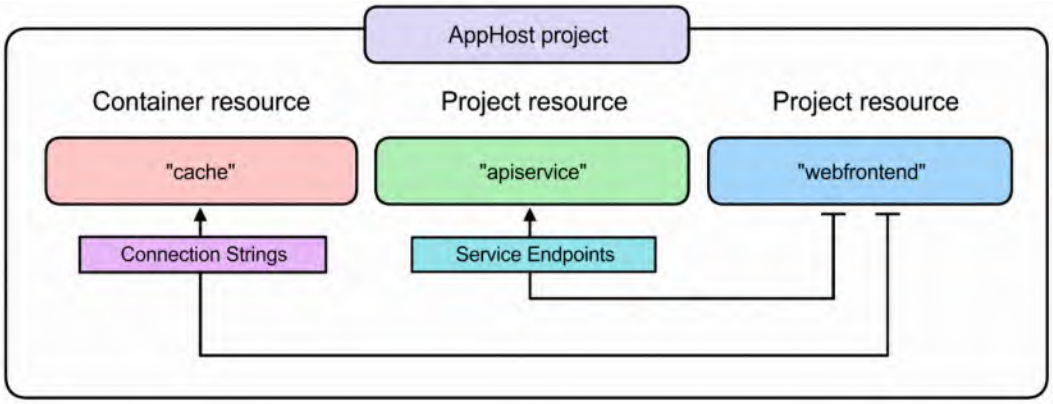


Figure 3.3 – Diagram showing AppHost project orchestrating container resource "cache," project resource "apiservice," and project resource "webfrontend," with arrows for connection strings and service endpoints

You can also add different kinds of resources to your AppHost using built-in APIs. Some commonly used APIs are the following:

- `AddProject`: References a .NET project within your solution.
- `AddContainer`: Adds any Docker image resource you want to bring into your ecosystem.
- `AddExecutable`: Adds an executable, such as a Node.js application or other external process.
- `AddParameter`: Passes external parameters such as environment variables or secrets.

In sum, you define your app model in the AppHost project, add and configure resources such as microservice projects, caches, and databases using these built-in integrations, and then use `WithReference` and `WaitFor` to express dependencies and ensure the correct startup order. Building and running the AppHost then starts the configured resources, wires networking and service discovery, and injects the required environment variables.

Integrations in Aspire

Integrations in Aspire are specialized **NuGet packages** that simplify connecting your microservices to external resources such as caches, databases, or message brokers. These integrations automate tasks such as container provisioning and standardize configuration patterns using environment variables, allowing you to adopt cloud-native features without writing extensive boilerplate code. You will explore the two main types of integrations – hosting integrations and client integrations – and see how they work together or independently to support your distributed application.

Hosting integrations for resource provisioning

Hosting integrations are added to your AppHost project and are responsible for modeling resources such as databases, caches, or message queues. They instruct Aspire on how to provision or reference these services. For example, you can use the `Aspire.Hosting.Redis` or `Aspire.Hosting.PostgreSQL` NuGet packages to automatically manage Redis or PostgreSQL resources.

Client integrations for resource consumption

Client integrations are included in your consumer projects, such as an ASP.NET Web API project. These packages handle the setup for **dependency injection**, telemetry, and health checks, making it easier for your microservices to consume external resources. Examples include `Aspire.StackExchange.Redis.DistributedCaching` or `Aspire.Npgsql.EntityFrameworkCore.PostgreSQL`, which connect your client projects to the necessary resources.

You can use hosting and client integrations together or independently, depending on your needs. If you require both provisioning and a client library for your microservices, you can combine them. Alternatively, you might choose to use only hosting integrations for resource provisioning or only client integrations for resource consumption.

The following diagram illustrates how hosting integrations in the AppHost project provide configuration (such as environment variables) to client integrations in your .NET projects.

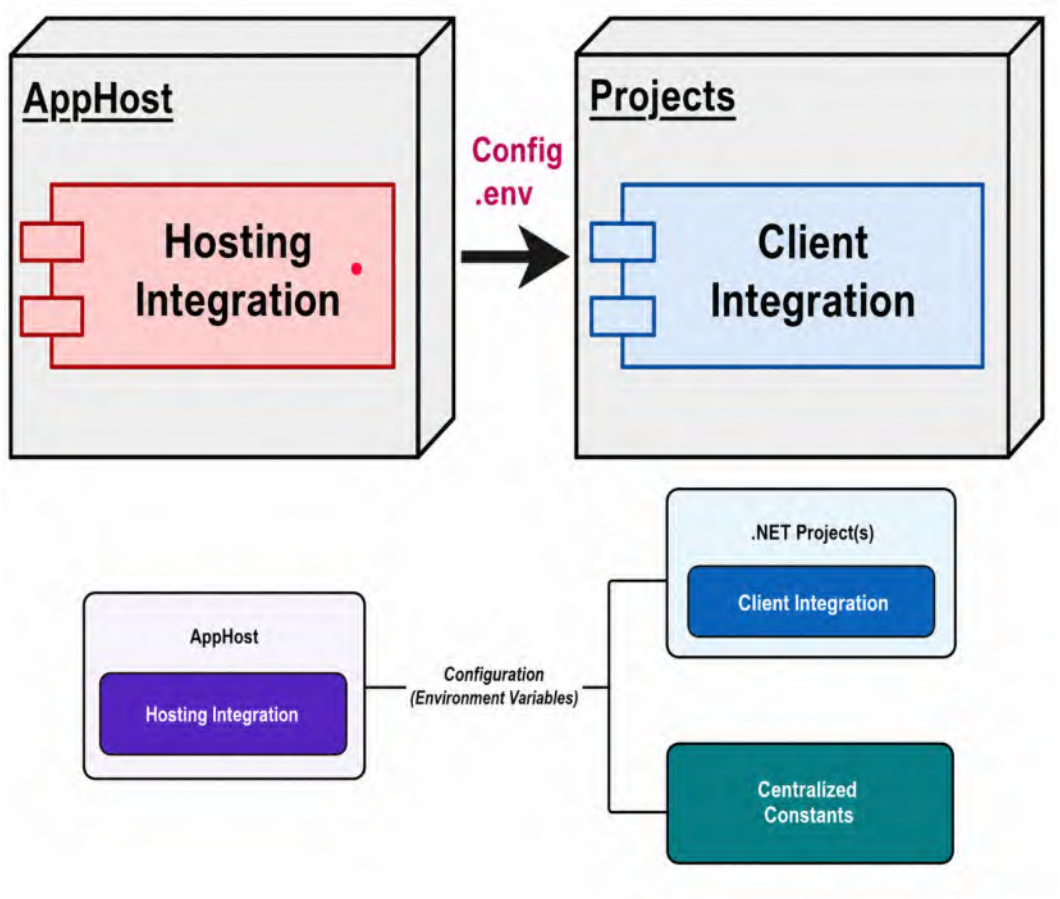


Figure 3.4 – Relationship between hosting and client integrations

Hosting integrations — Resource provisioning via AppHost integrations

Hosting integrations focus on provisioning the resources your distributed application needs. These integrations are attached to your AppHost project, which acts as the orchestrator, and they extend the `IDistributedApplicationBuilder` interface by adding methods such as `AddRedis` and `AddPostgreSQL`. When you use these methods, hosting integrations automatically provision container-based resources or point to an existing instance, whether local or remote. They handle passing connection strings to dependent projects using environment variables or built-in logic.

For example, to add a Redis hosting integration in your AppHost project, use the `AddRedis` method and specify the cache name as `"cache"`. By doing this, Aspire will automatically spin up a Redis container or resource as needed.

```
var builder =
    DistributedApplication.CreateBuilder(args);
// Hosting integration call:
var cache = builder.AddRedis("cache");
// This sets up a Redis container named "cache"
```

Client integrations — Extending Host Builder for Resources

Client integration connects your microservices to resources by extending the `IHostApplicationBuilder` in the consuming project. In the `Program.cs` of an ASP.NET Web API project, you typically use the `builder.AddRedisDistributedCache` extension method. The `AddRedisDistributedCache` method sets up health checks, logging, and metrics for cloud-native patterns.

For example, by referencing the `Aspire.StackExchange.Redis.DistributedCaching` client NuGet package and adding the `AddRedisDistributedCache` extension method with the connection name `"cache"`, you can connect to Redis resources without manual connection strings or configuration.

```
var builder = WebApplication.CreateBuilder(args);// Register shared service
defaults.builder.AddServiceDefaults();// Provided by
Aspire.StackExchange.Redis.DistributedCaching.builder.AddRedisDistributedCache(con
nectionName: "cache");
```

Next, let's examine the relationship between hosting and client integrations.

Hosting and client integration workflow

While hosting and client integrations in Aspire work together, they are not tightly coupled. The hosting integration in your AppHost project defines how to provision or attach resources, such as a Redis cache, and injects environment variables like connection strings into any project that references these resources. The client integration in your microservice projects reads these environment variables or standard configurations to automatically set up the connection to the resource, such as Redis, and enables advanced features like telemetry, resilience, and health checks.

To illustrate, the following code in your AppHost project uses a hosting integration to set up a Redis container named `cache`:

```
var builder = DistributedApplication.CreateBuilder(args);
// Hosting integration call:
var cache = builder.AddRedis("cache");
// This sets up a Redis container named "cache"
```

In your .NET project, you use a client integration to consume the Redis resource by reading the injected configuration:

```
var builder = WebApplication.CreateBuilder(args);// Register shared service
defaults.builder.AddServiceDefaults();// Provided by
Aspire.StackExchange.Redis.DistributedCaching.builder.AddRedisDistributedCache(con
nectionName: "cache");
```

You can use hosting integrations alone, client integrations alone, or both, depending on your requirements. Aspire integrations go beyond simply providing a connection string; they can automatically set up observability, logging, tracing, metrics, health checks, and resiliency by default.

Aspire offers official integrations for many popular services, including cloud-agnostic options such as Redis, PostgreSQL, RabbitMQ, Kafka, and Keycloak. There are also Azure and AWS packages for services such as Azure Storage, Azure Key Vault, and Azure Service Bus, as well as a community toolkit with additional integrations contributed by the community, such as Milvus and MongoDB.

These packages are available from NuGet. To explore official and community integrations, visit the Aspire integrations catalog (`https://aspire.dev/integrations/`).

Service discovery in Aspire

Service discovery in Aspire lets services locate each other without hardcoding addresses or ports. When a project references another service with `WithReference`, Aspire injects service-discovery configuration into the consuming project. Application code can then use the referenced resource's logical name rather than a fixed host or port.

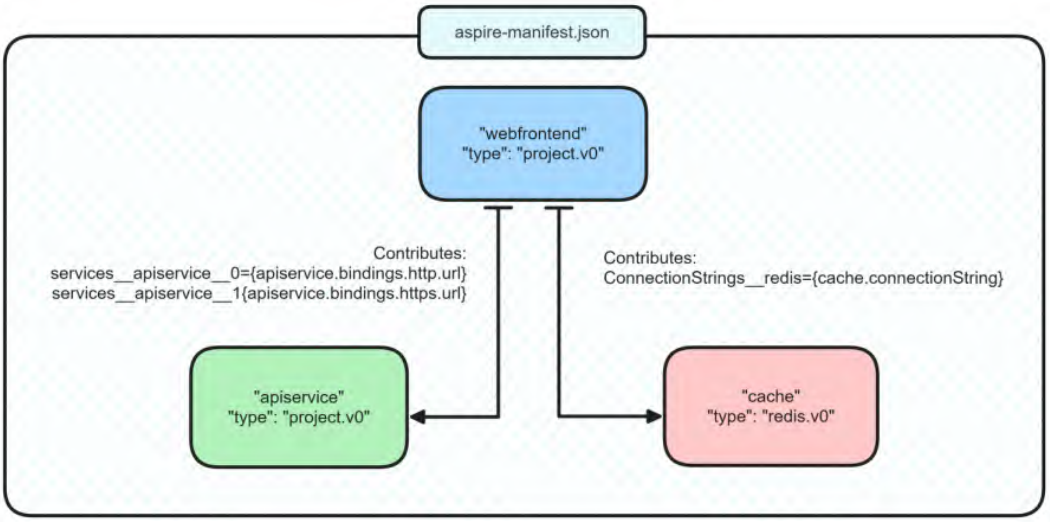


Figure 3.5 – aspire-manifest.json with webfrontend, apiservice, and cache resources. Arrows indicate contributed configuration: services__apiservice__0 and services__apiservice__1 for apiservice bindings, and ConnectionStrings__redis for cache connection string

Aspire handles service discovery implicitly when you reference one microservice from another. When you declare a dependency between resources, Aspire automatically generates the configuration entries needed for those services to find each other. For example, if your frontend project depends on the cache, catalog, and basket microservices, you express these dependencies using references in your AppHost project.

The following code in your AppHost's `Program.cs` demonstrates how to set up these references so that the frontend project receives service discovery information for the cache, catalog, and basket resources:

```
var builder = DistributedApplication.CreateBuilder(args);
var cache = builder.AddRedis("cache");
var catalog = builder.AddProject<Projects.CatalogService>("catalog");
var basket = builder.AddProject<Projects.BasketService>("basket");
var frontend = builder.AddProject<Projects.MyFrontend>("frontend")
    .WithReference(cache)
    .WithReference(catalog)
    .WithReference(basket);
```

By using the `WithReference` method for each dependency, Aspire ensures that the frontend project receives the necessary configuration for service discovery, allowing it to connect to the cache container and the catalog and basket projects without hardcoding addresses or ports.

Common built-in resource APIs in Aspire include: `AddProject`, `AddContainer`, `AddExecutable`, and `AddParameter`. `AddProject` is used to add .NET projects to the distributed application, while `AddContainer` adds a generic container image resource. `AddExecutable` can be used to run executable or binary-based applications, and `AddParameter` allows you to supply parameters or configuration values to services.

In our example, the Catalog and Basket microservices are project resources added with `AddProject`, while Redis is a container resource. Redis references are typically provided through connection string configuration, whereas project resources such as Catalog and Basket expose service discovery information. Aspire injects the appropriate configuration for these references into the frontend project as environment variables.

Redis is a container-backed resource, added through the Redis hosting integration with `AddRedis`. Catalog and Basket microservices are project resources, added with `AddProject`. Redis as a container resource references a connection string, while Catalog and Basket as project resources reference service discovery information. Aspire automatically injects all these references as environment variables into the frontend project.

Aspire handles different types of built-in resources by the way it injects environment variables and references between your projects, containers, and services. Here, you will see the distinction between connection string injection for containers and endpoint injection for project-to-project references, and see how you can explicitly configure container endpoints for service discovery.

When you add a container resource such as Redis, Aspire injects a connection string environment variable into your frontend project. For example, referencing the Redis container results in an environment variable like `ConnectionStrings__cache="localhost:62354"` being injected into the frontend.

Project-to-project references, on the other hand, are handled differently. When your frontend project references other microservices such as Catalog or Basket, Aspire injects service discovery endpoint variables instead of connection strings. These endpoint variables take the form of variables such as `services__apiservice__http__0="http://localhost:5455"`, which represent the HTTP endpoint for the referenced service.

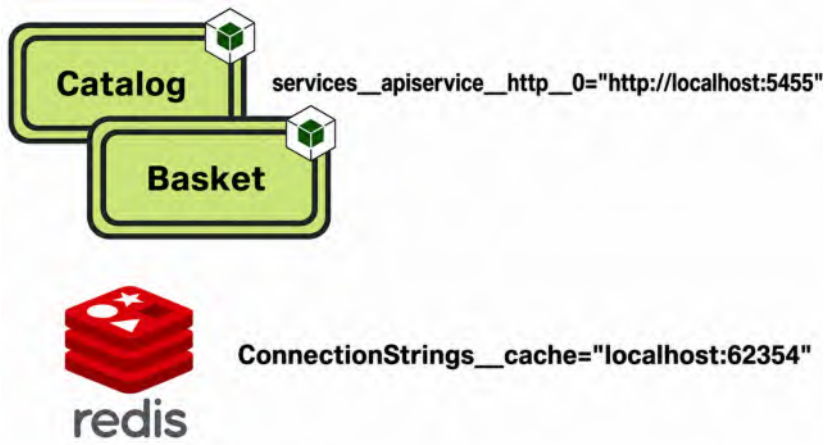


Figure 3.6 – Automatic service discovery and configuration injection in an Aspire distributed application

Here's the code, where the frontend project references the cache, catalog, and basket resources:

```
var builder = DistributedApplication.CreateBuilder(args);
var cache = builder.AddRedis("cache");
var catalog = builder.AddProject<Projects.CatalogService>("catalog");
var basket = builder.AddProject<Projects.BasketService>("basket");
var frontend = builder.AddProject<Projects.MyFrontend>("frontend")
    .WithReference(cache)
    .WithReference(catalog)
    .WithReference(basket);
```

With this setup, Aspire injects the appropriate environment variables for each reference. For the cache container, it injects the following:

```
ConnectionStrings__cache="localhost:62354"
```

For project-to-project references like the API service, it injects:

```
services__apiservice__http__0="http://localhost:5455"
services__apiservice__https__0="https://localhost:7356"
```

This distinction is important in this example: the Redis resource reference injects a connection string, while project-to-project references inject service-discovery endpoints.

For container resources, you can also define named endpoints that will later be referenced by your microservices. The following code demonstrates how to add a custom container with a named HTTP endpoint and then reference that endpoint from another project:

```
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);
```

This approach allows you to explicitly configure and reference container endpoints, making them available for service discovery in your application.

Summary

In this chapter, we explored how Aspire simplifies the development of cloud-native distributed applications by bringing orchestration, integrations, and service discovery together. Although Aspire 13 is a multi-language platform, our examples focused on its .NET support. We began by examining Aspire's key features and benefits and then introduced the core concepts that underpin its approach to distributed application development.

We looked at how the app model and AppHost allow you to define application resources and their relationships, and how resource types and methods such as `AddProject`, `AddContainer`, `WithReference`, and `WaitFor` help coordinate services and their dependencies. We also examined Aspire integrations, distinguishing between **hosting integrations** used to provision or reference resources and **client integrations** used by applications to consume those resources. Finally, we explored service discovery and saw how Aspire provides the configuration required for services to locate and communicate with one another without relying on hardcoded addresses or ports.

The key takeaway is that ***Aspire reduces much of the manual configuration*** involved in coordinating distributed applications while keeping individual services modular and focused on their own responsibilities. With these concepts in place, we are ready to put them into practice.

In the next chapter, we will build our first Aspire application using Visual Studio, bringing together a Blazor frontend, an API service, and a Redis cache. We will run and inspect the application through the **Aspire dashboard**, examine the roles of the different Aspire projects,

and see how orchestration, shared configuration, and service discovery work together in a running distributed application.

4

Building Your First Aspire Application

In the previous chapter, you learned how Aspire brings together orchestration, integrations, service discovery, and distributed application modeling through concepts such as the **AppModel**, **AppHost**, **resource types**, and `IDistributedApplicationBuilder`. With that foundation in place, this chapter moves from understanding the framework to using it in practice.

In this chapter, you will build your first Aspire application in Visual Studio by creating a distributed solution that connects a Blazor frontend, an ASP.NET Core API service, and a Redis cache. You will run the application and use the **Aspire dashboard** to inspect its resources and observe how the different components behave at runtime. From there, you will explore the responsibilities of the `AppHost`, `ServiceDefaults`, `ApiService`, and `Web` projects and examine how these projects work together within the solution.

You will then look more closely at AppHost configuration and orchestration, including how project references, resource dependencies, and startup behavior are defined. You will also see how `ServiceDefaults` provides shared configuration for concerns such as logging, tracing, health checks, resilience, and service discovery. Building on this, you will examine how the frontend locates and communicates with the API service without relying on hardcoded endpoints. Finally, you will compare Aspire orchestration with Docker Compose and walk through how you can move an existing containerized application to an Aspire-based setup.

By the end of this chapter, you will be able to create, run, inspect, and understand the structure of a working Aspire distributed application. These practical skills turn the concepts introduced in the previous chapter into a concrete development workflow and prepare you to work with more complex distributed applications as you progress through the book.

In this chapter, we'll be covering the following topics:

- Creating an Aspire starter application with **Visual Studio**
- Examining the Aspire project structure and roles
- Inspecting `AppHost` configuration and orchestration
- Implementing shared configuration with the `Service Defaults` project
- Managing inter-service communication and service discovery
- Adding Aspire to existing .NET applications

Creating an Aspire starter application with Visual Studio

In this chapter, you are going to build your first Aspire application using Visual Studio. You will create a starter application template, add a Redis cache, develop a simple API, and connect a frontend client to consume that API. You will see how Aspire simplifies connecting multiple services, such as databases, caches, and APIs, so you can focus on the business logic instead of configuration.

The application you are about to create is a basic *distributed system*. It includes a Redis cache, an API service project that exposes endpoints for the frontend, and a frontend that fetches data from the API and displays it to the user. All of these components are containerized and orchestrated by Aspire.

You will use Aspire project templates to develop this distributed architecture. In this quick start, you will complete the following steps:

1. Create a brand-new Aspire starter application.
2. Check Redis as your caching solution.
3. Build an API project within the solution.
4. See how Aspire handles service discovery.
5. Connect the API, frontend, and Redis under one solution, all orchestrated by Aspire.

You can refer to the following high-level diagram to visualize the architecture you are about to implement. The diagram shows the frontend, apiservice, and Redis cache, all managed by Aspire.

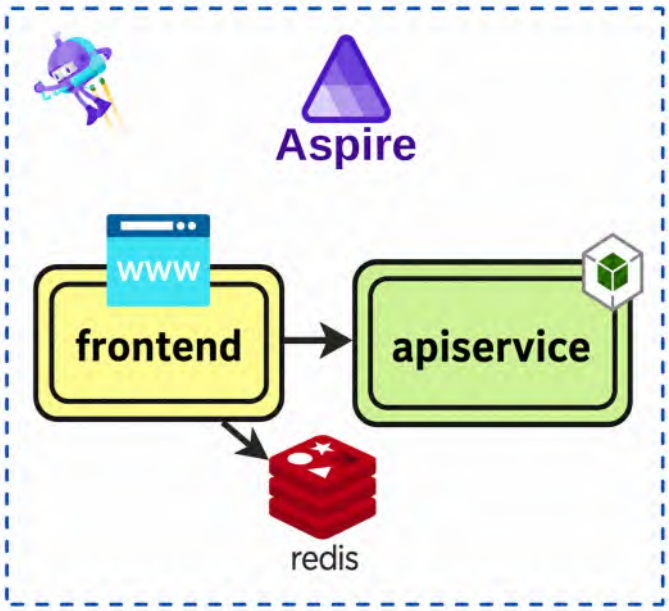


Figure 4.1 – Aspire orchestrating frontend, apiservice, and Redis cache in a distributed application

First, open Visual Studio and select **Create new project**. In the project template window, select the **Aspire Starter App** project template, which includes a Blazor web frontend, web API backend, and Redis caching. Double-click the template, enter `AspireSample` as the **Solution name**, set a **Location**, and select **Next**. Leave the configuration as is: **Framework** as **.NET 10.0**, and check the **Configure for HTTPS** option and the **Use Redis for caching (requires a supported container runtime)** option, which means you will need to start Docker Desktop. Click **Create** to scaffold the new solution with Aspire configurations, giving you a head start on building microservices.

To run and explore your newly created Aspire distributed application, use both Visual Studio and Docker Desktop and follow these steps:

1. Open Visual Studio and navigate to your solution folder.
2. In **Solution Explorer**, right-click the **AspireSample.AppHost** project and select **Set as Startup Project**.

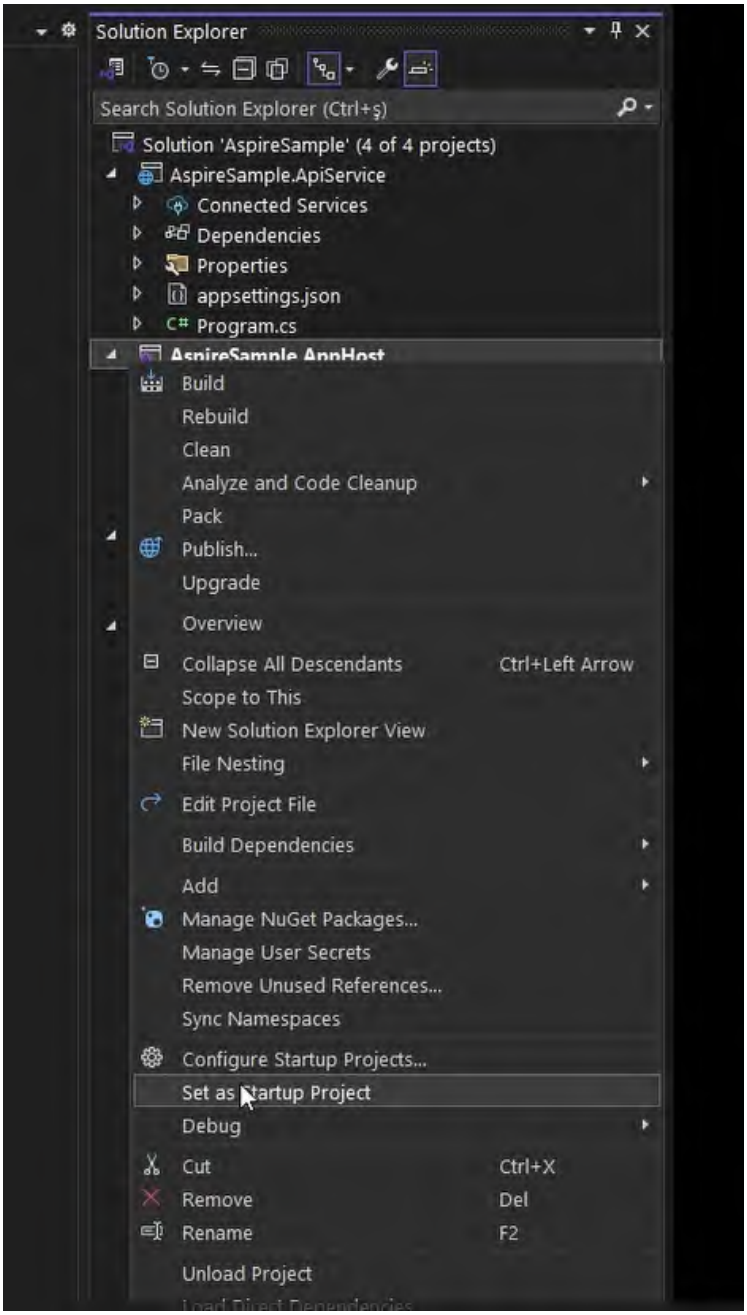


*Figure 4.2 – The **Solution Explorer** window with the* `AppHost` *project selected and the context menu open to set it as the startup project*

3. Before running the solution, ensure that Docker Desktop is running. Aspire requires Docker Desktop to run containerized applications behind the scenes.

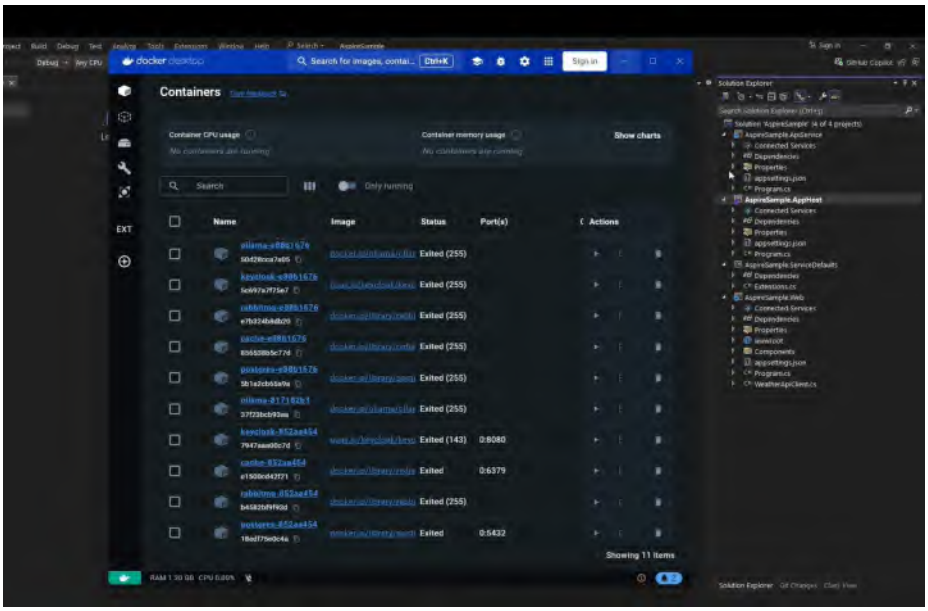


Figure 4.3 – Docker Desktop showing containers for Redis, RabbitMQ, and other services, with their status and ports

4. Once Docker Desktop is running, return to Visual Studio and start the Aspire project by pressing *F5* or selecting the **https** run button for the `AppHost` project.

You will see the project console log indicating that the application has started successfully. This launches your distributed application and automatically opens the **Aspire dashboard** in your browser.

The dashboard provides a visualization of your distributed services, including their status, logs, health checks, and real-time monitoring. The main menu on the left includes **Resources**, **Console**, **Structured**, **Traces**, and **Metrics**:

- **Resources**: This displays all .NET projects with their current state, endpoint addresses, and environment variables.
- **Console**: This allows you to view console outputs for each project. Select a project to see its logs.
- **Structured**: This shows logs in a table format with filtering and search capabilities. You can expand details for each log entry.

- **Traces**: This tracks requests as they flow through the application. For example, you can send a weather request and observe how it moves through the web frontend and API services.
- **Metrics**: This displays various instruments and counters, such as **HTTP** client active requests and connection durations, which you can filter by dimensions to analyze application performance.

When the application starts, the console log and dashboard appear, confirming that the distributed application is running:

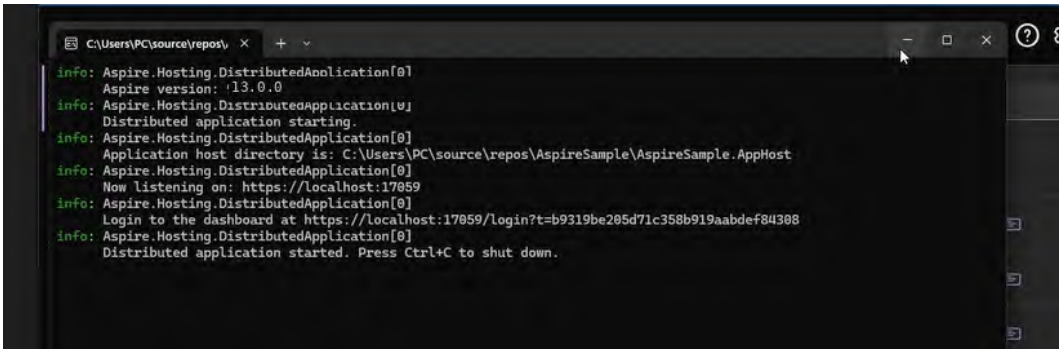


Figure 4.4 – Screenshot of the console log showing the distributed application started and listening on localhost

The dashboard's **Resources** tab lists the running services: the Redis cache container, the **API** service project, and the web frontend project, each with their state, start time, source, and endpoint **URLs**.

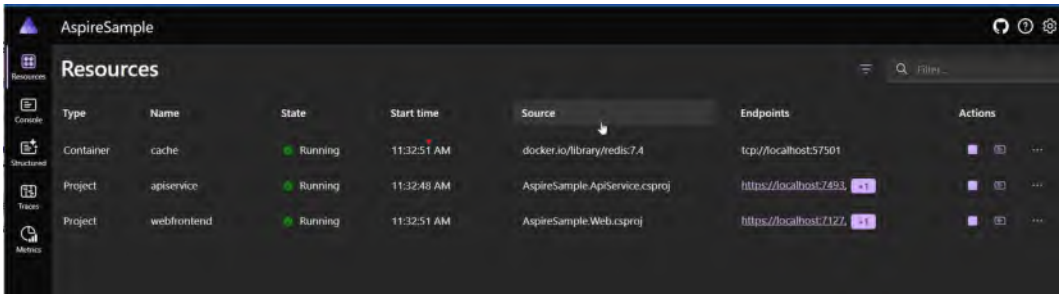


Figure 4.5 – AspireSample dashboard Resources tab listing cache, apiservice, and webfrontend with running status and endpoints

To access your frontend application, follow these steps:

1. In the dashboard, go to the **Resources** tab.

2. Select the **webfrontend** project.
3. Select the endpoint link to open your frontend application in the browser.

On the homepage, you will see a `Hello World!` message. You can also navigate to the weather page, which consumes weather data from the API service, backed by the Redis distributed cache. When you select **Weather**, the API service provides the data via service discovery. If you refresh the page within about 10 seconds, the data remains cached; after that, the output cache expires and the API service provides new data, demonstrating the integration of service discovery, frontend, API, and Redis caching.

You can view the request life cycle in the dashboard's **Traces** tab. Sending a `GET` request to the weather forecast endpoint shows how the web frontend, API service, and cache handle the request. The first request fetches data from the API service and cache, while Redis serves subsequent requests within the cache duration without hitting the API service.

Now that you have created and run your first Aspire distributed application and seen how its services work together, let's take a closer look at the project structure and understand the role each project plays in the solution.

Examining the Aspire project structure and responsibilities

When building cloud-native applications, managing the moving parts of a distributed architecture can quickly become overwhelming. In this section, you will unpack the anatomy of a standard Aspire starter solution to demonstrate how it streamlines local orchestration and service configuration.

The Aspire Sample solution is organized into four main projects, each with a distinct role in the distributed application architecture. Understanding the function of each project helps you manage orchestration, shared configuration, data exposure, and the user interface.

The solution includes `AspireSample.AppHost`, `AspireSample.ServiceDefaults`, `AspireSample.ApiService`, and `AspireSample.Web`. As mentioned, each project serves a specific responsibility: AppHost for orchestration, ServiceDefaults for shared configuration, ApiService for exposing data, and web for the frontend UI.

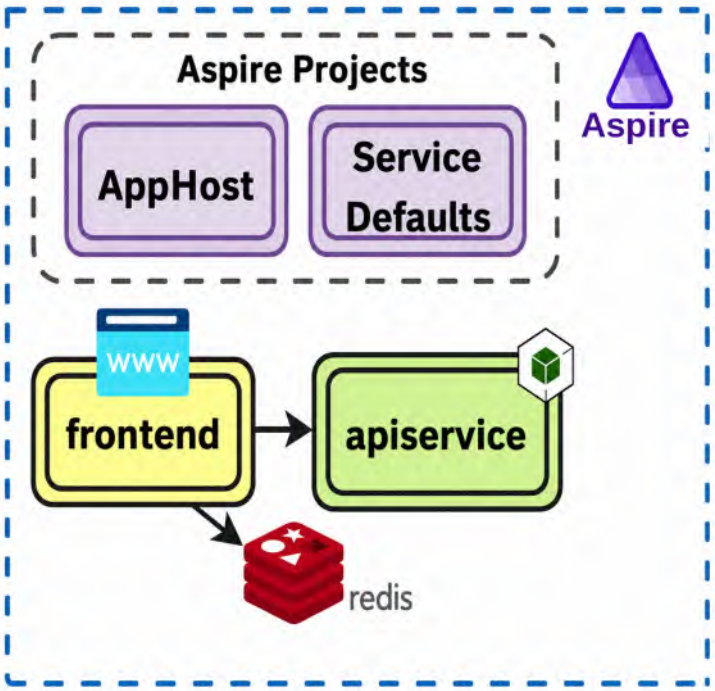


Figure 4.6 – Architecture diagram showing AspireSample.AppHost, AspireSample.ServiceDefaults, AspireSample.ApiService, AspireSample.Web, and their relationships, including frontend, apiservice, and redis components

The preceding figure shows the overall project organization and how these components interact; it highlights the separation between Aspire projects (AppHost and ServiceDefaults) and the distributed application components (frontend, API service, and Redis container).

Let's explore each of these components in detail.

AspireSample.AppHost

`AspireSample.AppHost` acts as the *orchestrator* for the projects and services in your distributed application. It is responsible for connecting and configuring the different components so that they work together as a cohesive solution. You should set `AspireSample.AppHost` as the startup project. The AppHost references both `AspireSample.ApiService` and `AspireSample.Web`, allowing it to orchestrate them as part of the distributed application.

AspireSample.ServiceDefaults

`AspireSample.ServiceDefaults` is a shared project that centralizes common cross-cutting configuration for your distributed application. It configures essential cloud-native capabilities,

including **OpenTelemetry** observability for logging, metrics, and distributed tracing, default health checks, service discovery, and HTTP client resilience features such as retries and circuit breakers. By managing these configurations in one place, you avoid duplicating boilerplate code across multiple projects. Both the API service and web projects reference `AspireSample.ServiceDefaults` and call `AddServiceDefaults()` to apply these shared configurations consistently.

AspireSample.ApiService

`AspireSample.ApiService` is an ASP.NET Core minimal API project that exposes endpoints for the web project to consume, such as fetching weather data. This project depends on the shared ServiceDefaults project for logging, tracing, and service discovery. Typical files in this project include `Program.cs` for ASP.NET Core configuration, `appsettings.json` for configuration values, and minimal route definitions for data retrieval. By referencing ServiceDefaults, the API service inherits best practices for consistency, resiliency, and telemetry across the solution.

AspireSample.Web

`AspireSample.Web` is a Blazor project that contains the UI components, layouts, and pages for your application. The project includes pages such as **Home** and **Weather**, and depends on ServiceDefaults for the same cross-cutting configurations as the API service. The web project communicates with the API service using a strongly typed HTTP client or minimal calls to fetch data, with Aspire's built-in service discovery used to locate the API service at runtime. As the user-facing component, the web project pulls data from the API and benefits from consistent logging and telemetry configured in ServiceDefaults.

With the role of each project established, the next step is to understand how these projects relate to one another and how Aspire manages their dependencies.

Understanding how the Aspire projects work together

The `AspireSample.AppHost` project references both the API service and the web (frontend) project. This allows AppHost to orchestrate these services, ensuring they are started and managed together as part of your distributed application. When you run the AppHost project (for example, by pressing *F5* in Visual Studio), it launches both the API and frontend projects collectively within the Aspire environment.

Both the API service and the frontend project reference the `AspireSample.ServiceDefaults` project. This shared reference enables them to use cross-cutting configurations such as logging, tracing, and health checks from a single location. The API service and frontend do not directly reference each other unless you explicitly add such a dependency. Instead, they use service discovery, as defined in the AppHost project, to locate and communicate with each other at runtime.

By following this structure, each project remains modular and focused on its own responsibilities, while Aspire handles orchestration and shared concerns.

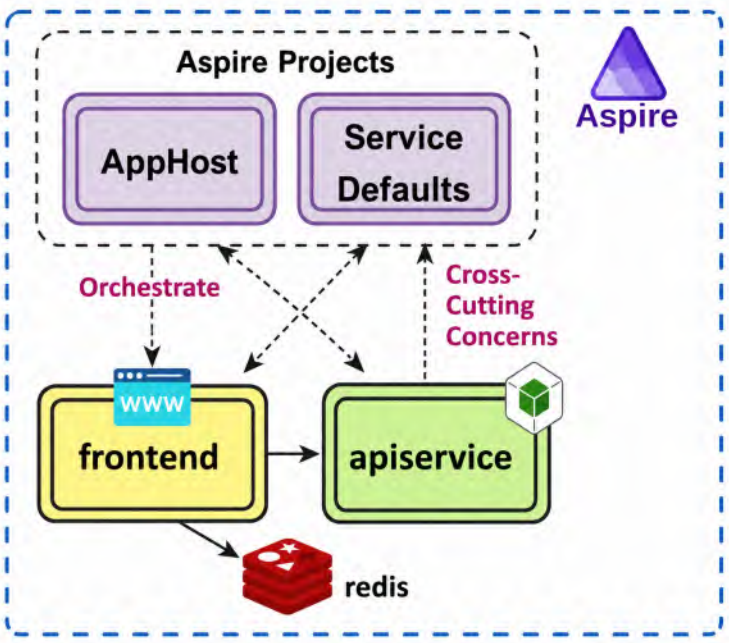


Figure 4.7 – Diagram showing AppHost orchestrating frontend and apiservice, with both referencing ServiceDefaults for cross-cutting concerns

Now that we have established the architectural boundaries of the solution, let's examine the AppHost project to see how its configuration orchestrates these services at runtime.

Inspecting AppHost configuration and orchestration

To examine the Aspire starter project, you start by opening Visual Studio and navigating to the Aspire Sample solution, focusing on the `AspireSample.AppHost` project. This project acts as the **orchestrator**, defining how your microservices, backing services, and integrations are composed into a distributed cloud-native application.

Double-click the `AspireSample.AppHost` project file to view its C# project file. The project file begins by specifying the SDK:

The following code shows the top of the `AspireSample.AppHost` project file, highlighting the custom SDK used for orchestration and container management:

```
<Project Sdk="Aspire.AppHost.Sdk/13.0.0">
```

In Aspire 13, `Aspire.AppHost.Sdk` is declared directly in the `Project` element and identifies the project as an AppHost.

Within the `PropertyGroup`, you'll find the following configuration:

```
<PropertyGroup>
<OutputType>Exe</OutputType>
<TargetFramework>net10.0</TargetFramework>
<ImplicitUsings>enable</ImplicitUsings>
<Nullable>enable</Nullable>
<UserSecretsId>28be9ea5-dc59-481c-96ee-16f46f778ff9</UserSecretsId>
</PropertyGroup>
```

Aspire 13 no longer requires the `IsAspireHost` property; the `Aspire.AppHost.Sdk` declaration identifies the project as the AppHost.

Next, you see the project references:

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

These references indicate that the AppHost project knows about your distributed microservices, the web project and the API project. This project reference enables the AppHost to orchestrate your distributed microservices and containers.

Finally, the file lists package references:

```
<ItemGroup>
<PackageReference Include="Aspire.Hosting.Redis" Version="13.0.0" />
</ItemGroup>
</Project>
```

The Aspire AppHost SDK includes `Aspire.Hosting.AppHost` automatically, while `Aspire.Hosting.Redis` adds Redis hosting support to the AppHost project.

This structure demonstrates how each project in the solution is configured for its specific role, with the AppHost orchestrating the overall distributed application, referencing the API and web projects, and integrating necessary packages for orchestration and caching.

To understand how the AppHost project orchestrates your distributed services, open the `Program.cs` file in the `AspireSample.AppHost` project. Here, you will find the core definitions that model and coordinate the distributed application.

The following code sets up the distributed application builder, adds the Redis cache container, registers the API service project, and registers the Blazor web frontend project:

```
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();
```

This code begins by creating an `IDistributedApplicationBuilder`, which is the foundation for modeling your distributed application. You then add a Redis container resource named `"cache"` using `builder.AddRedis`. Aspire handles the setup of this container behind the scenes.

Next, you register your minimal API project as `"apiservice"` using `builder.AddProject`, which enables **service discovery** for this service. You also register your Blazor frontend application as `"webfrontend"` and configure it with external HTTP endpoints so you can browse the frontend.

The `.WithReference(cache)` and `.WaitFor(cache)` methods tie the web frontend to the Redis cache, ensuring the cache is started before the frontend attempts to connect. Similarly, `.WithReference(apiService)` and `.WaitFor(apiService)` ensure the frontend references and waits for the API service to be running first.

Finally, `builder.Build().Run()` builds and runs the distributed application using Aspire and the AppHost project.

When you set `AspireSample.AppHost` as the startup project and run the distributed application, Aspire orchestrates the startup sequence for all services according to the dependencies defined with the `.WaitFor` and `.WithReference` methods. The application starts the cache (Redis container) first, followed by the API service, and finally the web frontend.

The running services in the **Aspire dashboard** provide a real-time view of the distributed application's resources and their states.

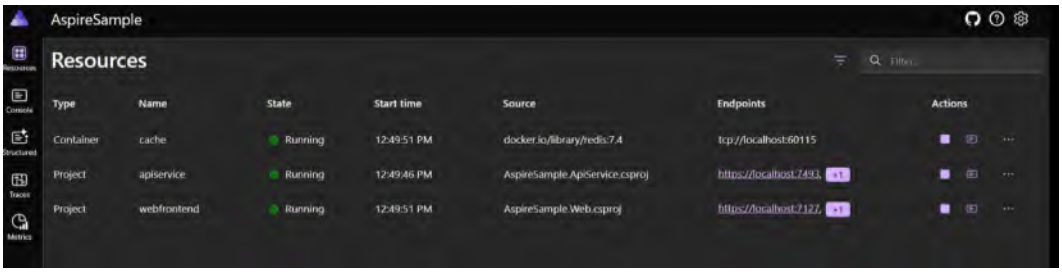


Figure 4.8 – Aspire dashboard showing the running services

The dashboard interface allows you to inspect each running service, including their endpoints and **environment variables**. For example, when you open the web frontend **project details** and scroll down, you can see environment variables such as `ASPNETCORE_ENVIRONMENT`, the Redis cache connection string, and connection information for dependent services such as the API service. Aspire injects these environment variables automatically to enable service discovery and connectivity between projects.

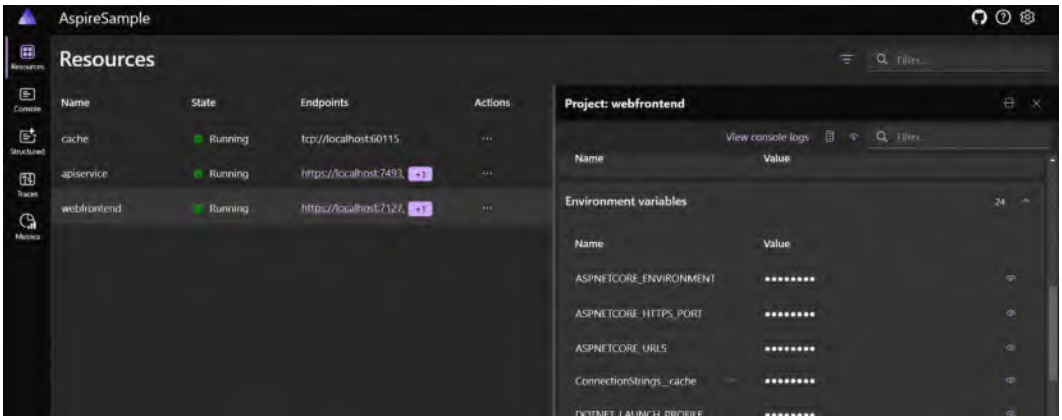


Figure 4.9 – Screenshot showing the environment variables of the webfrontend project

This orchestration means you do not need to manage separate Docker Compose files or manually launch each service. The `AppHost` project centralizes orchestration, handling the startup and configuration of all containers and microservice projects.

You can also verify that the Redis container is running in Docker Desktop, started by Aspire without requiring any manual configuration for the Redis container itself. Aspire handles all container management and orchestration behind the scenes.

Aspire injects the necessary environment variables for service discovery, allowing each project to locate and communicate with the others without relying on static IP addresses, ports, or additional configuration.

With the AppHost project handling orchestration and service connectivity, the next step is to see how Aspire applies shared configuration consistently across the services in the solution. This is where the `ServiceDefaults` project comes in.

Implementing shared configuration with the Service Defaults project

In this section, we're going to examine the `Service Defaults` project, and you will see how this shared project enforces consistent configuration across all distributed microservices. As mentioned previously, this project centralizes cross-cutting concerns such as resilience, service discovery, logging, and OpenTelemetry.

First, let's open the `AspireSample.ServiceDefaults` project file to see its configuration. You will see the key attribute `IsAspireSharedProject` is set to true, indicating that this is a shared project rather than a standalone application:

```
<Project Sdk="Microsoft.NET.Sdk">
<PropertyGroup>
<TargetFramework>net10.0</TargetFramework>
<ImplicitUsings>enable</ImplicitUsings>
<Nullable>enable</Nullable>
<IsAspireSharedProject>true</IsAspireSharedProject>
</PropertyGroup>
<ItemGroup>
<FrameworkReference Include="Microsoft.AspNetCore.App" />
<PackageReference Include="Microsoft.Extensions.Http.Resilience"
Version="10.0.0" />
<PackageReference Include="Microsoft.Extensions.ServiceDiscovery"
Version="10.0.0" />
<PackageReference Include="OpenTelemetry.Exporter.OpenTelemetryProtocol"
Version="1.12.0" />
<PackageReference Include="OpenTelemetry.Extensions.Hosting" Version="1.12.0" />
<PackageReference Include="OpenTelemetry.Instrumentation.AspNetCore"
```

```
Version="1.12.0" />
<PackageReference Include="OpenTelemetry.Instrumentation.Http" Version="1.12.0" />
<PackageReference Include="OpenTelemetry.Instrumentation.Runtime"
Version="1.12.0" />
</ItemGroup>
</Project>
```

These package references unify how your microservices handle logging, tracing, and fallback strategies, ensuring that resilience, service discovery, OpenTelemetry, and other cross-cutting concerns are consistently applied.

To see how Service Defaults integrates with other microservices, open the `AspireSample.ApiService` project file. There, you will find a project reference to `ServiceDefaults`:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
<PropertyGroup>
<TargetFramework>net10.0</TargetFramework>
<ImplicitUsings>enable</ImplicitUsings>
<Nullable>enable</Nullable>
</PropertyGroup>
<ItemGroup>
<ProjectReference Include="..
\AspireSample.ServiceDefaults\AspireSample.ServiceDefaults.csproj" />
</ItemGroup>
<ItemGroup>
<PackageReference Include="Microsoft.AspNetCore.OpenApi" Version="10.0.0" />
</ItemGroup>
</Project>
```

Similarly, the `AspireSample.Web` frontend project also references `ServiceDefaults`:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
<PropertyGroup>
<TargetFramework>net10.0</TargetFramework>
<ImplicitUsings>enable</ImplicitUsings>
<Nullable>enable</Nullable>
</PropertyGroup>
<ItemGroup>
<ProjectReference Include="..
\AspireSample.ServiceDefaults\AspireSample.ServiceDefaults.csproj" />
```

```
</ItemGroup>
<ItemGroup>
<PackageReference Include="Aspire.StackExchange.Redis.OutputCaching"
Version="13.0.0" />
</ItemGroup>
</Project>
```

This setup ensures that both the API and frontend projects share identical resilience and telemetry settings by referencing `ServiceDefaults`.

Within the API service's `Program.cs` file, we use the `builder.AddServiceDefaults()` extension method to register all necessary cloud-native telemetry, logs, traces, metrics, service discovery, and resilience features such as retry policies and circuit breakers for HTTP calls:

```
var builder = WebApplication.CreateBuilder(args);
// Add service defaults & Aspire client integrations.
builder.AddServiceDefaults();
```

The same extension method is included in the web project's `Program.cs` file:

```
var builder = WebApplication.CreateBuilder(args);
// Add service defaults & Aspire client integrations.
builder.AddServiceDefaults();
builder.AddRedisOutputCache("cache");
```

If you navigate to the definition of `AddServiceDefaults`, you find it in the `Extensions.cs` file under the ServiceDefaults project. The method configures OpenTelemetry, health checks, service discovery, and HTTP client defaults for resilience and service discovery:

```
public static TBuilder AddServiceDefaults<TBuilder>(this TBuilder builder) where
TBuilder : IHostApplicationBuilder
{
    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 allowed schemes for service discovery.
    // builder.Services.Configure<ServiceDiscoveryOptions>(options =>
    // {
    //     options.AllowedSchemes = ["https"];
    // });
    return builder;
}
```

You also find additional extension methods in this class, such as configuring OpenTelemetry and mapping default health endpoints:

```
public static TBuilder AddDefaultHealthChecks<TBuilder>(this TBuilder builder)
{
    builder.Services.AddHealthChecks()
    // Add a default liveness check to ensure app is responsive
    .AddCheck("self", () => HealthCheckResult.Healthy(), ["live"]);
    return builder;
}
public static WebApplication MapDefaultEndpoints(this WebApplication app)
{
    // 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.
    if (app.Environment.IsDevelopment())
    {
        // ...
    }
    return app;
}
```

By centralizing these configurations in `ServiceDefaults`, you avoid duplicating boilerplate code for logging, tracing, metrics, and distributed tracing. When you add a new microservice to your solution and reference `ServiceDefaults`, Aspire automatically integrates these settings at runtime, ensuring consistent cross-cutting concern configurations across your distributed environment.

With shared configuration now centralized through `ServiceDefaults`, the next step is to see how the services in the application locate and communicate with one another. We will now

look at how Aspire uses service discovery to enable this inter-service communication without relying on hardcoded endpoints.

Managing inter-service communication and service discovery

In this section, you will see how Aspire manages inter-service communication in a distributed application, focusing on how the `AppHost` project ties services together using service discovery names, and how your Blazor frontend calls the API project using a typed HTTP client that automatically locates the correct endpoint.

First, let's recall the service discovery mechanism in Aspire. When you add a reference from your frontend project to your API service in the `AppHost`, Aspire injects the API service's endpoint address into the frontend application's environment variables. This enables your web frontend to consume and call the API service using Aspire's service discovery, without hardcoding URLs.

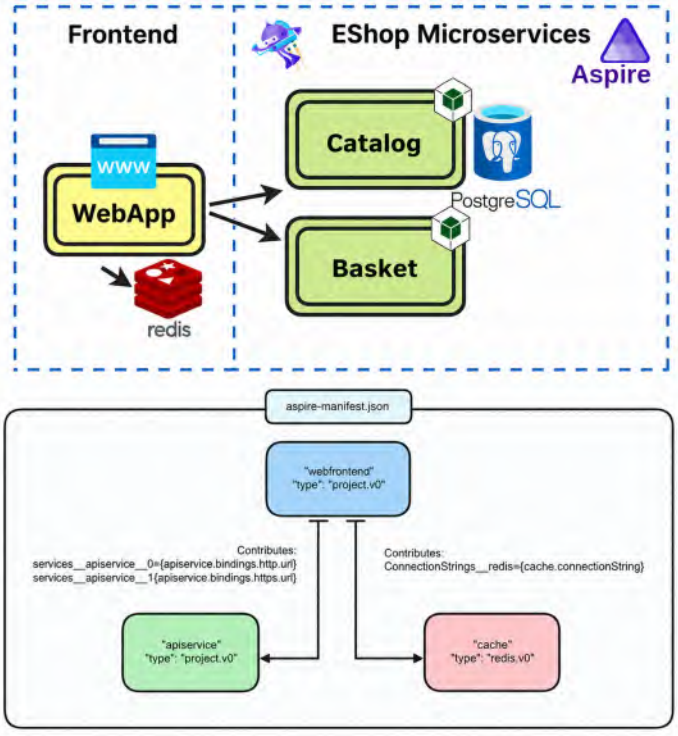


Figure 4.10 – Diagram showing injected connection strings and service endpoints for cache and API services, with Aspire manifest relationships

The following code in the `AppHost`'s `Program.cs` demonstrates how you define these project references and dependencies:

```
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();
```

By adding `.WithReference(apiService)` to the web frontend project, Aspire injects the API service's endpoint into the web frontend's environment variables, making it available for service discovery.

To consume the API service from your Blazor frontend, you register a typed HTTP client in the web project's `Program.cs`. The following code registers the `WeatherApiClient` and configures its base address using the service discovery name:

```
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");
});
```

The important part here is the base address: `https+http://apiservice`. This tells Aspire to use HTTPS if available; otherwise fallback to HTTP, and to resolve the endpoint using the service discovery name `apiservice` as defined in the AppHost project. At runtime, Aspire resolves this to the actual endpoint, such as `https://localhost:7281`, so you do not need to manually manage URLs or port numbers.

The `WeatherApiClient` class in your web project is a *strongly-typed client* focused on retrieving weather data from the API service. Its implementation is shown below:

```
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);
```

This client uses the injected `HttpClient` to call the `/weatherforecast` endpoint on the API service, fetches an array of weather forecast records, and returns them. You inject and use this API client in your Blazor weather page component, retrieving the forecast data in the `OnInitializedAsync` method and displaying it in a table.

The AppHost's `Program.cs` defines the distributed application and project references:

```
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();
```

The `WeatherApiClient` class in the web project handles fetching weather data:

```
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);
```

The web project's `Program.cs` registers the typed HTTP client and configures its base address for service discovery:

```
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");
});
```

By using Aspire's service discovery, your frontend and API projects remain modular and decoupled, with endpoint resolution handled automatically at runtime.

Now that we have seen how Aspire manages service discovery and communication between projects, we can look at how these capabilities can be introduced into an existing .NET application. The next section shows how to move from a Docker Compose-based setup to Aspire orchestration.

Adding Aspire to existing .NET applications

If you have been using Docker Compose to orchestrate your microservice projects, you are likely familiar with the process of manually adding `Dockerfiles` to each service and writing Docker Compose YAML files to manage the overall architecture. Before Aspire, this was the standard approach for handling distributed applications that included components like a Web API, a Blazor frontend, and dependencies such as a Redis container.

The following diagram contrasts the traditional Docker Compose setup with the Aspire approach, highlighting how Aspire introduces `AppHost` and `ServiceDefaults` projects to orchestrate and configure your distributed application.

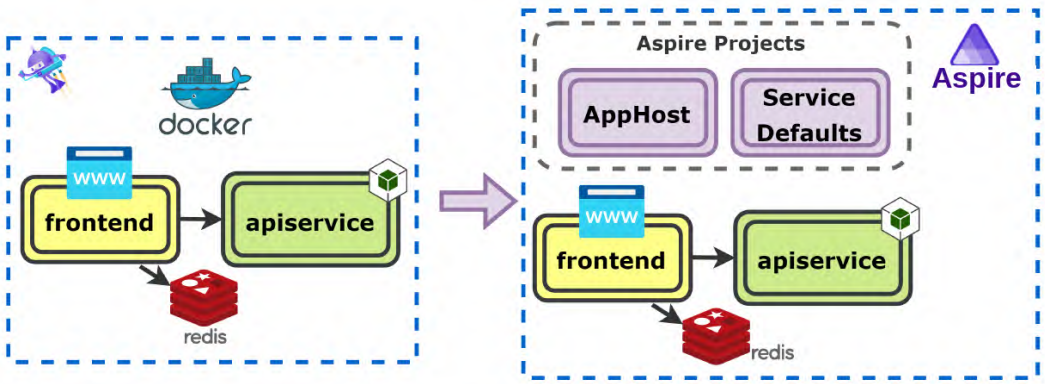


Figure 4.11 – Comparison diagram of Docker Compose architecture (frontend, apiservice, redis) versus Aspire Projects (AppHost, ServiceDefaults, frontend, apiservice, redis)

With this comparison in mind, let's now see how an existing Docker Compose-based application can be migrated to Aspire. To begin, open Visual Studio and create a new solution.

To begin migrating your existing application to Aspire, open Visual Studio and create a new solution. In this example, you start with two microservices: an API service built with ASP.NET Web API, and a web frontend built with Blazor.

The **Solution Explorer** in Visual Studio displays both the `apiservice` and `frontend` projects, each with their respective files and folders.

So, now the typical existing setup will be handled with the Dockerfiles and Docker Compose files. For that purpose, you right-click each microservice and go to **Add | Docker Support**, which creates a Dockerfile for each microservice with all the necessary configurations.

The following code is the `Dockerfile` generated for the API service, including port numbers and other settings:

```
# See https://aka.ms/customizecontainer to learn how to customize your debug
container and how VS...
# This stage is used when running from VS in fast mode (Default for Debug
configuration)
FROM mcr.microsoft.com/dotnet/aspnet:10.0 AS base
USER $APP_UID
WORKDIR /app
EXPOSE 8080
EXPOSE 8081
# This stage is used to build the service project
FROM mcr.microsoft.com/dotnet/sdk:10.0 AS build
ARG BUILD_CONFIGURATION=Release
WORKDIR /src
COPY ["apiservice/apiservice.csproj", "apiservice/"]
RUN dotnet restore "./apiservice/apiservice.csproj"
COPY . .
WORKDIR "/src/apiservice"
RUN dotnet build "./apiservice.csproj" -c $BUILD_CONFIGURATION -o /app/build
# This stage is used to publish the service project to be copied to the final
stage
FROM build AS publish
ARG BUILD_CONFIGURATION=Release
RUN dotnet publish "./apiservice.csproj" -c $BUILD_CONFIGURATION -o /app/publish
# This stage is used in production or when running from VS in regular mode
FROM base AS final
WORKDIR /app
```

```
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "apiservice.dll"]
```

Here is the `Dockerfile` for the frontend microservice:

```
# See https://aka.ms/customizecontainer to learn how to customize your debug
container and how VS...
# This stage is used when running from VS in fast mode (Default for Debug
configuration)
FROM mcr.microsoft.com/dotnet/aspnet:10.0 AS base
USER $APP_UID
WORKDIR /app
EXPOSE 8080
EXPOSE 8081
# This stage is used to build the service project
FROM mcr.microsoft.com/dotnet/sdk:10.0 AS build
ARG BUILD_CONFIGURATION=Release
WORKDIR /src
COPY ["frontend/frontend.csproj", "frontend/"]
RUN dotnet restore "./frontend/frontend.csproj"
COPY . .
WORKDIR "/src/frontend"
RUN dotnet build "./frontend.csproj" -c $BUILD_CONFIGURATION -o /app/build
# This stage is used to publish the service project to be copied to the final
stage
FROM build AS publish
ARG BUILD_CONFIGURATION=Release
RUN dotnet publish "./frontend.csproj" -c $BUILD_CONFIGURATION -o /app/publish /
p:UseAppHost=false
# This stage is used in production or when running from VS in regular mode
(Default when not using...
FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "frontend.dll"]
```

After generating Dockerfiles for both microservices, you need to orchestrate them in a Docker environment using Docker Compose **YAML** files. To do this, right-click `apiservice`, go to **Add | Container Orchestrator Support**, and select **Docker Compose**. This creates a `docker-compose.yml` file and adds the API service to it.

The following is the initial Docker Compose YAML file for the API service:

```
services:
apiservice:
image: ${DOCKER_REGISTRY-}apiservice
build:
context: .
dockerfile: apiservice/Dockerfile
```

Repeat the same steps for the frontend to add it to the orchestrator. After adding the frontend, the Docker Compose YAML file now includes both microservices:

```
services:
apiservice:
image: ${DOCKER_REGISTRY-}apiservice
build:
context: .
dockerfile: apiservice/Dockerfile
frontend:
image: ${DOCKER_REGISTRY-}frontend
build:
context: .
dockerfile: frontend/Dockerfile
distributedcache:
image: redis
You will also need a Redis container for distributed caching. Add the Redis image
to the Docker Compose file as shown in the preceding snippet.
```

Next, you must configure the microservices and containers to set the port numbers in the Docker Compose orchestrator. Expand the Docker Compose YAML file in **Solution Explorer** and select the Docker Compose override YAML file, which includes all the port numbers, environment variables, and configurations.

The following Docker Compose override YAML file defines the environment variables, port mappings, and volume mounts for the API service and frontend, and configures the Redis-based distributed cache:

```
services:
apiservice:
environment:
- ASPNETCORE_ENVIRONMENT=Development
```

```
- ASPNETCORE_HTTP_PORTS=8080
- ASPNETCORE_HTTPS_PORTS=8081
ports:
- "8080"
- "8081"
volumes:
- ${APPDATA}/Microsoft/UserSecrets:/home/app/.microsoft/usersecrets:ro
- ${APPDATA}/ASP.NET/https:/home/app/.aspnet/https:ro
frontend:
environment:
- ASPNETCORE_ENVIRONMENT=Development
- ASPNETCORE_HTTP_PORTS=8080
- ASPNETCORE_HTTPS_PORTS=8081
ports:
- "8080"
- "8081"
volumes:
- ${APPDATA}/Microsoft/UserSecrets:/home/app/.microsoft/usersecrets:ro
- ${APPDATA}/ASP.NET/https:/home/app/.aspnet/https:ro
distributedcache:
container_name: distributedcache
restart: always
ports:
- "6379:6379"
```

Managing all this configuration, including indentation, port numbers, and other settings, can be challenging as your solution grows. Even with only three microservices, the configuration fills almost the entire page.

Aspire replaces this overhead with a more integrated approach, allowing your .NET solution itself to orchestrate containers, references, and environment variables in C# code. To shift from Docker Compose to Aspire, you need to add a Aspire orchestrator to your project to handle all these configurations.

To migrate from Docker Compose to Aspire, you need to do the following:

1. Delete the Docker Compose YAML files and all related files inside each microservice.
2. Right-click the API service (`apiservice`) and add **Aspire Orchestrator Support**.
3. Select your solution and location, then select the **OK** button.

This scaffolds the code by adding the Aspire projects: the orchestrator project (`AppHost`) and the shared configuration project (`ServiceDefaults`).

You can also add these projects manually by right-clicking and adding a new project, then searching for Aspire to find the `AppHost` and `ServiceDefaults` project templates.

Once the Aspire projects are added, the AppHost project becomes the orchestrator, managing containers, environment variables, and service discovery. It acts as the startup project and references the other projects. The ServiceDefaults project provides shared configuration for resilience, service discovery, and telemetry across all distributed microservices.

As a result, you no longer need to rely on Docker Compose YAML files for orchestration. Instead, you can run the `AppHost` project, which orchestrates and spins up the containers and references using the Aspire distributed architecture.

The progress dialog in Visual Studio confirms that the API service is being added to the Aspire orchestration.

To examine the project structure after migrating from Docker Compose to Aspire orchestration, you start by looking at the `AppHost` project. When you open this project, you see that it currently only includes the API service. In the `Program.cs` file, only the API service is added, reflecting that the Aspire orchestration was initially set up from the API service.

If you want to include the frontend microservice in your distributed architecture, you can do one of two options:

- Right-click the `frontend` project and add **Aspire orchestrator support**.
- Manually add the `frontend` project reference in the orchestration code.

Choosing the first option is recommended, as it automatically adds the required NuGet packages and project references. When you add **Aspire orchestrator support** to the `frontend` project, Visual Studio detects that an Aspire AppHost project already exists and prompts you to update it to include the frontend in its orchestration.

You will see a dialog confirming that the orchestrator project will be updated to add the frontend project for orchestration.

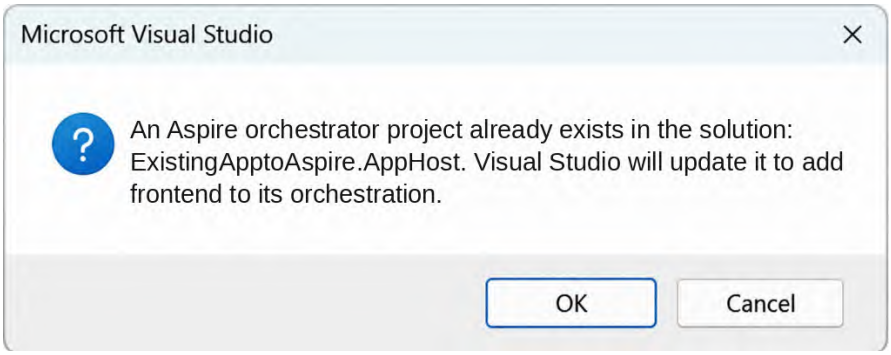


Figure 4.12 – Visual Studio prompting to add the frontend project to the existing Aspire orchestration

After confirming, Visual Studio adds a line of code to the orchestration and updates the `AppHost` project to reference the frontend application. Additionally, the frontend project now references the `ServiceDefaults` project, ensuring that all cross-cutting concerns are applied to the frontend microservice as well.

The following XML shows the `AppHost` project file after adding the frontend reference:

```
<Project Sdk="Aspire.AppHost.Sdk/13.0.0">
<PropertyGroup>
<OutputType>Exe</OutputType>
<TargetFramework>net10.0</TargetFramework>
<ImplicitUsings>enable</ImplicitUsings>
<Nullable>enable</Nullable>
<UserSecretsId>0ae76263-00b5-49b4-9083-71ba2c3866fc</UserSecretsId>
</PropertyGroup>
<ItemGroup>
<ProjectReference Include="..\apiservice\apiservice.csproj" />
<ProjectReference Include="..\frontend\frontend.csproj" />
</ItemGroup>
</Project>
```

You can also see that the frontend project file now includes a reference to the `ServiceDefaults` project:

```
<ProjectReference Include="..
\ExistingApptoAspire.ServiceDefaults\ExistingApptoAspire.ServiceDefaults.csproj" /
>
```

In the orchestration code, the frontend project is added alongside the API service:

```
var builder = DistributedApplication.CreateBuilder(args);
builder.AddProject<Projects.apiservice>("apiservice");
builder.AddProject<Projects.frontend>("frontend");
builder.Build().Run();
```

To add Redis as a backing service, you use the NuGet package manager. Right-click on **Dependencies** of the AppHost project and select the **Add Aspire package** option. This allows you to filter and install Aspire hosting packages, such as `Aspire.Hosting.Redis`, which provides Redis support for Aspire.

You select the Redis package and install it into the AppHost project. This enables the use of the Redis extension method in your orchestration code.

The following code assigns a variable for the Redis cache and configures the dependencies between the distributed microservices:

```
var builder = DistributedApplication.CreateBuilder(args);
var cache = builder.AddRedis("cache");
var apiService = builder.AddProject<Projects.apiservice>("apiservice");
builder.AddProject<Projects.frontend>("frontend")
    .WithExternalHttpEndpoints()
    .WithReference(cache)
    .WaitFor(cache)
    .WithReference(apiService)
    .WaitFor(apiService);
builder.Build().Run();
```

This configuration injects the connection settings for the cache and the service discovery endpoint for the API service into the frontend project.

Once you set the `AppHost` project as the startup project and run it, the entire distributed architecture is activated in the Aspire environment. All resources will be running successfully in the distributed environment.

Summary

In this chapter, you built and explored your first Aspire distributed application using Visual Studio project templates and Docker Desktop. You created a solution that brought together a Blazor frontend, an API service, and a Redis cache. You then used the **Aspire dashboard** to inspect the running resources and understand how the different projects fit together. You examined the responsibilities of `AppHost`, `ServiceDefaults`, `ApiService`, and `web`, explored how AppHost manages orchestration and startup dependencies, and saw how ServiceDefaults centralizes shared concerns such as logging, tracing, resilience, health checks, and service discovery.

You also looked at how services communicate without hardcoded endpoints and compared Aspire orchestration with Docker Compose, including how an existing containerized application can be migrated to Aspire. The key takeaway is that Aspire centralizes orchestration, shared configuration, and service discovery while allowing individual projects to remain modular and focused on their own responsibilities. With this foundation, you are ready to move beyond the starter application and apply these concepts to a more realistic distributed system.

In the next chapter, you will build an eShop microservices architecture using PostgreSQL, Redis, RabbitMQ, and a Blazor frontend, bringing together the orchestration and service-integration techniques you have learned so far.

5

Building an eShop Distributed Microservices Architecture with Aspire

In the previous chapter, you built and explored your first Aspire application, gaining practical experience with `AppHost`, `ServiceDefaults`, **service discovery**, shared configuration, and orchestration. With those fundamentals in place, you are now ready to apply them to a more realistic distributed application: an eShop built around independently managed **microservices** and supporting infrastructure.

In this chapter, we will lay the foundation for the eShop distributed microservices architecture. You will begin by planning the overall architecture and identifying how the Catalog and Basket microservices will interact with supporting services such as **PostgreSQL**, **Redis**, and **RabbitMQ**. You will then create a blank Visual Studio solution and organize it into dedicated folders for Aspire projects, microservices, and the frontend application. Within this structure, you will set up the AppHost project for orchestration and the ServiceDefaults project for shared **cross-cutting concerns**.

You will also define the development plan that will guide the eShop implementation through the chapters ahead. This roadmap covers building the Catalog and Basket microservices, enabling **synchronous and asynchronous communication**, securing services with **Keycloak** and **JWT** bearer tokens, developing the **Blazor** frontend, deploying the application to **Azure Container Apps**, and eventually introducing generative AI capabilities such as customer support Q&A and semantic product search.

By the end of this chapter, you will have a clearly organized solution, the core Aspire projects in place, and a development roadmap for the complete eShop system. Establishing this structure before implementing the individual microservices is important because it gives each component a clear responsibility and provides a consistent foundation for orchestration, communication, security, deployment, and future AI capabilities. In the chapters that follow, you will build on this foundation to implement each part of the architecture step by step.

In this chapter, we'll be covering the following topics:

- Planning and setting up the eShop distributed architecture
- Setting up the eShop solution structure
- Setting up the AppHost and ServiceDefaults projects
- Planning the eShop distributed microservices development

Planning and setting up the eShop distributed architecture

In this chapter, we are going to start building the eShop distributed microservices architecture with Aspire. We'll start by creating a new solution, organizing the solution into specific folders, and setting up the initial projects. The architecture begins with the development of the **Catalog microservice**, which uses a PostgreSQL database for product storage and management and publishes product price change integration events to RabbitMQ.

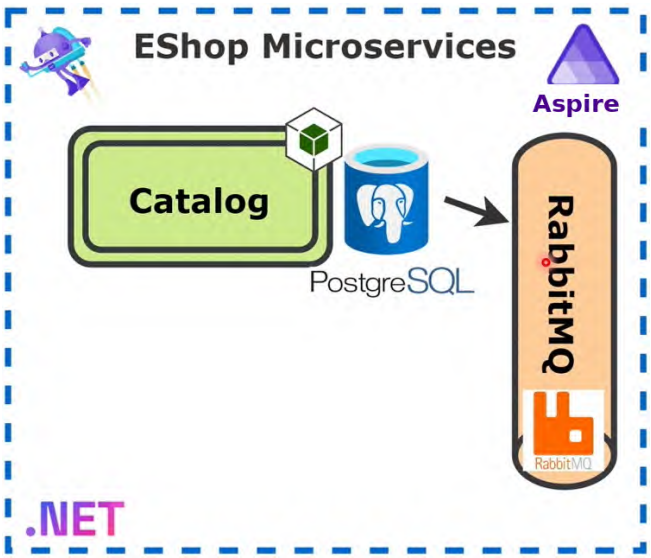


Figure 5.1 – Catalog microservice with PostgreSQL and RabbitMQ integration

Next, we will develop the **Basket microservice**, which uses Redis as a distributed cache for user baskets. When you add items to a basket, the Basket microservice synchronously calls the Catalog microservice and also consumes product price change integration events from RabbitMQ to keep basket prices up to date.

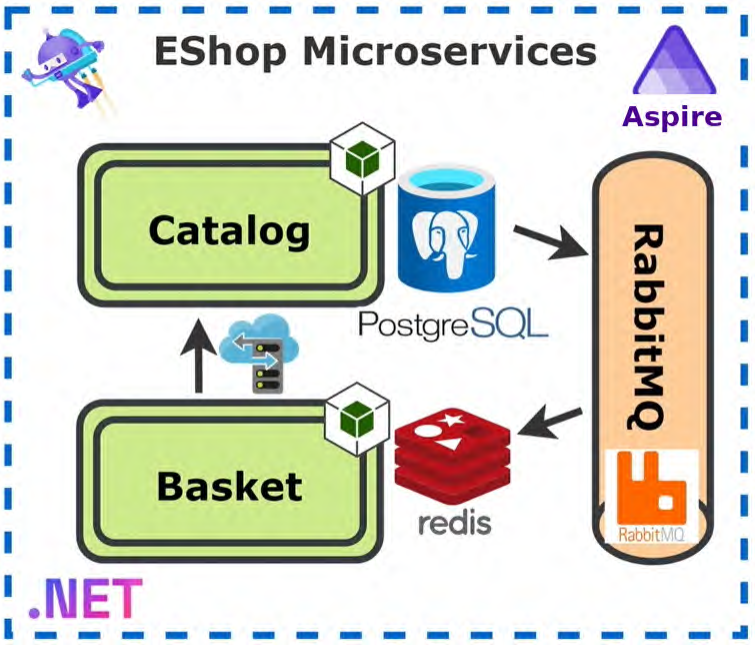


Figure 5.2 – Catalog and Basket microservices with PostgreSQL, Redis, and RabbitMQ

You will then secure the Basket endpoints with Keycloak, a robust, open-source identity-management server. You will use **JWT (JSON Web Token)** bearer tokens to verify API requests; the tokens will authorize those requests securely without maintaining session state. After that, you will develop the web application frontend, which consumes internal microservices using API clients and Aspire service discovery.

You will implement a **Products** page in the Blazor frontend to display data retrieved from the internal microservices and add output caching to improve the page's performance.

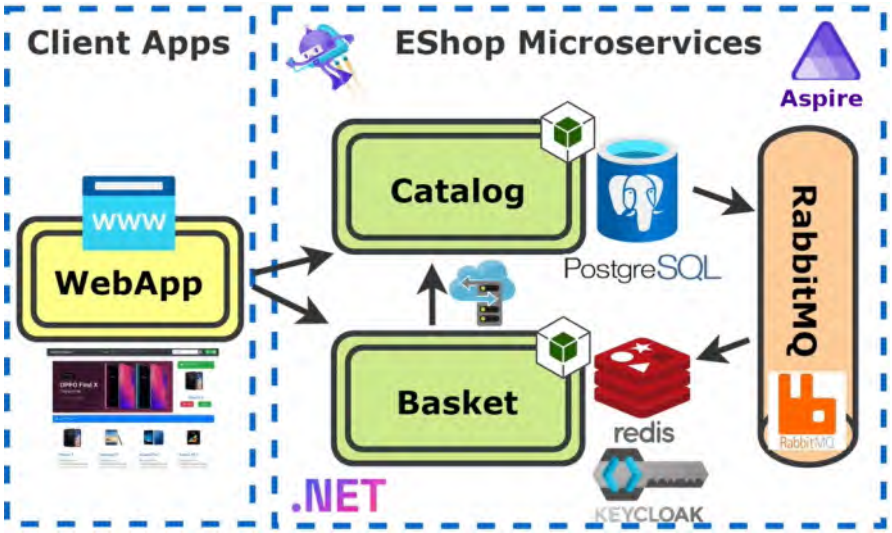


Figure 5.3 – Full eShop architecture with WebApp frontend, secured Basket, and service integrations

Finally, you will deploy the application to Azure Container Apps by containerizing the microservices and using the `azd up` and `azd down` commands with Aspire. You will also explore generative AI development in .NET using the `Microsoft.Extensions.AI` library, implementing customer support Q&A and semantic product search with a vector store.

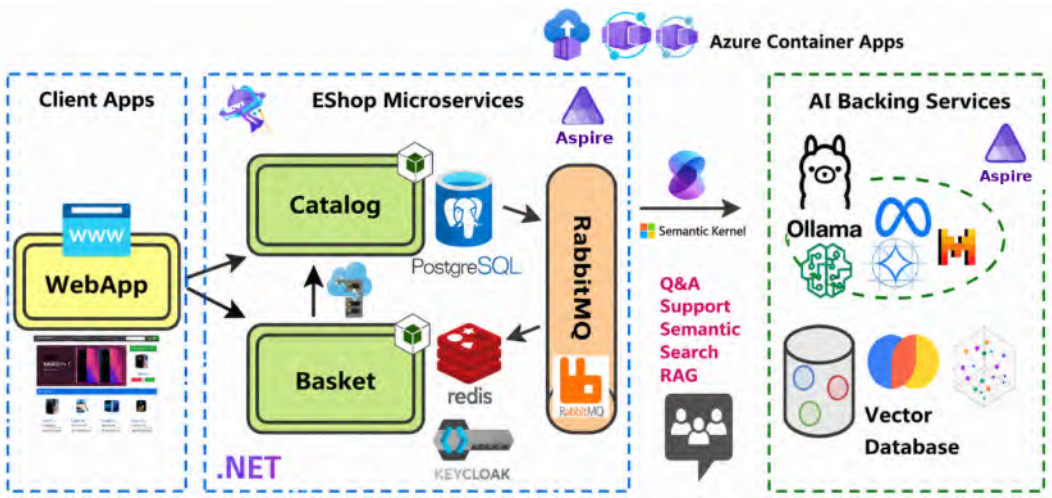


Figure 5.4 – High-level architecture of the eShop solution, showing the client application, core microservices, messaging and data components, and AI backing services orchestrated with Aspire and deployed on Azure Container Apps

With Aspire, you will orchestrate all these services into a cohesive system that can scale in the cloud. Specifically, you will:

1. Create a new blank solution named `eshop-distributed` to act as a container for all projects and distributed containers.
2. Organize your Visual Studio solution structure by creating three main folders: `aspire`, `services`, and `frontend`.
    - The `aspire` folder will include Aspire projects such as AppHost and ServiceDefaults.
    - The `services` folder will include individual microservices such as Catalog and Basket.
    - The `frontend` folder will include the Blazor web application.
3. Develop your Aspire projects in the `aspire` folder, including the AppHost project for orchestrating your distributed application and ServiceDefaults for centralized cross-cutting concerns for each microservice.
4. Make a step-by-step development plan for building your eShop distributed microservices architecture.

Setting up the eShop solution structure

To begin building your distributed eShop application, you first need to create a new blank solution and then organize its folder structure in Visual Studio. This section guides you through setting up the foundational solution and folders that will contain your Aspire projects, microservices, and frontend application.

Start by opening Visual Studio and selecting **Create a new project**. In the project template selection, choose **Blank Solution** so you can develop everything from scratch and add projects one by one.

After selecting the blank solution template, select the **Next** button. Enter the **solution name** as `eshop-distributed` and choose your preferred **location** for the solution files. Select the **Create** button to generate your root solution. This solution will serve as the container for your entire distributed application.

Once you create the solution, you will see it in **Solution Explorer** with no projects yet added.

Next, organize your solution by creating three main folders. Right-click the solution (`eshop-distributed`) in **Solution Explorer**, select **Add** and then **New Solution Folder**. Name the first folder `aspire`. This folder will contain your Aspire projects, such as `AppHost` and `ServiceDefaults`, which bootstrap the distributed application locally.

After creating the `aspire` folder, repeat the process to add a second folder named `services`. This folder will hold your individual microservices, such as `Catalog` and `Basket`, each corresponding to a **bounded context**.

Finally, add a third folder named `frontend`. This folder will contain your frontend web application, which consumes the internal microservices to perform eShop use cases.

The **Solution Explorer** will now display the three main folders: `aspire`, `services`, and `frontend`.

With this structure in place, you are ready to begin adding and developing your Aspire projects within the appropriate folders.

Setting up the AppHost and ServiceDefaults projects

You will now create the core Aspire projects for orchestrating your distributed microservices solution and providing its cross-cutting concerns.

To begin, open your solution in Visual Studio, navigate to the `aspire` folder, right-click, and select **Add | New Project**. In the project template search box, filter by `aspire` to locate the available Aspire project templates.

Select the **Aspire App Host** project template and select the **Next** button. On the configuration screen, set the **project name** to `AppHost` and confirm the **location** is within your solution directory. Select **Next**. Then, choose **.NET 10.0** as the **framework** and select **Create** to generate the project.

After creation, you will see the AppHost project under the `aspire` folder in **Solution Explorer**.

Double-click the AppHost project file to inspect its contents. In Aspire 13, the AppHost **SDK** is declared directly in the `Project` element and targets `.NET 10`:

```
<Project Sdk="Aspire.AppHost.Sdk/13.0.0">
<PropertyGroup>
<OutputType>Exe</OutputType>
<TargetFramework>net10.0</TargetFramework>
<ImplicitUsings>enable</ImplicitUsings>
```

```
<Nullable>enable</Nullable>
<UserSecretsId>e75a5779-1e13-440d-abe5-c82f601c2792</UserSecretsId>
</PropertyGroup>
</Project>
```

Open the `Program.cs` file in the AppHost project to start planning your distributed services. Initially, it contains the following code to create and run the distributed application builder:

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

You will need to add references to each microservice and declare cloud-native backing services such as Redis and PostgreSQL in this file. For now, add a comment to indicate where these resources will be declared:

```
var builder = DistributedApplication.CreateBuilder(args);
// add projects and cloud-native backing services here
builder.Build().Run();
```

Next, add the `ServiceDefaults` project to centralize cross-cutting concerns. Right-click the `aspire` folder, select **Add | New Project**, and again filter by `aspire` in the templates.

Select the **Aspire Service Defaults** template, select **Next**, and name the project `ServiceDefaults`. Select **Next**. Choose **.NET 10.0** as the **framework**, and select **Create**. The `ServiceDefaults` project will now appear under the `aspire` folder.

Each microservice will reference this project to ensure consistency across the solution. If you open the project file, you will see it is defined as an Aspire shared project and includes several cross-cutting NuGet packages. The included **extension method**, `AddServiceDefaults`, will be called in each microservice's `Program.cs` to register these concerns and map service defaults endpoints.

With the Aspire projects in place, you are ready to proceed with building your microservices under the `services` folder.

Planning the eShop distributed microservices development

In this section, you will map out the development plan for the eShop distributed microservices architecture. You will first review the overall architecture and then break the implementation into a sequence of steps covering the core microservices, service communication, security, frontend development, cloud deployment, and generative AI features.

The architecture includes client applications, core eShop microservices, and AI backing services. The Catalog and Basket microservices use PostgreSQL and Redis and communicate via RabbitMQ, while Aspire orchestrates them. The web frontend consumes these microservices, and AI features like Q&A and semantic search use Ollama and a vector database.

The development plan consists of the following steps:

1. **Setting up core microservices**: Develop the Catalog and Basket microservices, integrate Catalog with PostgreSQL and Basket with Redis, implement **CRUD** operations, and expose **HTTP** endpoints.
2. **Enabling microservice communication with Aspire**: Implement synchronous communication between Catalog and Basket using Aspire for service discovery and asynchronous messaging using RabbitMQ for events such as product price changes.
3. **Implementing authentication and security**: Secure the Basket microservice with Keycloak by creating the `eshop` realm, registering the `eshop-client`, adding a test user, and configuring JWT bearer authentication and authorization to protect the Basket endpoints..
4. **Developing the frontend application**: Build a Blazor web application frontend with a Razor page for product listing and basket functionality, using HTTP clients to call the Catalog and Basket microservices.
5. **Deploying to Azure Container Apps**: Containerize each microservice, push images to Azure Container Registry, and deploy the application to Azure Container Apps using the `azd up` and `azd down` commands.
6. **Adding generative AI features**: Integrate Ollama and a vector database to provide a **Q&A** chatbot and implement semantic search using **LLMs** and vector storage.

Now, let's follow this plan and start developing the core microservices in the next chapter.

Summary

In this chapter, we laid the foundation for the eShop distributed microservices architecture using Aspire. We began by defining the overall architecture and understanding how the Catalog and Basket microservices will work with supporting services such as PostgreSQL, Redis, and RabbitMQ, alongside the Blazor frontend and planned AI capabilities.

We then created and organized the Visual Studio solution into dedicated folders for Aspire projects, microservices, and the frontend application. Within this structure, we set up the AppHost project to orchestrate the distributed application and the ServiceDefaults project to provide shared cross-cutting concerns. Finally, we established a step-by-step development plan covering microservice implementation and communication, authentication and security, frontend development, deployment to Azure Container Apps, and generative AI features.

With the solution structure, Aspire projects, and development roadmap now in place, we are ready to begin implementing the individual microservices. In the next chapter, we will start with the Catalog microservice, where we will model the product domain, connect the service to PostgreSQL using Entity Framework Core and Aspire integrations, organize the application into distinct layers, and expose CRUD operations through minimal **API** endpoints.

6

Developing the Catalog Microservice with PostgreSQL and Aspire

In this chapter, you will develop the Catalog microservice with a PostgreSQL database, orchestrated using Aspire. You will begin by analyzing the Catalog domain, focusing on its domain models, use cases, REST APIs, and database requirements, before moving into the technical design, including the architecture, patterns, libraries, and folder organization that will shape the implementation.

With the design in place, you will build the microservice step by step. You will start with the `Product` domain entity and organize the project into `Models`, `Data`, `Services`, and `Endpoints` folders. You will then configure PostgreSQL as a backing service through Aspire, integrate **Entity Framework Core**, develop `ProductDbContext`, and set up database migrations and seeding. Finally, you will implement the Catalog application use cases and business service layer and expose REST API endpoints for **create, read, update, and delete** (**CRUD**) operations.

By the end of the chapter, you will have a maintainable Catalog microservice with clear separation of concerns, orchestrated infrastructure, and reliable PostgreSQL integration through Aspire.

In this chapter, we'll be covering the following topics:

- Domain analysis of the Catalog microservice
- Technical analysis of the Catalog microservice
- Setting up the Catalog microservice with Aspire orchestration
- Configuring PostgreSQL as the Catalog backing service
- Adding PostgreSQL to the AppHost
- Configuring the PostgreSQL client integration
- Creating the Catalog data layer
- Creating the Entity Framework Core migration
- Implementing the Product service layer
- Creating the Product endpoints
- Registering the Product endpoints
- Testing the endpoints

Domain analysis of the Catalog microservice

The primary **domain model** for the Catalog microservice is the `Product` entity. A domain model represents the core business concepts that an application needs to understand and manage; here, that means a product and the information associated with it. The `Product` entity therefore includes attributes such as `id`, `name`, `description`, `price`, and `image URL`.

Alongside the entity itself, the domain should account for important business events. For example, when a product's price changes, the microservice can raise a product price change **integration event** so that other parts of the system can respond accordingly.

The main use cases for the Catalog microservice are standard **CRUD** operations: listing products, retrieving a product by ID, creating a product, updating it, and deleting it. The REST API endpoints are designed around these operations, providing the interface through which other parts of the application interact with the Catalog microservice.

The following figure shows the Catalog microservice with its PostgreSQL backing service and its place within the overall eShop microservices architecture.

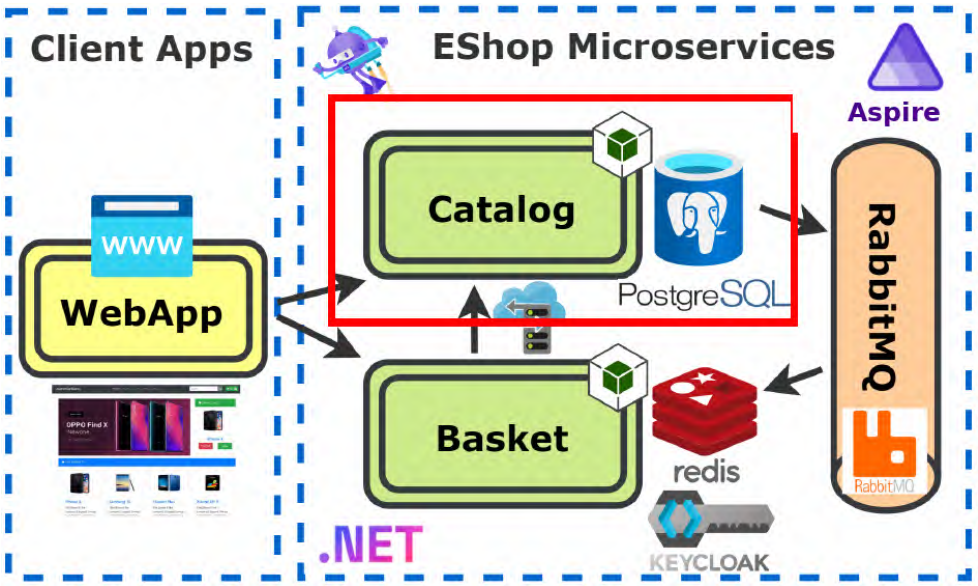


Figure 6.1 – Architecture diagram showing Catalog and Basket microservices, PostgreSQL, Redis, RabbitMQ, and WebApp frontend

With this broader context in place, we can look more closely at the domain elements that underpin the Catalog microservice. The domain elements include the product model, its use cases, the REST API endpoints through which those use cases are exposed, and the PostgreSQL database in which product data is stored.

The development process begins with the `Product` domain entity:

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; } = default!;
    public string Description { get; set; } = default!;
    public decimal Price { get; set; }
    public string ImageUrl { get; set; } = default!;
}
```

This class captures the core information required to represent a product in the Catalog. In addition to the entity itself, the domain should account for a product price change event, which can be used to trigger an integration event when a product's price is updated.

To work with these products, the Catalog microservice exposes REST API endpoints for the following CRUD operations:

1. Use `HTTP GET` on `/products` to list all products.
2. Use `HTTP GET` on `/products/{id}` to fetch a specific product.
3. Use `HTTP POST` on `/products` to create a new product.
4. Use `HTTP PUT` on `/products/{id}` to update a product.
5. Use `HTTP DELETE` on `/products/{id}` to remove a product.

The Catalog microservice stores its product data in a PostgreSQL relational database. You will use the Entity Framework Core **code-first approach** to map the domain model to the database schema and manage schema changes through migrations. The code-first approach keeps the database structure aligned with the application's domain entities as the microservice evolves.

With the domain model, use cases, API surface, and database requirements established, you can now turn to the technical analysis of the Catalog microservice, where you will examine the architecture, patterns, libraries, and project organization used to implement these requirements.

Technical analysis of the Catalog microservice

In this section, you will examine the application architecture style and the patterns and libraries you will use throughout development. You can now decide how to structure the Catalog microservice in code.

N-layer architecture pattern

For this implementation, you will use a classical **n-layer architecture**, an architectural approach that separates an application into logical layers, with each layer responsible for a specific set of concerns. The *n* indicates that the architecture can contain any number of layers; in our Catalog microservice, we will use four: domain, data access, business logic, and presentation.

- The **domain layer** stores and handles domain entities and domain events, representing the microservice's main domain.
- The **data access layer** manages interaction with the database or other storage systems, performing CRUD operations.
- The **business logic layer** processes the application's core business logic and rules, implementing business rules and acting as a bridge between the presentation and data access layers.
- The **presentation layer** displays data to users and captures user input as the front-end.

Splitting your code into these layers makes it easier to maintain and scale your microservices.

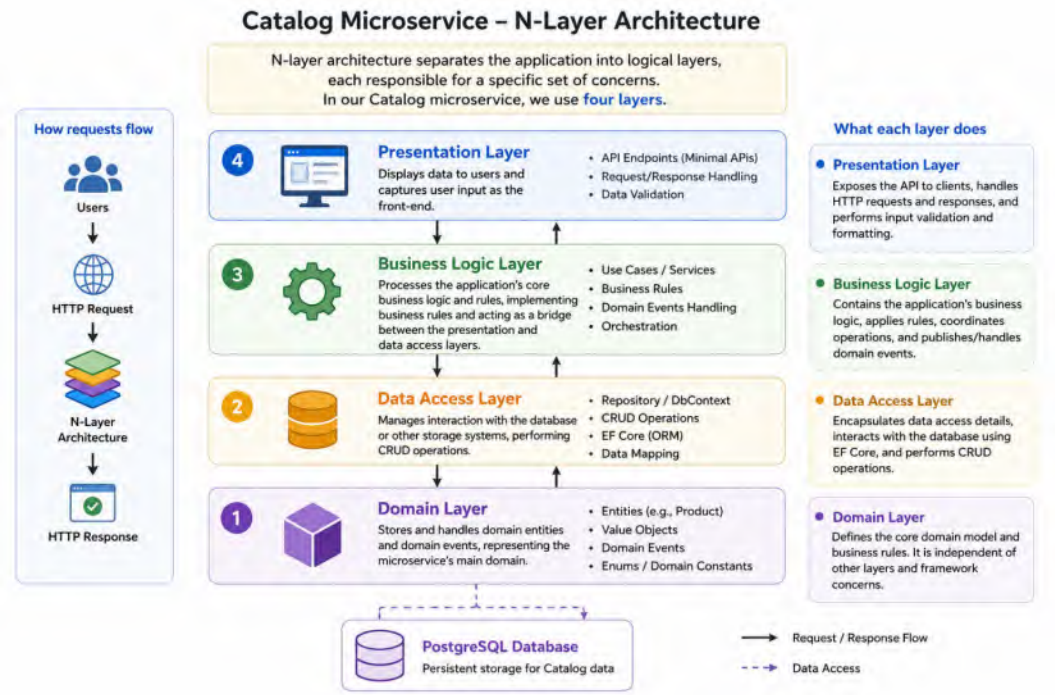


Figure 6.2 – Four-layer architecture of the Catalog microservice

Fundamental patterns and principles

We'll follow the "keep it simple" principle. We will keep the implementation deliberately simple rather than introducing additional architectural patterns unnecessarily. We'll rely on a small set of fundamental ASP.NET Core and Entity Framework Core patterns, together with Aspire integrations and orchestration:

- **Dependency injection in ASP.NET Core**, which supplies dependencies to classes rather than requiring them to create or hard-code those dependencies.
- **Minimal APIs and routing in ASP.NET Core**, which provide a lightweight way to define and handle HTTP endpoints.
- **The Entity Framework Core ORM pattern**, which abstracts database interactions and lets you work with database objects using high-level code.

To support this architecture, we will use Aspire hosting packages to orchestrate the distributed application and provision PostgreSQL, together with `Aspire.Npgsql.EntityFrameworkCore.PostgreSQL` as the client integration that connects the Catalog microservice to PostgreSQL through Entity Framework Core.

With the architecture and supporting technologies in place, the next step is to organize the implementation so that each responsibility has a clear home in the codebase.

Project folder structure

The Catalog microservice project will be organized into four main folders: `Models`, `Data`, `Services`, and `Endpoints`, each corresponding to a specific responsibility within the microservice architecture:

- The `Models` folder contains your domain entities and data Transfer objects (**DTOs**). For the Catalog microservice, you will create product domain models here.
- The `Data` folder includes the database context object, which in this case is `ProductDbContext`. This folder also contains database migrations, allowing you to manage schema changes over time.
- The `Services` folder is where you implement business logic. You will create the product service and any related business logic classes in this folder.
- The `Endpoints` folder contains the minimal API endpoints that interact with the service layer. Here, you will define `Product` endpoints and expose CRUD **HTTP** endpoints for your Catalog microservice.

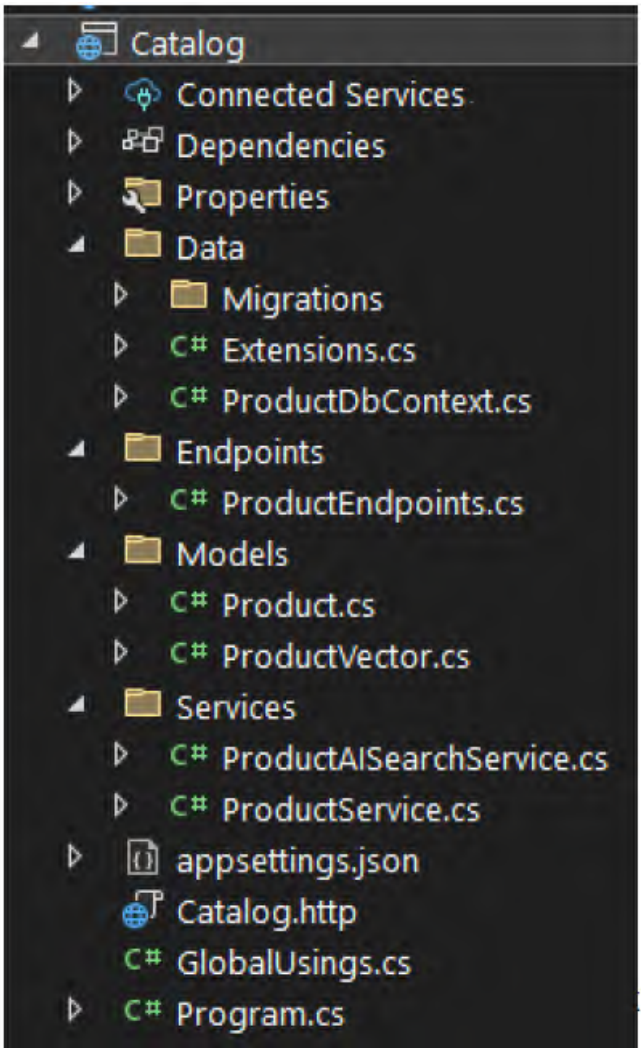


Figure 6.3 – Solution explorer view showing the Catalog microservice organized into Models, Data, Services, and Endpoints folders

The folder structure gives each layer a clear place in the codebase. With that organization defined, we can now create the Catalog microservice and connect it to the Aspire orchestration environment.

Setting up the Catalog microservice with Aspire orchestration

In this section, you will create the Catalog microservice as an ASP.NET Core Web API and add it to the Aspire orchestration environment. You will then examine how the generated project is connected to `ServiceDefaults` and `AppHost`, run the distributed application, and verify that the Catalog service is available through the **Aspire dashboard**.

Creating the Catalog Web API project

Let's start by creating the Catalog microservice project:

1. In Visual Studio, go to your solution (`eshop-distributed`), right select the `services` folder to open the context menu.
2. Select **Add** | **New Project** from the menu.
3. From the **Add a new project** dialog, choose the **ASP.NET Core Web API** template to represent your microservice.
4. Select the **Next** button.
5. In the **Configure your new project** window, enter the project name as `Catalog`.
6. Select **Next** to continue to the **Additional information** screen.

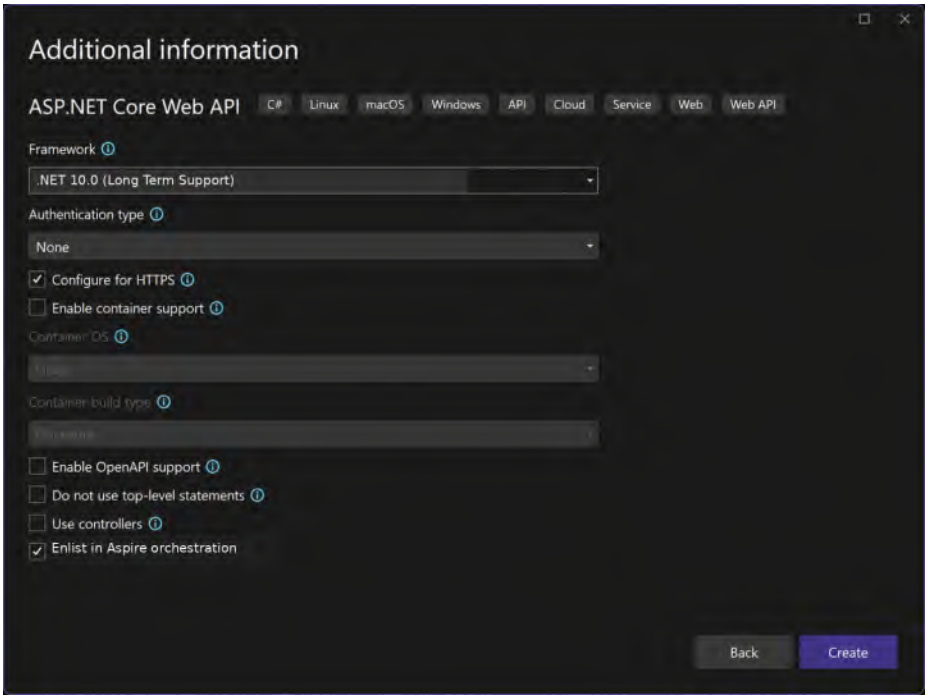


Figure 6.4 – Additional information screen

7. In the **Additional information** window, leave most configurations at their defaults:
    - **Framework**: **.NET 10.0 (Long Term Support)**
    - **Authentication type**: **None**
    - **Configure for HTTPS**: **Checked**
    - **Enable container support**: **Unchecked**

    Make sure to check **Enlist in Aspire orchestration** at the bottom.
8. Select the **Create** button to generate the Catalog microservice project under the `services` folder

Once the project is created, open the Catalog project file by double clicking on it. You will see that Visual Studio has added a reference to the `ServiceDefaults` project.

```
<ItemGroup>
<ProjectReference Include="..\ServiceDefaults\ServiceDefaults.csproj" />
</ItemGroup>
```

This reference gives the Catalog microservice access to the common configuration used across the distributed application, including service discovery and shared operational defaults.

Next, go to the `Program.cs` file of the Catalog microservice. You will see the typical ASP.NET Web API template, exposing a `"/weatherforecast"` endpoint with `MapGet`.

```
var builder = WebApplication.CreateBuilder(args);
builder.AddServiceDefaults();
// Add services to the container.
var app = builder.Build();
app.MapDefaultEndpoints();
// Configure the HTTP request pipeline.
app.UseHttpsRedirection();
var summaries = new[]
{
    "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;
});
app.Run();
record WeatherForecast(DateOnly Date, int TemperatureC, string? Summary);
```

Because the project is integrated with Aspire, you will also see the following calls:

```
builder.AddServiceDefaults();
```

and:

```
app.MapDefaultEndpoints();
```

`AddServiceDefaults` applies the common service configuration, while `MapDefaultEndpoints` registers the standard endpoints provided through ServiceDefaults.

These two **extension methods** from Aspire `ServiceDefaults` ensure that the Catalog microservice has consistent features.

Registering the Catalog project in AppHost

After understanding the `Program.cs` file of your Catalog microservice, you can move to the orchestrator part, which is the application host under the `Aspire` folder. Because you selected **Enlist in Aspire orchestration** when creating the Catalog project, the project is automatically added under your `AppHost` project for the Catalog microservice. Also, double-clicking the `AppHost` project shows that it automatically gets a reference from the Catalog microservice. All of this is handled by Aspire when enlisting your Catalog microservice during configuration.

If you open the `Program.cs` file in `AppHost`, you will see that your Catalog microservice is registered here with the name `"catalog"` so that it is discoverable by other microservices in the Aspire distributed architecture.

The following code in `AppHost`'s `Program.cs` demonstrates how the Catalog microservice project is added to the distributed application builder:

```
var builder = DistributedApplication.CreateBuilder(args);
// add projects and cloud-native backing services here
builder.AddProject<Projects.Catalog>("catalog");
// add projects and cloud-native backing services here
builder.Build().Run();
```

The `AppHost` project file includes a project reference to the Catalog microservice:

```
<ItemGroup>
<ProjectReference Include="..\Catalog\Catalog.csproj" />
</ItemGroup>
```

Let's now run the `AppHost` project. Running the AppHost project will start the distributed application, including one microservice, which is the Catalog microservice. Running the project opens your Aspire dashboard as well. In the **Resources** view, you can see the `catalog` project running. You can also see the endpoint for the service.



Figure 6.5 – Resources view showing the Catalog microservice endpoint

If you open the service endpoint directly, the browser displays a *page not found* response because no endpoint is mapped to the root URL. To verify that the Catalog microservice is running correctly, append `/weatherforecast` to the service URL and navigate to that endpoint. The returned response confirms that the microservice is running successfully.

With the Catalog microservice now running under Aspire orchestration, you are ready to replace the default Web API functionality with the domain model and application logic needed for the Catalog service.

Creating the Product domain model

As mentioned earlier in the chapter, to keep the Catalog microservice organized and maintainable, we use the classical *n*-layer architecture, and the first layer we will implement is the **domain layer**, which contains the domain entities and events that represent the core business concepts of the microservice.

Let's begin:

1. Open Visual Studio and navigate to the Catalog microservice project under the `services` folder.
2. Right-click the Catalog project and add a new folder (**Add | New Folder**) named `Models`.
3. In the `Models` folder, you will create your first domain model, which is the `Product` domain model. Add a new class file named `Product.cs`.
4. Define the `Product` domain entity class with the following members: `Integer Id`, `String Name`, `String Description`, `Decimal Price`, and `String ImageUrl`.

The following code defines the `Product` entity in the domain layer:

```
namespace Catalog.Models;
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; } = default!;
    public string Description { get; set; } = default!;
```

```
    public decimal Price { get; set; }
    public string ImageUrl { get; set; } = default!;
}
```

After adding the `Product` domain entity to the `Models` folder, you have completed the first layer of the architecture. The next step is to build the data layer. Before doing so, however, you need to configure the Catalog microservice's **backing service**: a **PostgreSQL** relational database. Because the data layer relies on Entity Framework Core for persistence, establishing this PostgreSQL connection in Aspire first ensures that your `ProductDbContext` has a target database to bind to.

Configuring PostgreSQL as the Catalog backing service

In this section, you will configure PostgreSQL as the backing service for the Catalog microservice using Aspire. Before setting it up, it is useful to understand the two types of Aspire integrations involved.

Aspire provides two main types of integrations for activating and connecting backing services from your microservices: **hosting integrations** and **client integrations**.

Hosting integrations are added to the AppHost project and define resources such as databases, caches, and message queues within your distributed application that Aspire provisions or references. Client integrations, on the other hand, are added to consumer projects, such as the Catalog Web API, and configure those projects to connect to and use the resources defined by the AppHost.

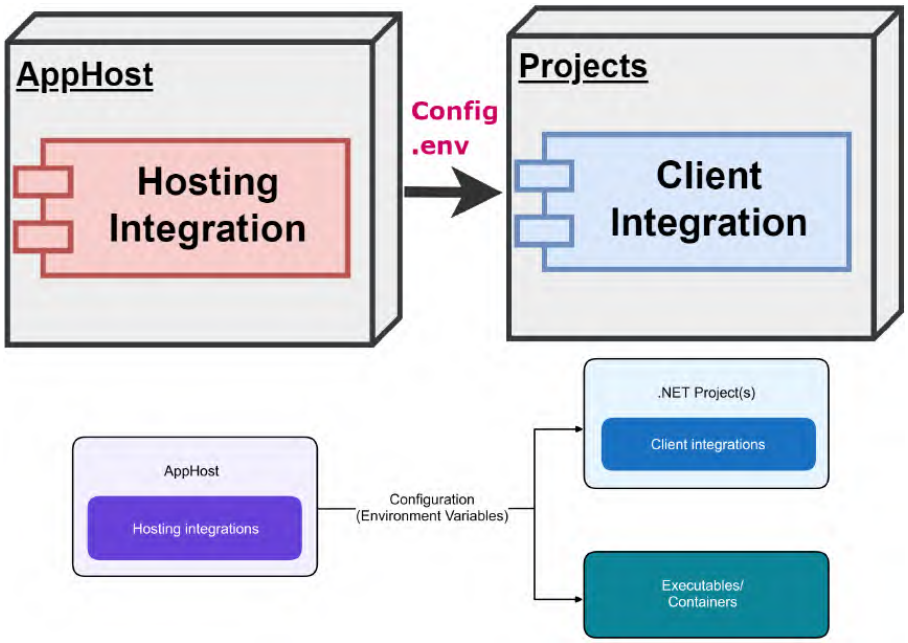


Figure 6.6 – Hosting versus client integrations

For the Catalog microservice, we will use `Aspire.Hosting.PostgreSQL` (**hosting integration**) in the `AppHost` project to configure the PostgreSQL resource and `Aspire.Npgsql.EntityFrameworkCore.PostgreSQL` (**client integration**) in the Catalog project to connect to it through Entity Framework Core.

While hosting and client integrations complement each other, they are not tightly coupled. The hosting integration makes resource configuration, including connection strings, available to projects that reference the resource, while the client integration uses that configuration to establish the connection from the consuming application.

For example, the following code configures a Redis container named `cache` in the AppHost project through a hosting integration:

```
var builder = DistributedApplication.CreateBuilder(args);
// Hosting integration call:
var cache = builder.AddRedis("cache");
// This sets up a Redis container named "cache"
```

The corresponding client integration in a .NET project can then use the resource name to connect to the distributed cache. The following figure illustrates how the hosting integration in the AppHost and the client integration in the consuming project work together.

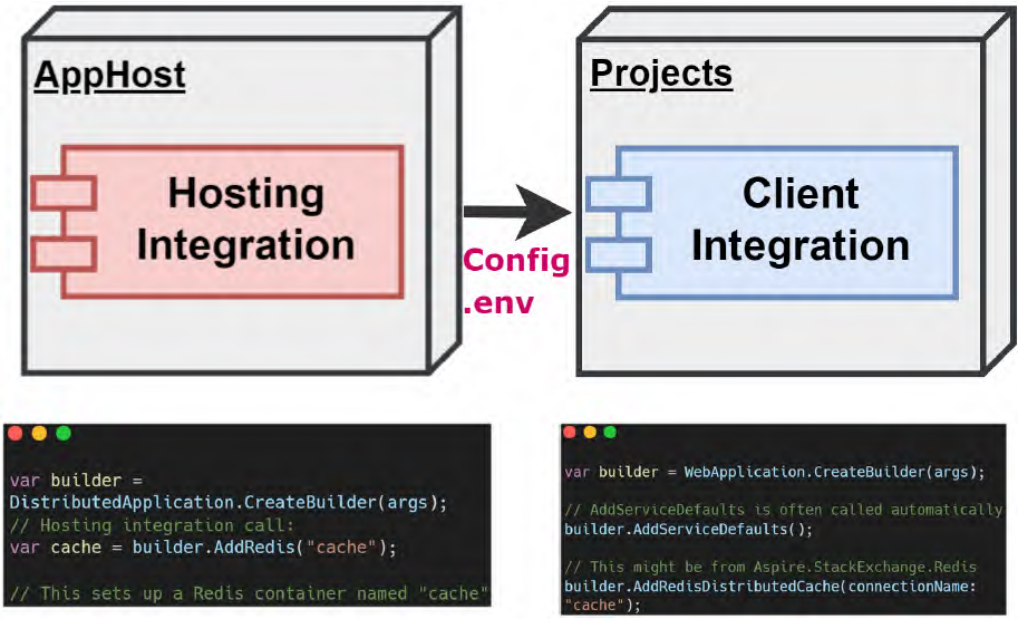


Figure 6.7 – Hosting and client integrations in Aspire, showing how configuration flows from the AppHost to a consuming project

With this distinction in mind, you can now configure PostgreSQL as the backing service for the Catalog microservice. This involves the following steps:

1. In **Solution Explorer**, right-click on **Dependencies** in your `AppHost` project.
2. Select **Add Aspire package** from the context menu.
3. In the Aspire wizard that pops up, search for `Aspire.Hosting.PostgreSQL`.
4. Select **Install**
5. Similarly, in the Catalog microservice Web API project, install the `Aspire.Npgsql.EntityFrameworkCore.PostgreSQL` NuGet package as your client integration.
6. Register the client integration in your `Program.cs` file to activate your `CatalogDbContext`.
7. Develop your data layer in the Catalog microservice using the `DbContext` object to retrieve and manage data from the PostgreSQL database.

With Aspire, backing services such as databases, caches, and message queues can be declared and orchestrated as part of the distributed application. The `AppHost` project provides an **infrastructure-as-code** approach in which you define the required services, while Aspire manages their provisioning and orchestration.

With the integration model established, you are now ready to add PostgreSQL as the backing service for the Catalog microservice in the `AppHost` project.

Adding PostgreSQL to the AppHost

To add PostgreSQL as a backing service for your Catalog microservice, you will use Aspire's infrastructure as code approach to provision and configure a PostgreSQL container directly from your `AppHost` project. This process involves installing the necessary Aspire hosting packages, updating your project file, and defining the backing service in your AppHost's `Program.cs`.

To add PostgreSQL to your `AppHost` project, you need to do the following:

1. Right-click the **Dependencies** node under your `AppHost` project in **Solution Explorer**.
2. Select the **Add Aspire package** option to filter and display only the hosting integrations available for your AppHost project.
   If the list takes too long to load, try repeating the process(right-click on **Dependencies** and select on the **Add Aspire package** option).
3. In the **NuGet Package Manager**, search for `Aspire hosting PostgreSQL` or paste `Aspire.Hosting.PostgreSQL` into the search bar to locate the required NuGet package.
   You will see a list of available hosting integrations, including `Aspire.Hosting.PostgreSQL` for PostgreSQL support in Aspire.

The following figure shows the **NuGet Package Manager** with the search filter applied for Aspire hosting integrations.

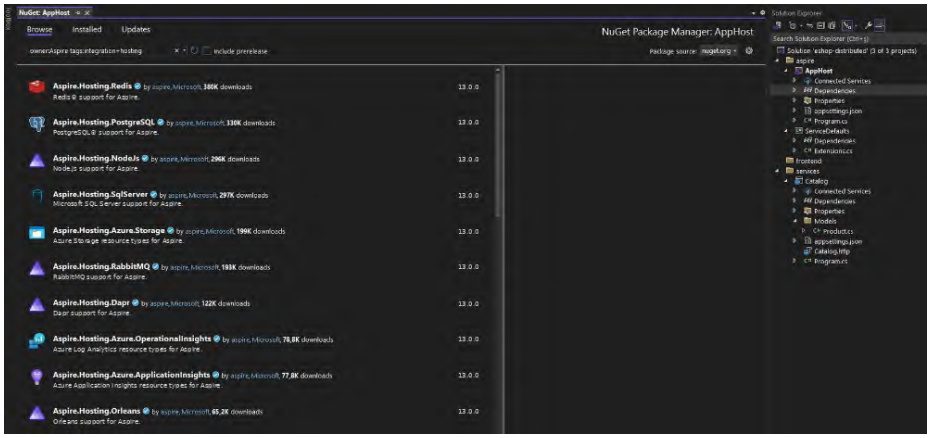


Figure 6.8 – NuGet Package Manager displaying hosting integration packages, including Aspire.Hosting.PostgreSQL, Redis, SqlServer, and others

1. After locating `Aspire.Hosting.PostgreSQL`, select it and install it under your `AppHost` project.
2. Once the package is installed, double-click your `AppHost` project file to confirm that `Aspire.Hosting.PostgreSQL` is listed as a package reference.

The following code block shows the relevant section of the `AppHost` project file with the PostgreSQL package reference included:

```
<ItemGroup><PackageReference Include="Aspire.Hosting.PostgreSQL"
Version="13.0.0" />
</ItemGroup>
<ItemGroup>
<ProjectReference Include="..\Catalog\Catalog.csproj" />
</ItemGroup>
```

After confirming the package installation, open the `Program.cs` file under your `AppHost` project. Here, you will define your backing services and projects.

The initial `Program.cs` file contains a placeholder for backing services and adds the Catalog project:

```
var builder = DistributedApplication.CreateBuilder(args);
// Backing Services
```

```
builder.AddProject<Projects.Catalog>("catalog");
builder.Build().Run();
```

Connecting the Catalog microservice to PostgreSQL

You will now update this file to define your PostgreSQL container and database as backing services, and then connect your Catalog microservice to these services.

First, add a section for backing services and another for projects to keep your configuration organized. You will activate your backing services before creating and connecting your projects.

Begin by creating the PostgreSQL resource:

```
var postgres = builder
    .AddPostgres("postgres")
    .WithPgAdmin()
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
```

In the code, the `AddPostgres` method tells Aspire to spin up a PostgreSQL container named `postgres`. The `WithPgAdmin` extension method adds the optional `pgAdmin` UI tool for managing your PostgreSQL instance. The `WithDataVolume` method ensures that a Docker volume is created for data persistence. The `WithLifetime(ContainerLifetime.Persistent)` method sets the container's lifetime to persistent, so data is retained across container restarts and the container remains running even if you close or restart your solution.

With the PostgreSQL resource defined, make it available to the Catalog microservice. Create a database named `catalogdb` within the PostgreSQL container:

```
var catalogDb = postgres.AddDatabase("catalogdb");
```

After defining your backing services, you will add your Catalog microservice project and connect it to the backing database using the `WithReference` and `WaitFor` extension methods:

```
var catalog = builder
    .AddProject<Projects.Catalog>("catalog")
    .WithReference(catalogDb)
    .WaitFor(catalogDb);
```

Here, `WithReference(catalogDb)` injects the connection string for the `catalogDB` database into your Catalog microservice, and `WaitFor(catalogDb)` ensures that the Catalog microservice waits for the PostgreSQL database to be ready before starting.

The complete `AppHostProgram.cs` file now looks like this:

```
var builder = DistributedApplication.CreateBuilder(args);
// Backing Services
var postgres = builder
    .AddPostgres("postgres")
    .WithPgAdmin()
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
var catalogDb = postgres.AddDatabase("catalogdb");
// Projects
var catalog = builder
    .AddProject<Projects.Catalog>("catalog")
    .WithReference(catalogDb)
    .WaitFor(catalogDb);
builder.Build().Run();
```

This configuration enables you to provision a PostgreSQL container and database as backing services and connect your Catalog microservice to them using Aspire's infrastructure as code approach.

Several important points need to be reiterated about how Aspire handles backing service references for your Catalog microservice. The line `var catalogDb = postgres.AddDatabase("catalogdb");` creates the PostgreSQL database automatically when the PostgreSQL server becomes ready. It also defines a database resource and provides the connection information that the Catalog microservice needs. The `.WithReference(catalogDb)` extension method makes this connection information available to the Catalog microservice through an environment variable, ensuring that it receives the correct connection string for `catalogdb`. The `.WaitFor(catalogDb)` extension method ensures that the PostgreSQL database resource is ready before the Catalog microservice starts, helping to prevent startup race conditions.

Running and inspecting the backing service

Before running the AppHost project to activate and spin up the backing services (PostgreSQL database and PG admin), make sure Docker Desktop is running on your machine.

Once Docker Desktop is started, you can right-click the `AppHost` project, set it as the **startup project**, and run it to initialize your distributed environment.

When you run the AppHost project, your backing services and the Catalog microservice are started together in the distributed environment. The **Aspire dashboard** displays the resources in the distributed application, including the PostgreSQL container, `catalogdb`, pgAdmin, and the Catalog microservice.

Once everything is running, the Catalog microservice receives the injected environment variable for the CatalogDB connection string. This environment variable contains the host, port, username, password, and database name for connecting to the PostgreSQL instance.

The following code in your `AppHostProgram.cs` file is responsible for injecting the CatalogDB connection string as an environment variable into the Catalog microservice:

```
var builder = DistributedApplication.CreateBuilder(args);
// Backing Services
var postgres = builder
    .AddPostgres("postgres")
    .WithPgAdmin()
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
var catalogDb = postgres.AddDatabase("catalogdb");
// Projects
var catalog = builder
    .AddProject<Projects.Catalog>("catalog")
    .WithReference(catalogDb)
    .WaitFor(catalogDb);
builder.Build().Run();
```

This setup allows the Catalog microservice to communicate with and retrieve data from the PostgreSQL container using the injected connection string.

You can verify the environment variables by opening the Catalog microservice in Visual Studio, navigating to the **Environment variables** section, and inspecting the CatalogDB connection string using the **text visualizer**. This confirms that the connection string is correctly injected.

To further inspect your distributed environment, open the **Aspire dashboard** and select the **endpoint** for PG admin. This launches the PG admin UI running in a container within your distributed environment.

In the UI, you can expand the **Servers** option in PG admin to see that the PostgreSQL container is created successfully. However, if you expand **Databases**, you should now see the `catalogdb`

database because Aspire creates it automatically when the PostgreSQL server becomes ready. The database is initially empty; Entity Framework Core migrations will create and update the application schema, which we will cover in the next steps.

Configuring the PostgreSQL client integration

The next step is to focus on client integration for the Catalog microservice, allowing it to connect to the PostgreSQL database using Entity Framework Core. First, you will need to install the necessary `Aspire.Npgsql.EntityFrameworkCore.PostgreSQL` package in your Catalog microservice project to enable this integration. This package enables you to easily register your Entity Framework Core DB context with Aspire integrations, automatically utilizing environment variables for connection strings and providing optional built-in resilience and logging features that align with Aspire's service defaults.

You can search for the `Aspire.Npgsql.EntityFrameworkCore.PostgreSQL` package in the **NuGet Package Manager** and install it into your Catalog microservice project.

The following screenshot shows the **NuGet Package Manager** with the `Aspire.Npgsql.EntityFrameworkCore.PostgreSQL` package selected for installation.

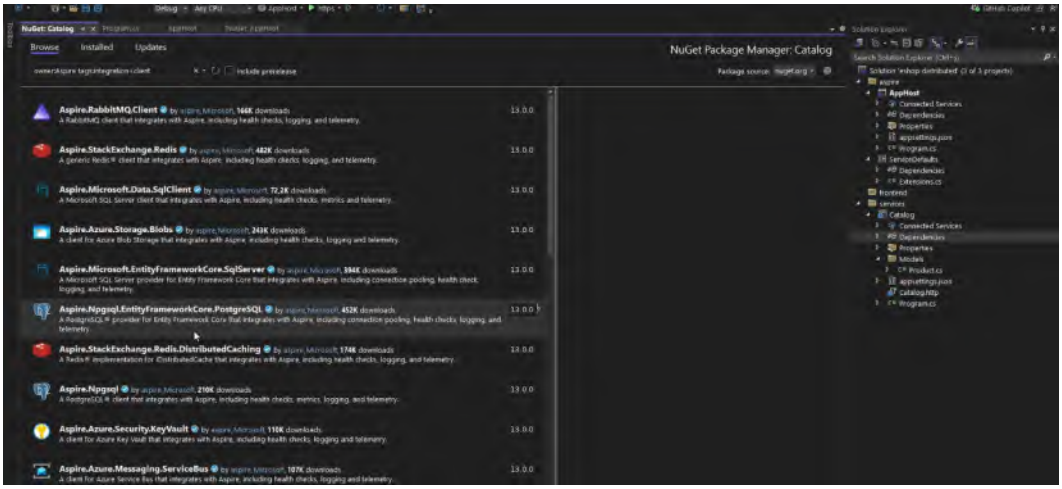


Figure 6.9 – NuGet Package Manager with Aspire.Npgsql.EntityFrameworkCore.PostgreSQL selected for installation

After installing the package, you can verify its presence in your Catalog project file. The `.csproj` file should now include a `PackageReference` for `Aspire.Npgsql.EntityFrameworkCore.PostgreSQL` with the correct version.

The following code shows the updated `Catalog.csproj` file with the package reference:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
<PropertyGroup><TargetFramework>net10.0</TargetFramework>
<Nullable>enable</Nullable>
<ImplicitUsings>enable</ImplicitUsings>
</PropertyGroup>
<ItemGroup><PackageReference
Include="Aspire.Npgsql.EntityFrameworkCore.PostgreSQL" Version="13.0.0" />
</ItemGroup>
<ItemGroup>
<ProjectReference Include="..\ServiceDefaults\ServiceDefaults.csproj" />
</ItemGroup>
</Project>
```

This package provides extension methods to register your DB context with Aspire, automatically fetching the connection string for CatalogDB and connecting it to your `ProductDbContext`. It also registers `ProductDbContext` in ASP.NET Core's dependency injection container, allowing you to inject and use it for CRUD operations at the database level.

Before registering `ProductDbContext` in `Program.cs`, first create the Catalog data layer and the `ProductDbContext` class. We will return to the registration step immediately afterward.

Creating the Catalog data layer

According to the n-layer architecture, you have already developed the `Models` folder and `Product` domain model. Now, you will need to create a `Data` folder and add the `ProductDbContext` class to interact with the PostgreSQL database.

First, let's create the `ProductDbContext` class:

1. Open Visual Studio and go to the `Catalog` microservice project.
2. Right-click the project and select **Add | New Folder**. Name the folder `Data`.
3. Now, right-click the `Data` folder and select **Add | Class**. Name the class `ProductDbContext`.

We define the `ProductDbContext` class as follows, inheriting from `DbContext` and exposing `DbSet<Product>` for CRUD operations:

```
using Catalog.Models;
using Microsoft.EntityFrameworkCore;
```

```
namespace Catalog.Data;
public class ProductDbContext : DbContext
{
    public ProductDbContext(DbContextOptions<ProductDbContext> options)
    : base(options) { }
    public DbSet<Product> Products => Set<Product>();
}
```

`ProductDbContext` inherits from Entity Framework Core's DbContext and exposes a `DbSet<Product>` named `Products`. Through this context, the application can query and persist `Product` entities in PostgreSQL.

After developing the `ProductDbContext`, you are ready to register it in your Catalog microservice project and use it for database operations.

To make the required namespaces available throughout the Catalog microservice, add a new C# class file named `GlobalUsings.cs` to the root of the Catalog project. Open the file, delete any default boilerplate, and add the global `using` statements needed across your project. For example:

```
global using Catalog.Models;
global using Microsoft.EntityFrameworkCore;
```

Defining these shared namespaces here eliminates the need to repeat the same `using` statements at the top of every individual file.

To configure the global usings and register `ProductDbContext`, follow these steps:

1. In the Catalog project, add a `GlobalUsings.cs` file if one does not already exist.
2. Add the `Catalog.Data` namespace to `GlobalUsings.cs`.
3. Move the `Microsoft.EntityFrameworkCore` using statement from `ProductDbContext.cs` to `GlobalUsings.cs`.

Before moving the namespace imports, `ProductDbContext.cs` includes the following `using` statements:

```
using Catalog.Models;
using Microsoft.EntityFrameworkCore;
```

Create or update `GlobalUsings.cs` with the following namespaces:

```
global using Catalog.Models;
global using Microsoft.EntityFrameworkCore;
global using Catalog.Data;
```

Now, return to `Program.cs` and register `ProductDbContext` after `AddServiceDefaults()` using the following line:

```
builder.AddNpgsqlDbContext<ProductDbContext>(connectionName: "catalogdb");
```

This line registers `ProductDbContext` and connects it to the `CatalogDb` using the environment variable connection setting. When you reference CatalogDb in `Program.cs`, the connection string is injected as an environment variable, such as:

```
ConnectionStrings__catalogdb
Host = localhost; Port = 58927; Username = postgres; Password =
(zNZ~UvXD~63jnkMhsP2gH; Database = catalogdb
```

This setup ensures that `ProductDbContext` is registered and ready to perform CRUD operations over the database.

Aspire orchestration injects shared configuration and service discovery through `ServiceDefaults` and environment variables.

The Catalog project requires two NuGet packages for its data layer: the Aspire PostgreSQL client integration package for connecting `ProductDbContext` to `catalogdb`, and the Entity Framework Core tools package for creating and managing migrations. The relevant package references in `Catalog.csproj` are as follows::

```
<PackageReference Include="Aspire.Npgsql.EntityFrameworkCore.PostgreSQL"
Version="13.0.0" /><PackageReference
Include="Microsoft.EntityFrameworkCore.Tools" Version="10.0.0">
<PrivateAssets>all</PrivateAssets>
<IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</
IncludeAssets>
</PackageReference>
```

With `ProductDbContext` registered, you are now ready to create the Catalog database schema using Entity Framework Core migrations.

Creating the Entity Framework Core migration

To generate the initial migration for your Entity Framework Core setup in Visual Studio, you need to use the **Package Manager Console** (can search for it using the **Search** bar). This allows you to scaffold the migration files that define how your database schema should be created or updated.

1. Open the **Package Manager Console** window in Visual Studio.
2. Right-click the `Catalog` microservice project and select **Set as Startup Project**. This ensures that migration commands target the correct project.
3. In the **Package Manager Console**, set the **Default Project** dropdown to `services\Catalog`.
4. Run the following command to create the initial migration and specify the output directory for the migration files:

```
Add-Migration InitialCreate -OutputDir Data/Migrations -Project Catalog
```

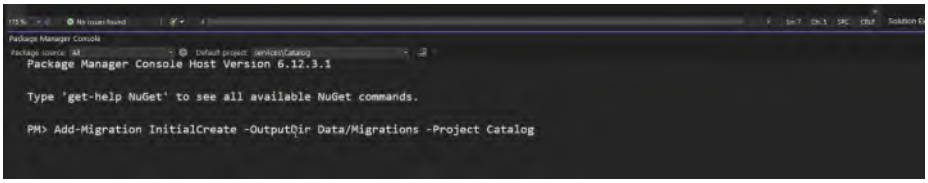


Figure 6.10 – Package Manger Console with default project set to services/Catalog and command to create initial migration

This command creates a new migration named `InitialCreate` in the `Data/Migrations` folder of your Catalog microservice project.

After running the command, you will see output indicating that the build started and succeeded, and the migration files are generated.

EF Core generates the migration from the `ProductDbContext` model you created earlier.

Once the migration is created, you will find a new folder named `Migrations` under the `Data` directory. Inside, you will see a migration class (for example, `20250210104238_InitialCreate.cs`) that contains the logic for creating and dropping the `Products` table.

The `Up` method in the migration class defines the schema for the `Products` table:

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "Products",
        columns: table => new
        {
            Id = table.Column<int>(type: "integer", nullable: false)
            .Annotation("Npgsql:ValueGenerationStrategy",
                NpgsqlValueGenerationStrategy.IdentityByDefaultColumn),
            Name = table.Column<string>(type: "text", nullable: false),
            Description = table.Column<string>(type: "text", nullable: false),
            Price = table.Column<decimal>(type: "numeric", nullable: false),
            ImageUrl = table.Column<string>(type: "text", nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Products", x => x.Id);
        });
}
```

The `Down` method provides the logic to drop the `Products` table if you need to revert the migration:

```
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(
    name: "Products");
}
```

Applying migrations and seeding data

You now have two options for applying this migration and creating the database schema in your PostgreSQL container.

The first is to run the following command in the **Package Manager Console** to apply the migration and create the database schema and `Products` table:

```
Update-Database
```

This approach is manual and suitable for one-time or local development use.

The other method is more automated. Instead of running migration commands manually, you can configure your Catalog microservice to automatically apply migrations at application startup. This is the preferred approach for distributed environments, ensuring your database schema is always up to date when the microservice runs.

To automate Entity Framework Core migrations and seed your PostgreSQL database at application startup, create an extension method and a data seeder class in your Catalog microservice.

1. Right-click the `Data` folder in your Catalog project, select **Add**, then **New Item**, and create a new C# class named `Extensions.cs`.
2. In `Extensions.cs`, define a static class and add a static extension method called `UseMigration` for the `WebApplication` type. This method will handle both migration and seeding logic.
   The following is the code for creating the `Extensions` class and the `UseMigration` extension method:

```
namespace Catalog.Data;
public static class Extensions
{
    public static void UseMigration(this WebApplication app)
    {
        using var scope = app.Services.CreateScope();
        var context =
scope.ServiceProvider.GetRequiredService<ProductDbContext>();
        context.Database.Migrate();
        DataSeeder.Seed(context);
    }
}
```

3. Next, implement a `DataSeeder` class within the same file (or separately, as preferred). This class provides a static `Seed` method that checks whether any products exist in the database and, if not, adds a predefined set of products.
The following code defines the `DataSeeder` class and its methods:

```
public class DataSeeder
{
    public static void Seed(ProductDbContext dbContext)
    {
        if (dbContext.Products.Any())
        return;
        dbContext.Products.AddRange(Products);
        dbContext.SaveChanges();
    }
    public static IEnumerable<Product> Products =>
     [
        new Product { Name = "Solar Powered Flashlight", Description = "A
fantastic product for outdoor adventures.", Price = 19.99M, ImageUrl =
"https://example.com/flashlight.jpg" },
        new Product { Name = "Hiking Poles", Description = "Ideal for
camping and hiking trips", Price = 29.99M, ImageUrl = "https://example.com/
hikingpoles.jpg" },
        new Product { Name = "Outdoor Rain Jacket", Description = "This
product will keep you warm and dry.", Price = 49.99M, ImageUrl = "https://
example.com/rainjacket.jpg" },
        new Product { Name = "Survival Kit", Description =
"A must-have for any outdoor adventurer!", Price = 39.99M, ImageUrl =
"https://example.com/survivalkit.jpg" },
        new Product { Name = "Outdoor Backpack", Description = "This
backpack is perfect for carrying your gear.", Price = 59.99M, ImageUrl =
"https://example.com/backpack.jpg" },
        new Product { Name = "Camping Cookware", Description = "This
cookware set is ideal for cooking outdoors.", Price = 24.99M, ImageUrl =
"https://example.com/cookware.jpg" },
```

```
        new Product { Name = "Camping Stove", Description = "This stove is
perfect for cooking outdoors.", Price = 34.99M, ImageUrl = "https://
example.com/stove.jpg" },
        new Product { Name = "Camping Lantern", Description =
"This lantern is perfect for lighting your campsite.", Price = 14.99M,
ImageUrl = "https://example.com/lantern.jpg" },
        new Product { Name = "Camping Tent", Description = "This tent is
perfect for camping trips.", Price = 99.99M, ImageUrl = "https://
example.com/tent.jpg" }
    ];
}
```

4. To activate the migration and seeding logic, call the `UseMigration` extension method in your `Program.cs` file, after configuring the HTTP request pipeline.
   The following code shows where to add the `UseMigration` call in `Program.cs`:

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.AddServiceDefaults();
builder.AddNpgsqlDbContext<ProductDbContext>(connectionName: "catalogdb");
var app = builder.Build();
// Configure the HTTP request pipeline.
app.MapDefaultEndpoints();
app.UseMigration();
app.UseHttpsRedirection();
```

   By adding `app.UseMigration();` after mapping default endpoints, you ensure that migrations are applied and the database is seeded with initial data every time the Catalog microservice starts.

The **Solution Explorer** in Visual Studio shows the new `Extensions.cs` file under the `Data` folder, alongside your migrations and `ProductDbContext` class.

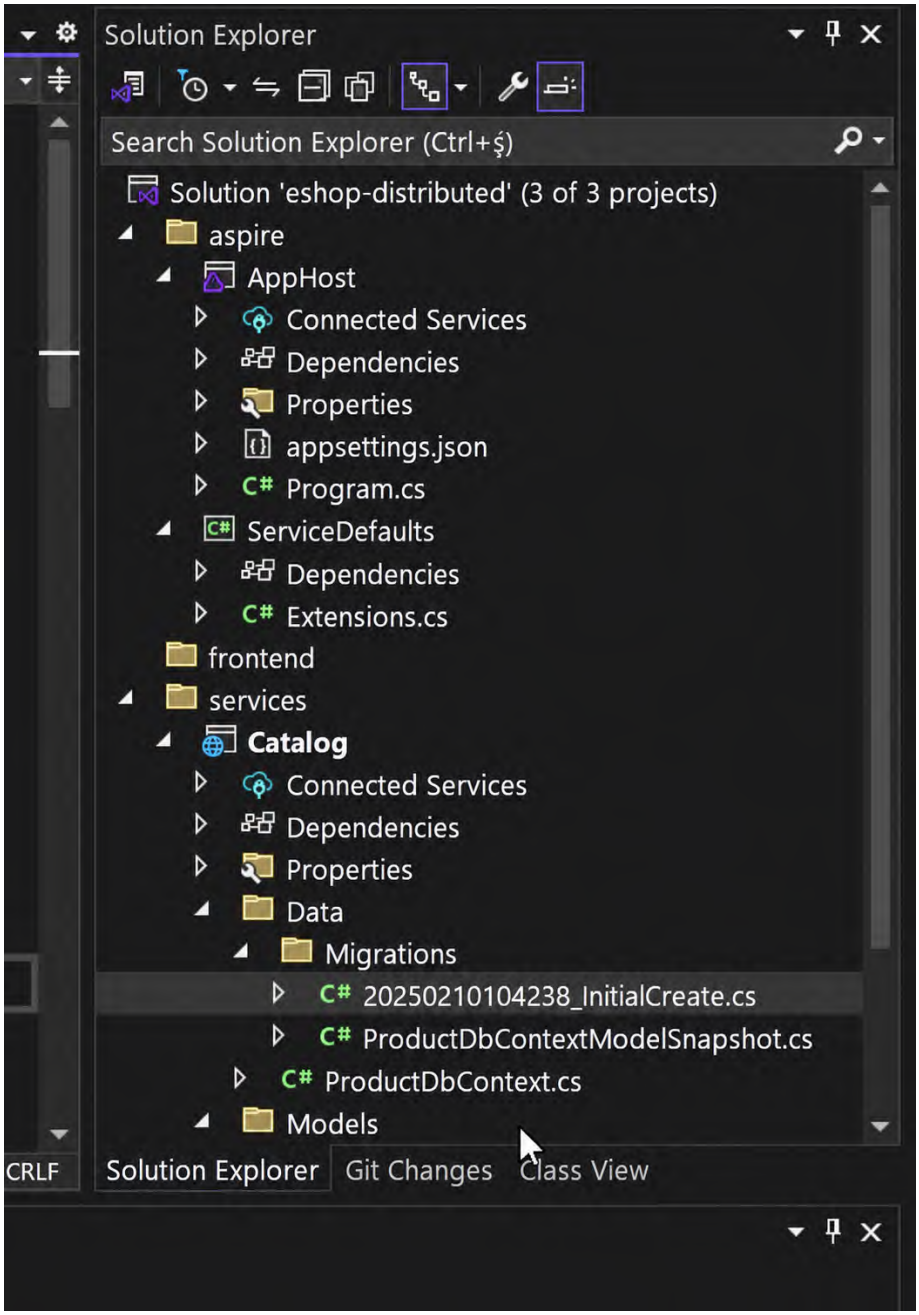


Figure 6.11 – Solution Explorer structure

To automate database migration and seeding on application startup, you use an extension method in Visual Studio that calls the `context.Database.Migrate()` method. This method is synchronous and will attempt to connect to your PostgreSQL database every time the Catalog microservice starts. Because the Catalog project waits for the PostgreSQL database resource to become ready before it starts, the migration can run against an available database rather than relying on a restart after an initial connection failure.

You are encouraged to use hosted or background services for migration operations in production scenarios, and to wrap migration logic in an environment check (such as `if (app.Environment.IsDevelopment())`) to restrict auto-migration to development environments. For simplicity and testing purposes, in this example, the migration is triggered unconditionally at startup.

Verifying migrations, seeding, and persistence

To verify the setup, you build the entire solution (right-click the solution and select on **Build Solution**) and then run the distributed application using Aspire. Make sure Docker Desktop is running before you start the AppHost project. If you want to debug the migration process, you can set a breakpoint in the `UseMigration` method, but here you simply observe the application's behavior.

When you run the AppHost project, Aspire starts the backing services in dependency order. Because Catalog waits for `catalogdb`, it starts after the PostgreSQL database resource is ready. You can monitor this startup sequence in the **AppHost dashboard**.

If the Catalog microservice cannot connect to the database for another reason, the **AppHost dashboard** can show the service in a failed state, as in the following example.



Figure 6.12 – AppHost dashboard showing an example Catalog microservice database connection failure

By selecting the **error indicator (1)**, you will be able to access the structured logs, which show an error message indicating that a connection to the `'catalogdb'` database could not be established.

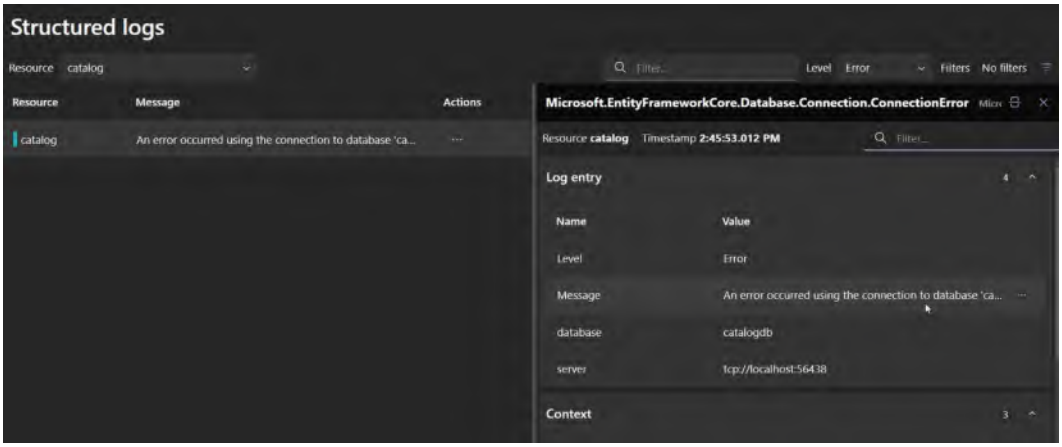


Figure 6.13 – Structured logs in AppHost showing a database connection error for the Catalog microservice

With the AppHost dependency configured as shown earlier, Aspire starts the Catalog microservice after the PostgreSQL database resource is ready. The migration and seeding commands can then run during application startup. If you want to restrict migration to development environments, you can wrap the call in an environment check:

```
if (app.Environment.IsDevelopment())
app.UseMigration();
```

The `ServiceDefaults` project provides shared logging, health checks, telemetry, and resilience defaults for the Catalog microservice. Startup ordering for the database dependency is controlled by AppHost through `WaitFor(catalogDb)`.

Once the migration and seeding complete successfully, the Catalog microservice starts up without errors, and your database is ready for use.

When your application starts, it applies the pending migration and creates the `Products` table by running the `Up` method in your migration. This ensures that the required database schema changes are applied at startup. You can see the Catalog microservice applying these migrations in the application logs.

Additionally, `insert` commands are executed to seed the database with initial data during application startup. This seeding operation populates your tables with default entries so your application has data to work with immediately.

To verify the results, return to the **Resources** view and open **pgAdmin** by clicking its endpoint. **pGAdmin** is the management tool for the PostgreSQL database.



Figure 6.14 – The pgAdmin dashboard interface with options to add a new server or configure pgAdmin

If you expand `Servers`, you will see the PostgreSQL database container. Expanding **Databases** you will see that the `catalogdb` database has been created successfully as a result of your application's startup process.

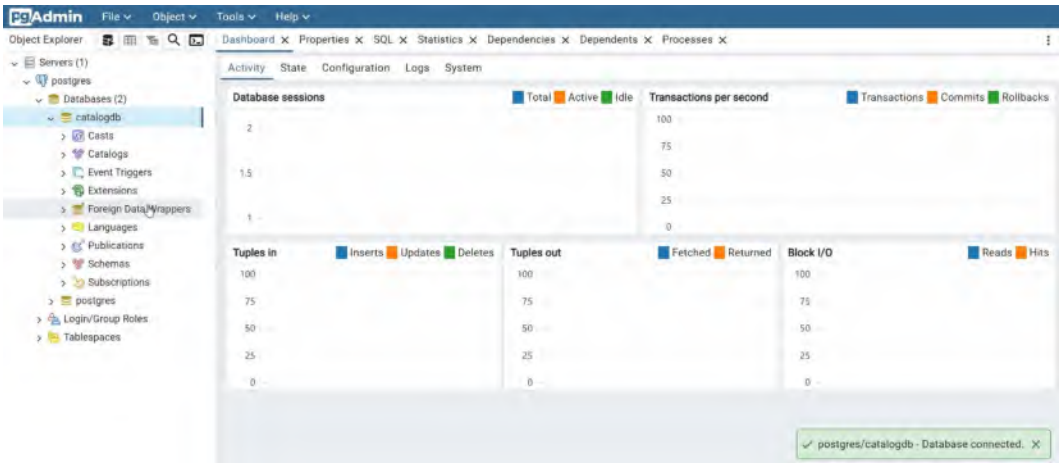


Figure 6.15 – The catalogdb database is visible under the Databases node in pgAdmin, confirming its creation

Next, if you expand `catalogdb`, then navigate to `Schemas` | `public` | `Tables`. Here, you will find the `Products` table, which was created by the migration. Right-click on the `Products` option, select **Scripts**, and run a **SELECT** script to view its contents. This confirms that the data seeding operation has completed successfully.

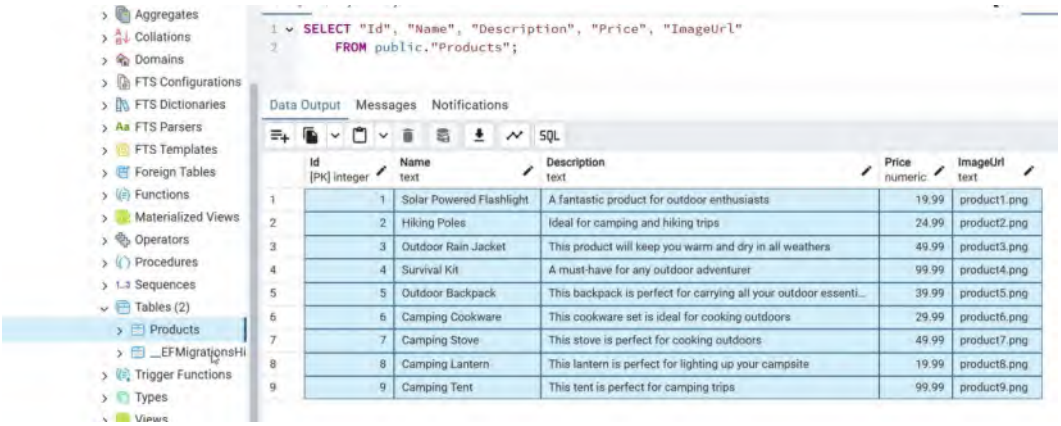


Figure 6.16 – The Products table in pgAdmin displays nine seeded rows with columns Id, Name, Description, Price, and ImageUrl

You will see nine rows in the `Products` table, indicating that the seeding process inserted the expected data at application startup. You will also notice the `__EFMigrationsHistory` table, which tracks which migrations have been applied.

If you stop your application and restart it, the Catalog microservice will check the migrations history table on startup. Since the migrations have already been applied, it will not attempt to re-apply them. You can confirm this by checking the structured logs for the Catalog microservice, where you will see that no new migrations are executed if they already exist.

This persistence across application restarts is achieved by configuring your PostgreSQL container in `AppHost` with a data volume and setting the container lifetime to persistent. The following code in your `AppHost Program.cs` file demonstrates these settings:

```
var builder = DistributedApplication.CreateBuilder(args);
// Backing Services
var postgres = builder
    .AddPostgres("postgres")
    .WithPgAdmin()
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
var catalogDb = postgres.AddDatabase("catalogdb");
// Projects
var catalog = builder
    .AddProject<Projects.Catalog>("catalog")
    .WithReference(catalogDb)
```

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

By using a data volume and persistent lifetime, your database schema and data remain intact even if you stop and restart your solution. This setup ensures that your Entity Framework Core migrations are applied automatically, your seeded data is present in the PostgreSQL database, and the restart behavior is reliable and graceful.

Implementing the Product service layer

With the PostgreSQL database, migrations, and seeded data now persisting across application restarts, the Catalog data layer is ready. The next step is to build the service and endpoint layers that expose product operations through the Catalog microservice.

Creating the Catalog service layer

As you might recall, the Catalog project is organized into four main folders: `Models`, `Data`, `Services`, and `Endpoints`.

With the `Models` and `Data` layers already in place, the next step is to add the service layer for the Catalog microservice before moving to the HTTP endpoints. The service layer will contain the operations used to retrieve, create, update, and delete products through `ProductDbContext`.

To implement the business logic for your Catalog microservice, we'll need to first create a `Services` folder. To do that, right select on the Catalog microservice and go to **Add | New Folder**. Name the folder `Services`. Then, create a `ProductService` class inside this `Services` folder. We will write the business logic inside this class.

The following code creates an empty `ProductService` class in the `Catalog.Services` namespace:

```
namespace Catalog.Services;
public class ProductService
{
}
```

You inject `ProductDbContext` into the `ProductService` class using primary constructor syntax, which allows you to access the database context for CRUD operations:

```
namespace Catalog.Services;
public class ProductService(ProductDbContext dbContext)
```

```
{
}
```

To implement the create operation for products, add an asynchronous method that takes a `Product` entity as a parameter, adds it to the database, and saves the changes asynchronously:

```
namespace Catalog.Services;
public class ProductService(ProductDbContext dbContext)
{
    public async Task CreateProductAsync(Product product)
    {
        dbContext.Products.Add(product);
        await dbContext.SaveChangesAsync();
    }
}
```

We will expand this `ProductService` class with additional methods for updating, deleting, and retrieving products as you build out the Catalog microservice use cases.

To implement the update operation, add an `UpdateProductAsync` method that takes two `Product` parameters: the existing `Product` entity from the database (`updatedProduct`) and the new values (`inputProduct`). Update the properties of the existing entity with the new values, then call the update and save methods on the `DbContext`:

```
public async Task UpdateProductAsync(Product updatedProduct, Product inputProduct)
{
    // update product with new values
    updatedProduct.Name = inputProduct.Name;
    updatedProduct.Description = inputProduct.Description;
    updatedProduct.ImageUrl = inputProduct.ImageUrl;
    updatedProduct.Price = inputProduct.Price;
    dbContext.Products.Update(updatedProduct);
    await dbContext.SaveChangesAsync();
}
```

This approach allows you to keep your business logic in the services layer, separate from your data access and API endpoint logic.

To implement product queries using the `ProductDbContext` object, we will add methods to the `ProductService` class that handle retrieving products from the database. Begin by defining the `GetProductsAsync` method at the top of the `ProductService` class.

The following code defines the `GetProductsAsync` method, which loads all rows from the `Products` table and returns them as an `IEnumerable<Product>`:

```
public async Task<IEnumerable<Product>> GetProductsAsync()
{
    return await dbContext.Products.ToListAsync();
}
```

This method uses Entity Framework Core to asynchronously retrieve all products from the database.

Next, add the `GetProductByIdAsync` method to retrieve a single product by its primary key. This method accepts an integer id as a parameter and uses the `FindAsync` method for primary key lookups:

```
public async Task<Product?> GetProductByIdAsync(int id)
{
    return await dbContext.Products.FindAsync(id);
}
```

If the product exists, this method returns the product; otherwise, it returns null. You will use this lookup whenever an endpoint needs to work with an existing product.

Finally, we add the `DeleteProductAsync` method. It accepts an existing `Product` entity, removes it from the `Products` set, and saves the change:

```
public async Task DeleteProductAsync(Product deletedProduct)
{
    dbContext.Products.Remove(deletedProduct);
    await dbContext.SaveChangesAsync();
}
```

When you later implement the update and delete endpoints, first call `GetProductByIdAsync` to retrieve the existing tracked entity. If no product is found, the endpoint can return `404 Not Found` without calling `UpdateProductAsync` or `DeleteProductAsync`.

With these additions, your `ProductService` class now includes methods for reading (`GetProductsAsync` and `GetProductByIdAsync`), creating (`CreateProductAsync`), updating (`UpdateProductAsync`), and deleting (`DeleteProductAsync`) products. All business-related operations for products are encapsulated within this service class.

After implementing these methods, right-click the solution (`eshop-distributed`) and build the entire solution to confirm that the build succeeds.

Registering ProductService with dependency injection

We now need to register the `ProductService` class with the ASP.NET Core dependency injection container in the Catalog microservice before we can use it. To do this, open the Catalog project's `Program.cs` file and navigate to the section labeled **Add services to the container**.

To register your `ProductService` class with the ASP.NET Core dependency injection container, you need to add it to the services collection in your `Program.cs file`. Registering the service ensures that a new instance of `ProductService` is created for each HTTP request, following the scoped lifetime pattern commonly used in dependency injection.

The following code registers `ProductService` as a scoped service immediately after registering your `ProductDbContext`:

```
builder.Services.AddScoped<ProductService>();
```

This line tells the dependency injection container to instantiate `ProductService` once per HTTP request and dispose of it at the end of the request. The `ProductService` class itself injects `ProductDbContext`, which is also registered in the container. This means that when `ProductService` is created, `ProductDbContext` is automatically provided as a dependency.

The relevant section of `Program.cs`, where both `ProductDbContext` and `ProductService` are registered is as follows:

```
using Catalog.Services;
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.AddServiceDefaults();
builder.AddNpgsqlDbContext<ProductDbContext>(connectionName: "catalogdb");
builder.Services.AddScoped<ProductService>();
var app = builder.Build();
// Configure the HTTP request pipeline.
app.MapDefaultEndpoints();
app.UseMigration();
app.UseHttpsRedirection();
var summaries = new[]
{
```

```
    "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;
});
```

You should move any using statements for `Catalog.Services` to your global usings file (`GlobalUsings.cs`) for clarity and maintainability.

Now that we have registered our `ProductService` successfully, we can inject it into our endpoint handlers to implement the business logic for your Catalog microservice.

Creating the Product endpoints

The Catalog project uses minimal APIs to expose the product operations implemented in ProductService. To keep the endpoint definitions separate from the service and data layers, create an `Endpoints` folder in the Catalog project. In **Solution Explorer**, right-click the Catalog project and select **Add | New Folder**. Name the folder `Endpoints`. Then, create a `ProductEndpoints.cs` file inside this folder.

Begin by defining `ProductEndpoints` as a static class:

```
namespace Catalog.Endpoints
{
    public static class ProductEndpoints
    {
    }
}
```

Next, add a static extension method called `MapProductEndpoints` that registers all product-related endpoints with the application's HTTP request pipeline. Define a `MapProductEndpoints` as an extension method on `IEndpointRouteBuilder`:

```
public static void MapProductEndpoints(this IEndpointRouteBuilder app)
{
}
```

Within this method, use the `MapGroup` method to group all endpoints under the `"/products"` route.

```
var group = app.MapGroup("/products");
```

This creates a route group with `/products` as the common prefix. Any endpoint you define through `group` will therefore be exposed under the `/products` route. You can now add the CRUD endpoints to this group.

Retrieving all products

We'll start by implementing the `GET` operation to retrieve all products. Inject `ProductService` to handle the business logic, and use the `GetProductsAsync` method to fetch the data. Return a `200 OK response` with the list of products.

```
// GET all
group.MapGet("/", async (ProductService service) =>
{
    var products = await service.GetProductsAsync();
    return Results.Ok(products);
})
.WithName("GetAllProducts")
.Produces<List<Product>>(StatusCodes.Status200OK);
```

This configuration names the endpoint `"GetAllProducts"` and specifies that it produce a list of `Product` objects with a 200 OK status code. When a request is sent to `GET /products/`, ASP.NET Core resolves `ProductService` from the dependency injection container. The service retrieves the products from the database, and the endpoint returns them with a `200 OK` response.

Retrieving a product by ID

Here, we define the `GET` endpoint for retrieving a product by its ID, using the `ProductService` to fetch the product and returning either the product or a `404 Not Found` result:

```
// GET by ID
group.MapGet("/{id}", async (int id, ProductService service) =>
{
    var product = await service.GetProductByIdAsync(id);
    if (product is null) return Results.NotFound();
    return Results.Ok(product);
})
.WithName("GetProductById")
.Produces<Product>(StatusCodes.Status200OK)
.Produces(StatusCodes.Status404NotFound);
```

The `{id}` segment captures the product ID from the request URL. `GetProductByIdAsync` searches for the corresponding product. If no product is found, the endpoint returns `404 Not Found`; otherwise, it returns the product with `200 OK`.

Creating a product

After defining the `GET by ID` operation, we'll add the `POST` endpoint to create a new product. This endpoint receives the product information from the request body, injects the `ProductService`, and calls the `CreateProductAsync` method. The response returns a `201 Created` status with the location of the new product and the product data:

```
// POST (Create)
group.MapPost("/", async (Product product, ProductService service) =>
{
    await service.CreateProductAsync(product);
    return Results.Created($"/products/{product.Id}", product);
})
    .WithName("CreateProduct")
    .Produces<Product>(StatusCodes.Status201Created);
```

The product information is supplied in the request body and bound to a `Product` object. `CreateProductAsync` adds the product to the database and saves the changes. The endpoint then returns `201 Created`, together with the location of the newly created product.

Updating a product

We then implement the HTTP `PUT` operation to update an existing product. This endpoint requires the product ID as a route parameter and the updated product data in the request body. To handle the business logic, the endpoint uses the injected `ProductService`.

```
// PUT (Update)
group.MapPut("/{id}", async
(int id, Product inputProduct, ProductService service) =>
{
    var updatedProduct = await service.GetProductByIdAsync(id);
    if (updatedProduct is null) return Results.NotFound();
    await service.UpdateProductAsync(updatedProduct, inputProduct);
    return Results.NoContent();
})
    .WithName("UpdateProduct")
    .Produces(StatusCodes.Status404NotFound)
    .Produces(StatusCodes.Status204NoContent);
```

Before updating the product, the endpoint retrieves the existing entity using `GetProductByIdAsync`. If the product does not exist, it returns `404 Not Found`. Otherwise, the existing entity and the new values are passed to `UpdateProductAsync`. A successful update returns `204 No Content`.

Deleting a product

Lastly, we're going to expose the HTTP `DELETE` endpoint:

```
// DELETE
group.MapDelete("/{id}", async (
    int id,
    ProductService service) =>
{
    var deletedProduct =
        await service.GetProductByIdAsync(id);
    if (deletedProduct is null)
        return Results.NotFound();
    await service.DeleteProductAsync(deletedProduct);
    return Results.NoContent();
})
    .WithName("DeleteProduct")
```

```
    .Produces(StatusCodes.Status404NotFound)
    .Produces(StatusCodes.Status204NoContent);
```

The HTTP `DELETE` operation is exposed using the `MapDelete` method, mapped to the `/products/{id}` endpoint. In this method, you retrieve the product by ID from the database, then call `service.DeleteProductAsync` with the entity to remove it from the DB context and save the changes. The endpoint returns a `204 No Content` response to indicate successful deletion.

With this, your `ProductEndpoints` class now exposes all CRUD operations for the `Product` entity, completing the presentation layer for your Catalog microservice.

Defining the endpoints is not enough; you must also add them to the Catalog microservice's HTTP request pipeline.

Registering the Product endpoints

To make the `Product` endpoints available, open the Catalog microservice's `Program.cs` file and add a call to the `MapProductEndpoints` extension method. This method is defined in the `Catalog.Endpoints` namespace and registers the product routes with the application.

Before adding the call in `Program.cs`, recall how the extension method is defined in `ProductEndpoints.cs`:

```
namespace Catalog.Endpoints;
public static class ProductEndpoints
{
    public static void MapProductEndpoints(this IEndpointRouteBuilder app)
    {
        var group = app.MapGroup("/products");
        // GET all
        group.MapGet("/", async (ProductService service) =>
        {
            var products = await service.GetProductsAsync();
            return Results.Ok(products);
        })
        .WithName("GetAllProducts")
        .Produces<List<Product>>(StatusCodes.Status200OK);
        // GET by ID
        group.MapGet("/{id}", async (int id, ProductService service) =>
        {
```

```
            var product = await service.GetProductByIdAsync(id);
            if (product is null) return Results.NotFound();
            return Results.Ok(product);
        })
        .WithName("GetProductById")
        .Produces<Product>(StatusCodes.Status200OK)
        .Produces(StatusCodes.Status404NotFound);
        // POST (Create)
        group.MapPost("/", async (Product product, ProductService service) =>
        {
            await service.CreateProductAsync(product);
            return Results.Created($"/products/{product.Id}", product);
        })
        .WithName("CreateProduct")
        .Produces<Product>(StatusCodes.Status201Created);
        // PUT (Update)
        group.MapPut("/{id}", async (int id, Product inputProduct, ProductService
service) =>
        {
            var updatedProduct = await service.GetProductByIdAsync(id);
            if (updatedProduct is null) return Results.NotFound();
            await service.UpdateProductAsync(updatedProduct, inputProduct);
            return Results.NoContent();
        })
        .WithName("UpdateProduct")
        .Produces(StatusCodes.Status404NotFound)
        .Produces(StatusCodes.Status204NoContent);
        // DELETE
        group.MapDelete("/{id}", async (int id, ProductService service) =>
        {
            var deletedProduct = await service.GetProductByIdAsync(id);
            if (deletedProduct is null) return Results.NotFound();
            await service.DeleteProductAsync(deletedProduct);
            return Results.NoContent();
        })
        .WithName("DeleteProduct")
        .Produces(StatusCodes.Status404NotFound)
        .Produces(StatusCodes.Status204NoContent);
    }
}
```

The `MapProductEndpoints` method groups all product-related endpoints under the `/products` route and registers handlers for the `GET`, `POST`, `PUT`, and `DELETE` operations.

To complete the registration, we need to add a call to `MapProductEndpoints` in your `Program.cs` file, ensuring that all `Product` endpoints are exposed as part of your Catalog microservice's HTTP **API**.

To configure the HTTP request pipeline for your Catalog microservice, you need to add the appropriate extension methods in your `Program.cs` file. This ensures that your `Product` endpoints are exposed and accessible.

The following code in `Program.cs` demonstrates how to set up the request pipeline, including the addition of the `MapProductEndpoints` extension method:

```
using Catalog.Endpoints;
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.AddServiceDefaults();
builder.AddNpgsqlDbContext<ProductDbContext>(connectionName: "catalogdb");
builder.Services.AddScoped<ProductService>();
var app = builder.Build();
// Configure the HTTP request pipeline.
app.MapDefaultEndpoints();
app.UseMigration();
app.MapProductEndpoints();
app.UseHttpsRedirection();
app.Run();
```

By calling `app.MapProductEndpoints();`, you expose all product-related endpoints for your Catalog microservice. This method groups the endpoints under the `/products` route and registers handlers for `GET`, `POST`, `PUT`, and `DELETE` operations.

You should also remove any obsolete or default endpoints that are no longer relevant to your Catalog microservice. For example, you can delete the `WeatherForecast` endpoint and its related code to keep your `Program.cs` file clear and concise.

To do this, select and remove the following code block from `Program.cs`:

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

After removing the `WeatherForecast` code, your `Program.cs` should look like this:

```
using Catalog.Endpoints;
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.AddServiceDefaults();
builder.AddNpgsqlDbContext<ProductDbContext>(connectionName: "catalogdb");
builder.Services.AddScoped<ProductService>();
var app = builder.Build();
// Configure the HTTP request pipeline.
app.MapDefaultEndpoints();
app.UseMigration();
app.MapProductEndpoints();
app.UseHttpsRedirection();
app.Run();
```

This streamlined setup ensures that only your Catalog microservice endpoints are exposed and that the HTTP request pipeline is configured according to your requirements.

You are now ready to test your Catalog microservice endpoints using the configured HTTP pipeline.

Testing the endpoints

To enable testing and development of your Catalog microservice endpoints, we will use the `Catalog.http` file provided by ASP.NET Web API out-of-the-box. This file allows you to define and invoke HTTP requests directly within Visual Studio, making it easy to interact with your endpoints during development. You can access `Catalog.http` from **Solution Explorer** by expanding the Catalog project and opening the file directly.

Preparing Catalog.http and starting the application

We will begin by defining the Catalog host address at the top of the `Catalog.http` file. This address is set to the endpoint you copy from the Aspire dashboard, ensuring that your requests target the correct running instance of the Catalog microservice. To get the endpoint for the Catalog microservice, right-click on **AppHost**, go to **Set as Startup Project**, and run the application once you confirm Docker Desktop is running. Copy the Catalog microservice endpoint.

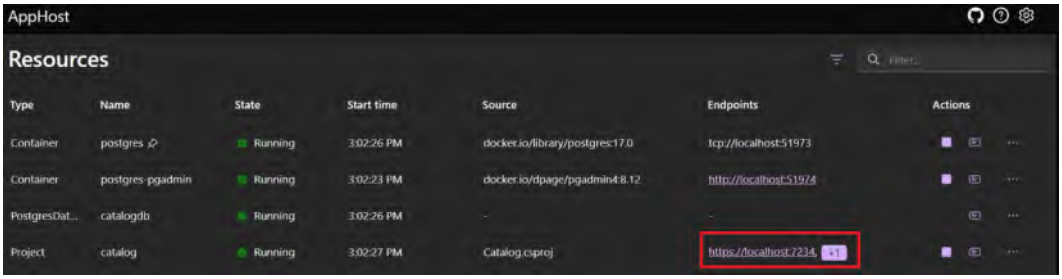


Figure 6.17 – Copy the Catalog microservice endpoint

The following code sets the Catalog host address to the local HTTPS endpoint:

```
@Catalog_HostAddress = http://localhost:5188
```

Replace the default weather forecast endpoint with your copied endpoint, as your Catalog microservice no longer exposes any weather-related endpoints. Instead, you focus on product-related endpoints.

To align with your `Product` endpoints, you update the Catalog host address to include the `/products` route, which matches the endpoint group defined in your `ProductEndpoints` class.

The updated Catalog host address is shown below:

```
@Catalog_HostAddress = https://localhost:7234/products
```

With the host address configured, replace the default `WeatherForecast` request with the product requests used in this exercise. The completed `Catalog.http` file can contain the following requests:

```
GET {{Catalog_HostAddress}}/
Accept: application/json
###
GET {{Catalog_HostAddress}}/1
Accept: application/json
###
POST {{Catalog_HostAddress}}/
Content-Type: application/json
{
    "id": 10,
    "name": "NEW Swn Flashlight",
    "description": "A NEW swn product for outdoor enthusiasts",
    "price": 59.99,
    "imageUrl": "product10.png"
}
###
PUT {{Catalog_HostAddress}}/10
Content-Type: application/json
{
    "name": "UPDATED Swn Flashlight",
    "description": "An UPDATED swn product for outdoor enthusiasts",
    "price": 19.99,
    "imageUrl": "product10.png"
}
###
DELETE {{Catalog_HostAddress}}/10
Content-Type: application/json
```

With `Catalog.http` prepared, run the distributed application before sending the requests:

1. Right-click the **AppHost** project in **Solution Explorer**.
2. Set it as the startup project.
3. Start Docker Desktop.
4. Run the application.

When the **Aspire dashboard** opens, verify that the PostgreSQL container, **pgAdmin**, **catalogdb**, and the **Catalog project** are all running. The dashboard also shows the endpoint for the Catalog project, which should match the host address you configured in `Catalog.http`.

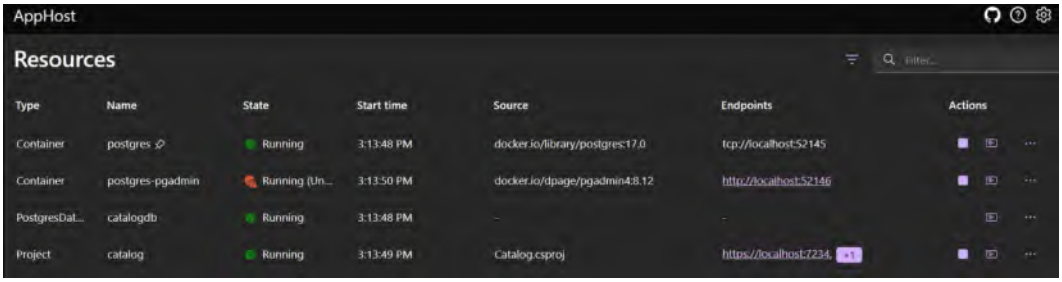


Figure 6.18 – AppHost dashboard showing running containers for postgres, postgres-pgadmin, the catalogdb database, and the catalog project, with their respective endpoints and statuses.

Testing and debugging product retrieval

To retrieve all products, send an HTTP `GET` request using the `Catalog.http` file. Select **Send request** next to the `GET` method to list all products:

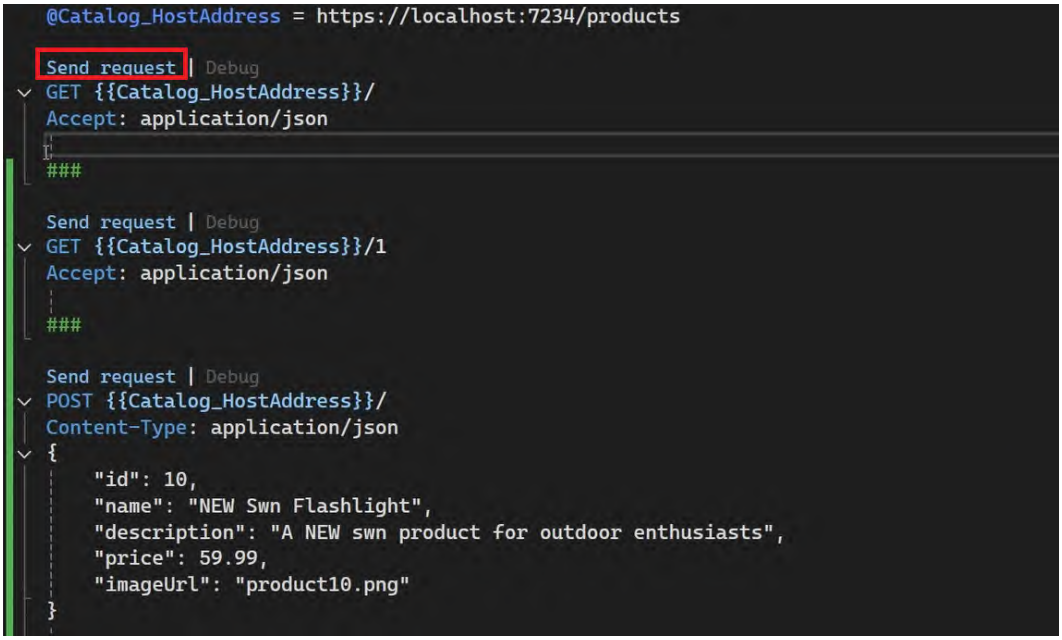


Figure 6.19 – Selecting "Send request" retrieves list of all products

A successful request returns `200 OK` and the nine products inserted during database seeding. This confirms that the `GET` endpoint can retrieve the seeded Catalog data and return it as **JSON**.

Next, debug the `GET by ID` endpoint. In `ProductEndpoints.cs`, place a breakpoint on the first executable line inside the handler: `var product = await service.GetProductByIdAsync(id);`. The surrounding endpoint code is shown here:

```
// GET by ID
group.MapGet("/{id}", async (int id, ProductService service) =>
{
    var product = await service.GetProductByIdAsync(id);
    if (product is null) return Results.NotFound();
    return Results.Ok(product);
})
    .WithName("GetProductById")
    .Produces<Product>(StatusCodes.Status200OK)
    .Produces(StatusCodes.Status404NotFound);
```

With the breakpoint set, send the `GET by ID` request already defined in `Catalog.http` for product ID 1:

```
GET {{Catalog_HostAddress}}/1
Accept: application/json
```

When you select **Send request**, execution pauses at the breakpoint. Confirm that id is 1, then step through `GetProductByIdAsync` to observe the lookup. Continue execution to complete the request; the response should return `200 OK` with the product whose ID is 1.

Testing create, update, and delete operations

Next, debug product creation. In the `MapPost` handler, place a breakpoint on `await service.CreateProductAsync(product);`:

```
// POST (Create)
group.MapPost("/", async (Product product, ProductService service) =>
{
    await service.CreateProductAsync(product);
    return Results.Created($"/products/{product.Id}", product);
})
```

```
    .WithName("CreateProduct")
    .Produces<Product>(StatusCodes.Status201Created);
```

Send the `POST` request already defined in `Catalog.http`. When execution pauses, inspect the product object and confirm that the values from the JSON request body have been bound correctly.

Continue execution and step into `CreateProductAsync` if you want to observe the product being added to `ProductDbContext` and saved to PostgreSQL.

After the request completes, send the `GET-all` request again. The response should now include the newly created product with ID 10.

If the request remains paused at a breakpoint for longer than the HTTP client's configured 20-second timeout, the `Catalog.http` request can report that it timed out. In the demonstration, this occurs while debugging the `POST` operation.

A client timeout does not necessarily mean that the database operation failed. If execution later continues and the insert completes, resending the same `POST` request with ID 10 attempts to insert the same primary key again, and PostgreSQL returns a duplicate-key error.

Before retrying a timed-out create request, check the current state by sending a `GET` request. If product 10 is already present, continue with the update test instead of posting it again.

Next, to test the update operation, send the `PUT` request in `Catalog.http` for product ID 10.

A successful update returns a `204 No Content` response. You can confirm the update by sending a `GET` request for all products and checking that the name and description have been updated.

Next, send the `DELETE` request for product ID 10.

A successful deletion returns `204 No Content`. Send the `GET-all` request once more and confirm that product 10 has been removed, returning the Catalog to the original nine seeded products.

You can also monitor all operations from the Aspire dashboard. The dashboard provides access to structured logs, which help you identify exceptions and see which commands were executed. You can also view traces for each CRUD operation, including `GET`, `GET BY ID`, `POST`, `PUT`, and `DELETE`.

Open **Structured logs** for the Catalog resource to inspect the database commands and any errors generated while the requests run. Then, open **Traces** to view the `GET`, `GET-by-ID`, `POST`, `PUT`, and `DELETE` requests and their durations.

You have now verified the Catalog microservice end to end: `Catalog.http` sends the requests, the minimal API endpoints route them through `ProductService`, PostgreSQL persists the changes, and the Aspire dashboard provides logs and traces for troubleshooting.

Summary

In this chapter, we developed the Catalog microservice from its domain and technical design through to a working implementation. We began by identifying the `Product` domain model and the CRUD operations the service needs to support, then organized the project using an n-layer structure with `Models`, `Data`, `Services`, and `Endpoints` folders. We also saw how Aspire separates hosting integrations in AppHost from client integrations in the consuming microservice.

We then configured PostgreSQL as the Catalog backing service, connected the Catalog project to it through Entity Framework Core, created `ProductDbContext`, and generated the initial migration. We automated migration and data seeding at application startup and configured a persistent PostgreSQL data volume so that the database schema, migration history, and seeded product data survive application restarts.

With the data layer in place, we implemented `ProductService` to encapsulate the operations for retrieving, creating, updating, and deleting products. We registered this service with ASP.NET Core dependency injection and then exposed those operations through minimal API endpoints grouped under `/products`. Finally, we used `Catalog.http` to exercise the `GET`, `POST`, `PUT`, and `DELETE` endpoints and used the Aspire dashboard, structured logs, and traces to observe and troubleshoot the requests.

At this point, the Catalog microservice can persist product data in PostgreSQL and expose it through a clear service and API layer. In the next chapter, we will build the Basket microservice, using Redis as a **distributed cache** for shopping-cart data. We will also connect the Basket microservice to the Catalog microservice so that basket operations can work with up-to-date product information, extending the distributed application from an independently functioning Catalog service to collaborating microservices.

7

Developing the Basket Microservice with Redis and Aspire

In the previous chapter, we developed the Catalog microservice with PostgreSQL and Aspire, establishing the product data and **APIs** that other services can consume. In this chapter, we will build the Basket microservice, using Redis as a distributed cache and Aspire to orchestrate the service and its backing infrastructure.

We will begin by analyzing the Basket domain, focusing on its domain models, use cases, **REST** APIs, and data-storage requirements, before moving into the technical design, including the architecture, patterns, libraries, and project organization that will shape the implementation. We will then create the `ShoppingCart` and `ShoppingCartItem` domain models and organize the project into `Models`, `Services`, and `Endpoints` folders.

With the structure in place, we will configure Redis as a backing service through Aspire, connect the Basket microservice to Redis using `StackExchange.Redis` and `IDistributedCache`, and implement the Basket service and minimal API endpoints for retrieving, creating or updating, and deleting shopping carts. Finally, we will test the basket operations and inspect the stored shopping-cart data in Redis.

By the end of the chapter, you will have a functional Basket microservice backed by Redis, integrated with the Catalog microservice, and orchestrated through Aspire, with clear separation between domain models, business logic, and API endpoints.

In this chapter, we'll be covering the following topics:

- Domain analysis of the Basket microservice
- Technical analysis of the Basket microservice
- Creating the Basket microservice project with Aspire
- Creating the Basket domain models
- Integrating Redis as a distributed cache with Aspire
- Developing the Basket service class caching logic
- Developing the Basket API endpoints with ASP.NET Core Minimal APIs
- Registering the Basket endpoints
- Testing the Basket microservice operations with Redis

Domain analysis of the Basket microservice

As mentioned, in this chapter, we'll develop the Basket microservice using Redis as a distributed cache, orchestrated with Aspire. We'll begin by analyzing the Basket microservice domain, focusing on its core entities, use cases, REST API endpoints, and data storage approach.

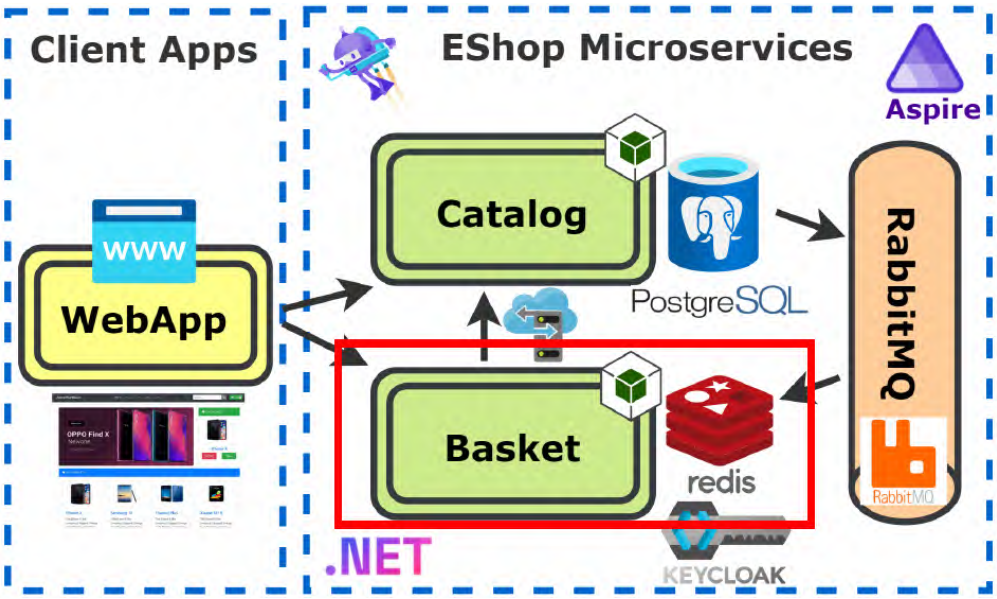


Figure 7.1 – Architecture diagram showing WebApp, Catalog, and Basket microservices, with Basket connected to Redis and highlighted in red

Domain models

The primary domain models of the Basket microservice are `ShoppingCart` and `ShoppingCartItem`. The `ShoppingCart` entity includes the user name, a list of shopping cart items, and total price.

```
public class ShoppingCart
{
    public string UserName { get; set; } = default!;
    public List<ShoppingCartItem> Items { get; set; } = new();
    public decimal TotalPrice => Items.Sum(x => x.Price * x.Quantity);
}
```

The `ShoppingCartItem` entity includes the quantity, color, product ID, price, and product name.

```
public class ShoppingCartItem
{
    public int Quantity { get; set; }
    public string Color { get; set; } = default!;
    public int ProductId { get; set; }
    // will come from Catalog microservice
    public decimal Price { get; set; }
    public string ProductName { get; set; } = default!;
}
```

The `ShoppingCartItem`'s `price` and `product` name come from the `Catalog` microservice.

Application use cases

The main use cases of the Basket microservice are as follows:

- `GetBasket`
- `UpdateBasket` (create or update, upsert)
- `DeleteBasket`
- `UpdateBasketItemProductPrices`

> **NOTE**
>
> In this chapter, we will focus on implementing the Basket **CRUD** operations. The communication-related logic, including calling the Catalog microservice to retrieve the latest product information and update basket item prices, will be implemented in *Chapter 8*, where we introduce synchronous communication between the Basket and Catalog microservices.

Before we update the shopping cart, let's call the Catalog microservice's `GetProductById` endpoint synchronously to get the latest product information and set the price and product name when adding an item.

REST API endpoints

To support these use cases, the Basket microservice exposes the following REST API endpoints:

- HTTP `GET/basket/{userName}` to fetch a specific basket
- HTTP `POST/basket` to create or update a basket (upsert with Redis)
- HTTP `DELETE/basket/{userName}` to remove a basket

Data storage

Basket data is stored in a Redis distributed cache. Unlike the Catalog microservice, which persists product information in PostgreSQL, the Basket microservice uses Redis to provide fast access to shopping cart state. Later in the chapter, we'll configure Redis through Aspire and access it from the Basket service using the `IDistributedCache` interface.

With the domain models, use cases, API surface, and data storage approach established, we can now turn to the technical design of the Basket microservice.

Technical analysis of the Basket microservice

The technical design determines how the Basket domain requirements will be translated into code. We'll keep the implementation deliberately simple, use established ASP.NET Core patterns, integrate Redis through Aspire, and organize the project so that domain models, business logic, API endpoints, and external service communication have clear responsibilities.

Fundamental patterns and principles

To guide the development of the Basket microservice, we will follow the **KISS** principle (keep it simple, stupid) principle and avoid introducing unnecessary architectural patterns. Instead, we will focus on integration and orchestration with Aspire while implementing the required use cases directly.

We will still rely on fundamental ASP.NET Core patterns, including dependency injection and minimal APIs and routing, together with data-access abstractions for interacting with Redis.

Redis will serve as the **distributed cache** for the Basket microservice. Aspire will provide the hosting integration needed to define and orchestrate the Redis resource. The Basket project will use the Redis client integration to connect to that resource through `IDistributedCache`. We'll configure both integrations in detail when we build the Redis backing service later in the chapter.

Project folder structure

With the overall technical approach established, the next step is to organize the Basket microservice so that each responsibility has a clear place in the codebase.

The Basket microservice is organized into several key folders: `Models`, `Services`, `Endpoints`, and `ApiClients`:

- The `Models` folder contains the domain entities, including `ShoppingCart` and `ShoppingCartItem`.
- The `Services` folder contains the business logic for managing basket operations and interacting with Redis.
- The `Endpoints` folder contains the minimal **API** endpoints that expose Basket functionality to client applications and delegate business operations to the service layer.
- The `ApiClients` folder contains **HTTP** client objects used for synchronous communication with the Catalog microservice.

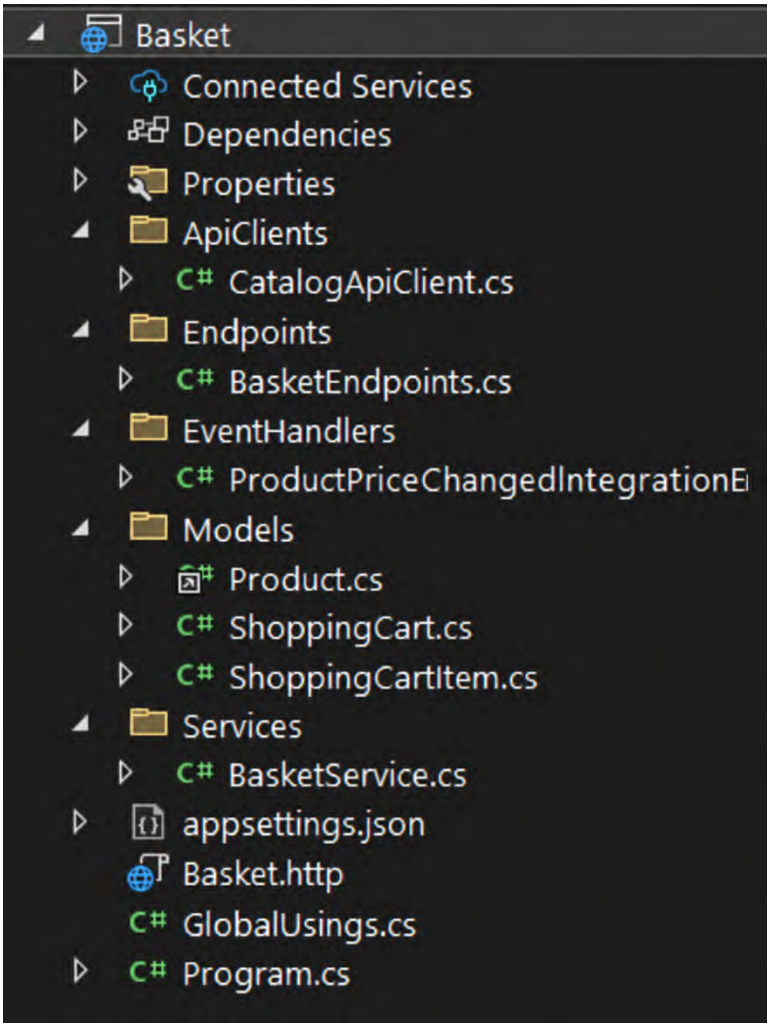


Figure 7.2 – Visual Studio Solution Explorer showing the Basket microservice folder structure, with folders for ApiClients, Endpoints, EventHandlers, Models, and Services

With the project organization defined, you are ready to create the Basket microservice and add it to the Aspire orchestration environment.

Creating the Basket microservice project with Aspire

We will now use Visual Studio to create the Basket microservice as a new ASP.NET Core Web API project and add it to the Aspire orchestration environment.

1. Open Visual Studio and locate the `services` folder in your solution. Right-click the `services` folder to open the context menu.
2. Select **Add | New Project** from the menu.
3. From the **Add a new project** dialog, choose the **ASP.NET Core Web API** template to represent your microservice.
4. Select the **Next** button.
5. In the **Configure your new project** window, enter the project name as `Basket`.
6. Select **Next** to continue to the **Additional information** screen.
7. In the **Additional information** window, leave most configurations at their defaults: Make sure to check **Enlist in Aspire orchestration** at the bottom.
    - **Framework**: **.NET 10.0 (Long Term Support)**
    - **Authentication type**: **None**
    - **Configure for HTTPS**: **Checked**
    - **Enable container support**: **Unchecked**
8. Select the **Create** button to generate the Basket microservice project under the `services` folder.

We have successfully added the Basket microservice.

Next, let's examine how Aspire integrates and orchestrates the Basket microservice within the distributed application.

Start by opening the `Program.cs` file in the `AppHost` project. When you enlist the Basket microservice in Aspire orchestration, Aspire automatically adds the following line:

```
builder.AddProject<Projects.Basket>("basket");
```

This registers the Basket microservice with the distributed application so that Aspire can start and manage it as part of the overall application environment.

Next, double-click the Basket microservice project file (`Basket.csproj`). You will see that Aspire has also added a reference to the `ServiceDefaults` project:

```
<ItemGroup>
<ProjectReference Include="..\ServiceDefaults\ServiceDefaults.csproj" />
</ItemGroup>
```

This reference gives the Basket microservice access to the shared configuration and cross-cutting defaults defined in ServiceDefaults.

If you open `Program.cs` in the Basket microservice, you will see that Aspire has added `builder.AddServiceDefaults()` and `app.MapDefaultEndpoints()` to apply the shared configuration provided by the `ServiceDefaults` project.

The following code in `Program.cs` demonstrates these default configurations:

```
var builder = WebApplication.CreateBuilder(args);
builder.AddServiceDefaults();
// Add services to the container.
var app = builder.Build();
app.MapDefaultEndpoints();
// Configure the HTTP request pipeline.
app.UseHttpsRedirection();
```

This is the standard approach used by Aspire to apply common configuration and cross-cutting concerns to projects that participate in the distributed application.

As you develop the Basket microservice, organize the statements in `Program.cs` according to their purpose. Keep `builder.AddServiceDefaults()` together with the other service registrations before `builder.Build()`. After building the application, keep `app.MapDefaultEndpoints()` with the other endpoint mappings and HTTP request pipeline configuration.

With the Basket microservice registered and its default configuration in place, you can run the distributed application and verify that Aspire is orchestrating the new service:

1. Right-click the `AppHost` project and set it as the startup project.
2. Make sure Docker Desktop is running.
3. Run the Aspire application.

You can confirm that the Basket microservice is running from the Aspire dashboard; you will see the `basket` project as a resource in the distributed application, as shown in the following figure:

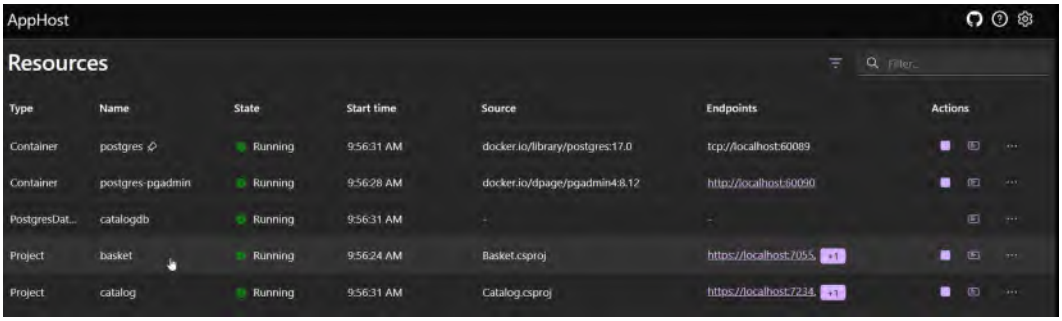


Figure 7.3 – Basket microservice in Resources view

With the Basket microservice running under Aspire orchestration, the next step is to implement its domain models.

Creating the Basket domain models

To implement the shopping cart domain entity models in your basket microservice, begin by organizing your project in Visual Studio.

1. Locate the `services` folder and open the Basket microservice project.
2. Right-click the `Basket` project and add a new folder named `Models` to represent your domain models.
3. Add a new class file named `ShoppingCart.cs` to the `Models` folder.
   The following code defines the `ShoppingCart` domain entity with its members:

```
namespace Basket.Models;
public class ShoppingCart
{
    public string UserName { get; set; } = default!;
    public List<ShoppingCartItem> Items { get; set; } = new();
    public decimal TotalPrice => Items.Sum(x => x.Price * x.Quantity);
}
```

   The properties serve the following purposes:

   - `UserName` identifies the user who owns the basket and will be used as the key in the Redis distributed cache.

- `Items` contains a list of `ShoppingCartItem` objects, each representing a product in the basket.
- `TotalPrice` calculates the total basket price by multiplying the price of each item by its quantity and summing the results.

4. Next, add another class file named `ShoppingCartItem.cs` to the `Models` folder. The following code defines the `ShoppingCartItem` entity and its members:

```
namespace Basket.Models;
public class ShoppingCartItem
{
    public int Quantity { get; set; }
    public string Color { get; set; } = default!;
    public int ProductId { get; set; }
    // will comes from catalog microservice
    public decimal Price { get; set; }
    public string ProductName { get; set; } = default!;
}
```

The properties serve the following purposes:

- `Quantity` indicates how many units of the product are in the cart.
- `Color` stores the selected color of the product.
- `ProductId` references the product's ID from the Catalog microservice.
- `Price` is retrieved from the Catalog microservice through synchronous communication so that the Basket microservice can use the latest product price when updating the basket.
- `ProductName` stores the product name in the basket so it can be displayed without retrieving the same information from the Catalog microservice each time.

With these steps, you have completed the domain layer by developing the `ShoppingCart` and `ShoppingCartItem` entities under the `Models` folder. The next step is to configure the backing service that will store basket data.

Integrating Redis as a distributed cache with Aspire

The Basket microservice uses Redis as a distributed cache, with Aspire handling the provisioning and orchestration of the Redis resource.

To enable distributed caching for your Basket microservice, you need to integrate Redis using Aspire's **infrastructure-as-code** approach. The integration takes place in two stages. First, you will add Redis as a backing service in the AppHost project and connect it to the Basket microservice. You will then configure the Basket project to access Redis through the Aspire Redis client integration.

Connecting the Basket microservice to Redis using StackExchange.Redis

Let's begin by opening Visual Studio and navigating to the `AppHost` project under the `aspire` folder. The next steps involve installing the Redis Nuget package to activate Redis as a distributed cache:

1. Right-click the **Dependencies** node under the `AppHost` project.
2. Select **Add Aspire package**
3. Search for `Aspire hosting Redis` in the NuGet Package Manager.
4. Select the **Aspire.Hosting.Redis** package from the list of available Aspire hosting integration packages.
5. Install the **Aspire.Hosting.Redis** Nuget package into the `AppHost` project.

This process ensures that the Redis container can be orchestrated and managed by Aspire with minimal configuration.

After installing the package, you can confirm that the package's presence in your project file. The following code block shows the updated `.csproj` file for the `AppHost` project with the Redis package reference included:

```
<PackageReference Include="Aspire.Hosting.Redis" Version="13.0.0" />
```

Now, open the `Program.cs` file in the `AppHost` project to define Redis as a backing service. You will add the Redis container resource and configure its persistence and monitoring options.

The initial version of the `Program.cs` file, before adding Redis, contains the existing PostgreSQL backing service and the Catalog and Basket projects:

```
var builder = DistributedApplication.CreateBuilder(args);
// Backing Services
var postgres = builder
    .AddPostgres("postgres")
    .WithPgAdmin()
    .WithDataVolume()
```

```
    .WithLifetime(ContainerLifetime.Persistent);
var catalogDb = postgres.AddDatabase("catalogdb");
// Projects
var catalog = builder
    .AddProject<Projects.Catalog>("catalog")
    .WithReference(catalogDb)
    .WaitFor(catalogDb);
builder.AddProject<Projects.Basket>("basket");
builder.Build().Run();
```

To add Redis as a backing service, define a Redis resource named `cache` under the backing services section:

```
var cache = builder
    .AddRedis("cache")
    .WithRedisInsight()
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
```

Here, `AddRedis("cache")` adds the Redis resource to the distributed application. `WithRedisInsight()` enables Redis Insight for inspecting and monitoring Redis data, while `WithDataVolume()` and `WithLifetime(ContainerLifetime.Persistent)` configure persistent storage and container lifetime so that the Redis data can persist across application restarts.

To connect your Basket microservice to the cache, you add the project reference and wait for the cache to be ready. This is done using the `WithReference(cache)` and `WaitFor(cache)` methods:

```
var basket = builder
    .AddProject<Projects.Basket>("basket")
    .WithReference(cache)
    .WaitFor(cache);
```

The `WithReference(cache)` method injects environment variables and connection information for Redis into the Basket microservice. This eliminates the need for a separate Docker file or manual IP configuration, as Aspire automatically provides environment variables containing the Redis host, port, and password. The `WaitFor(cache)` ensures that the Redis resource is ready before the Basket project starts.

After configuring these settings, set your startup project to AppHost and run the application. Make sure Docker Desktop is running in the background to activate the newly added backing services, which include both the cache and cache-insight containers.

You can confirm that the cache and cache-insight resources are running by checking the **Resources** tab in the **AppHost** dashboard.

The following figure shows the AppHost dashboard with the cache and cache-insight containers running, along with other services and projects in the distributed environment.

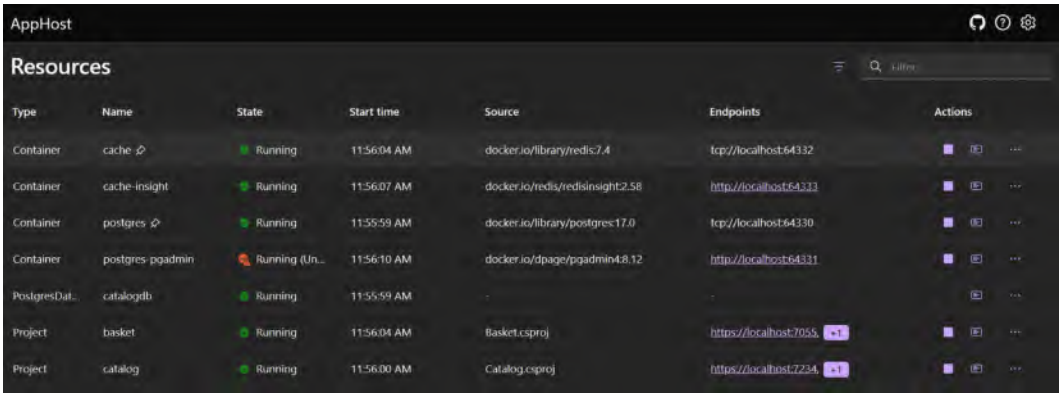


Figure 7.4 – AppHost dashboard showing the running cache, cache-insight, PostgreSQL, and application project resources in the Aspire distributed environment

If you select the `basket` project in the AppHost dashboard and scroll down, you can browse the environment variables and confirm that connection information for the `cache` resource (`ConnectionStrings__cache`) has been added.

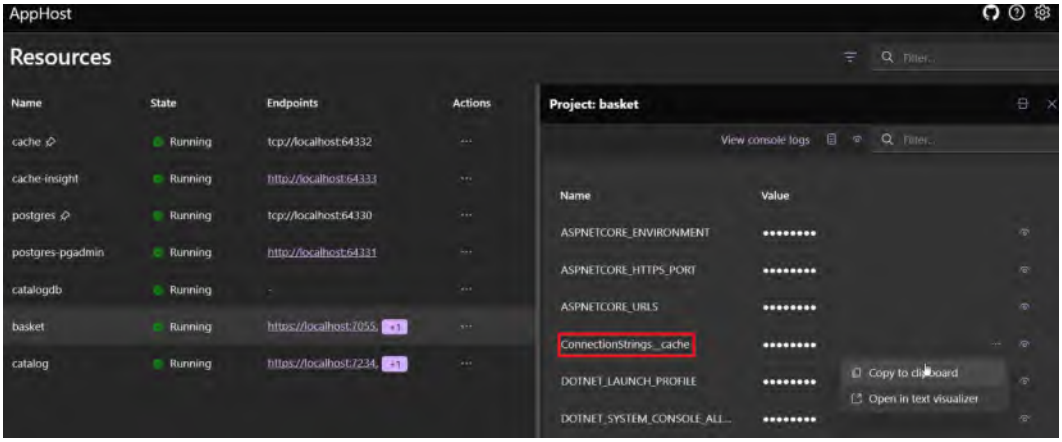


Figure 7.5 – Environment variables

Opening this in **text visualizer** will show that the Redis container is injected successfully as a distributed cache using the reference keyword.

As a result, you have completed the hosting integration, added a Redis container, and enabled your basket microservice to consume the Redis container as a distributed cache.

Redis client integration

You are now ready to perform the **client integration** for your Basket microservice, connecting it to the Redis cache. The client integration requires installing the correct package, registering it in your project, and configuring your application to use Redis as a distributed cache.

1. In **Solution Explorer**, right-click on the **Dependencies** node of the Basket microservice and select the **Add Aspire package** button.
2. In the package selection dialog, filter and select the `Aspire.StackExchange.Redis.DistributedCaching` package. Be sure to choose the distributed caching package, not any other similarly named package.
3. Install this package into the Basket microservice project.

After installing the package, you can confirm its presence by opening the Basket microservice's project file. The following code in your `.csproj` file will show the package reference:

```
<PackageReference Include="Aspire.StackExchange.Redis.DistributedCaching"
Version="13.0.0" />
```

This package sets up the `StackExchange.Redis` client library according to Aspire standards, automatically using environment variables from the `AppHost`.

Next, configure your `Program.cs` file in the Basket microservice to register the Redis client. Go to the section where you add services to the dependency injection container. After the call to `AddServiceDefaults()`, add the following line to register the distributed cache:

```
builder.AddRedisDistributedCache(connectionName: "cache");
```

This line instructs the Basket microservice to connect to the cache resource declared in the `AppHost` project. The connection name `cache` matches the Redis container configuration in AppHost's `Program.cs`, ensuring the correct resource is consumed.

Aspire will look for the environment variable `ConnectionStrings__cache` in the Basket microservice and configure the client object using this connection information. This approach eliminates the need to manually manage Docker container Ips, ports, or custom application settings for secrets, as Aspire handles all of this for you.

The Redis container declared as `cache` in the AppHost's `Program.cs` file:

```
var cache = builder
    .AddRedis("cache")
    .WithRedisInsight()
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
```

With this registration complete, you are now ready to inject the `IDistributedCache` interface into your service classes for use in your business logic.

The following diagram illustrates how the Basket microservice integrates with Redis using Aspire, showing the flow from hosting integrations to client integrations and the relevant package and configuration:

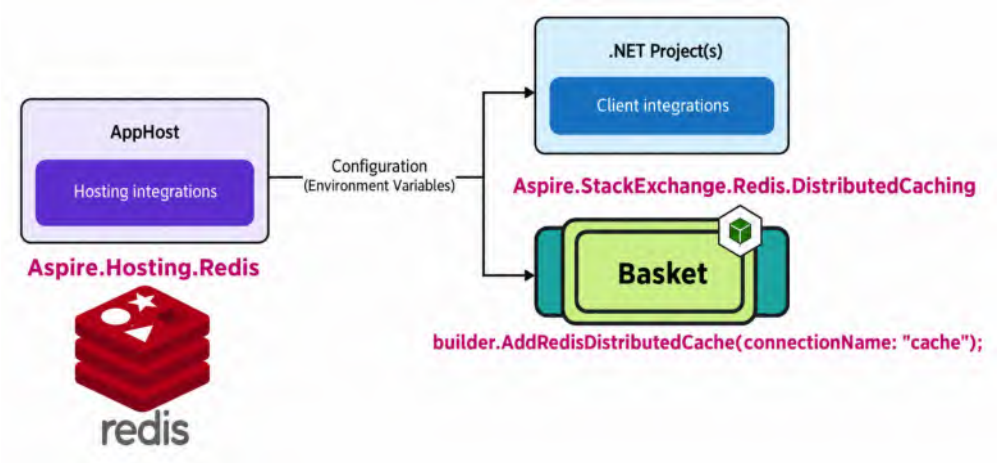


Figure 7.6 – Architecture diagram showing AppHost hosting integrations, Redis, Aspire.Hosting.Redis, Aspire.StackExchange.Redis.DistributedCaching, and the Basket microservice with builder.AddRedisDistributedCache(connectionName: "cache")

Now that the Basket microservice is connected to Redis, the next step is to implement the business logic that reads, updates, and deletes basket data.

Developing the Basket service class with caching logic

In this section, you will implement the business logic for your Basket microservice by creating a service class that interacts with Redis through the injected **IDistributedCache** interface. This class will reside in the `Services` folder and encapsulate all caching operations for basket management.

1. Open Visual Studio and navigate to the Basket microservice project.
2. Right-click the project, and add a new folder named `Services`.
3. Inside the `Services` folder, add a new class named `BasketService.cs`.

The following code defines the `BasketService` class, which uses a *primary constructor* to inject the `IDistributedCache` dependency:

```
using Basket.Models;
using Microsoft.Extensions.Caching.Distributed;
using System.Text.Json;
namespace Basket.Services;
public class BasketService(IDistributedCache cache)
{
    public async Task<ShoppingCart?> GetBasket(string userName)
    {
        var basket = await cache.GetStringAsync(userName);
        return string.IsNullOrEmpty(basket) ? null :
JsonSerializer.Deserialize<ShoppingCart>(basket);
    }
    public async Task UpdateBasket(ShoppingCart basket)
    {
        await cache.SetStringAsync(basket.UserName,
JsonSerializer.Serialize(basket));
    }
    public async Task DeleteBasket(string userName)
    {
        await cache.RemoveAsync(userName);
    }
}
```

This class provides three main methods:

- `GetBasket`: Retrieves the basket for a given username from Redis, deserializing the JSON string into a `ShoppingCart` object if it exists.
- `UpdateBasket`: Performs an **upsert** operation by serializing the `ShoppingCart` object and storing it in Redis under the username key.
- `DeleteBasket`: Removes the basket entry for the specified username from Redis.

To simplify the use of namespaces across your project, you should add global `using` statements. In the Basket microservice project, add a new class named `GlobalUsings.cs`, and add the following global `using` statements to this file:

```
global using Basket.Models;
global using Basket.Services;
```

Save the file and return to `BasketService.cs` to remove the now-redundant `using` statements for `Basket.Models` and `Basket.Services`.

After implementing the `BasketService` class, you need to register it with the ASP.NET Core dependency injection container so it can be injected into your endpoint classes.

1. Open `Program.cs` in the Basket microservice project.
2. Locate the section where services are added to the container.
3. After the line registering the distributed cache, add the following line to register the `BasketService` with a scoped lifetime:

   ```
   builder.Services.AddScoped<BasketService>();
   ```

Registering the service with a *scoped lifetime* ensures that each incoming request receives a fresh instance of `BasketService`, which is the recommended practice for handling business logic in ASP .NET Core applications.

You are now ready to inject your basket service classes into a folder called `Endpoints` to handle your business logic.

Developing the Basket API endpoints with ASP.NET Core Minimal APIs

To develop Basket API endpoints with ASP .NET Core Minimal APIs, begin by organizing your presentation layer. In Visual Studio, navigate to the Basket microservice project, right-click, and add a new folder named `Endpoints`. This folder will represent your presentation layer.

Inside the `Endpoints` folder, create a new class named `BasketEndpoints.cs`. We'll define this as a static class and implement an **extension method** for mapping the Basket endpoints:

```
namespace Basket.Endpoints
{
    public static class BasketEndpoints
    {
    }
}
```

Next, add a static extension method called `MapBasketEndpoints` to the `BasketEndpoints` class. This method extends `IEndpointRouteBuilder` and is responsible for mapping your basket endpoints:

```
public static void MapBasketEndpoints(this IEndpointRouteBuilder app)
{
    var group = app.MapGroup("basket");
    group.MapGet("/{userName}", async (string userName, BasketService service) =>
    {
        var shoppingCart = await service.GetBasket(userName);
        if (shoppingCart is null)
        {
            return Results.NotFound();
        }
        return Results.Ok(shoppingCart);
    })
    .WithName("GetBasket")
    .Produces<ShoppingCart>(StatusCodes.Status200OK)
    .Produces(StatusCodes.Status404NotFound);
```

```
    group.MapPost("/", async (ShoppingCart shoppingCart, BasketService service) =>
    {
        await service.UpdateBasket(shoppingCart);
        return Results.CreatedAtRoute("GetBasket", new { userName =
shoppingCart.UserName }, shoppingCart);
    })
    .WithName("UpdateBasket")
    .Produces<ShoppingCart>(StatusCodes.Status201Created);
    group.MapDelete("/{userName}", async (string userName, BasketService service)
=>
    {
        await service.DeleteBasket(userName);
        return Results.NoContent();
    })
    .WithName("DeleteBasket")
    .Produces(StatusCodes.Status204NoContent);
}
```

The `MapBasketEndpoints` method groups all basket-related endpoints under the `"basket"` route. It exposes three main HTTP operations:

- `GET /basket/{userName}`: Retrieves the shopping cart for a given user. If the cart does not exist, it returns a `404 Not Found` response; otherwise, it returns a `200 OK` response with the shopping cart as **JSON**.
- `POST /basket`: Accepts a shopping cart as a JSON body and updates or creates the basket in Redis using the `UpdateBasket` method. This operation returns a `201 Created` response with the updated shopping cart.
- `DELETE /basket/{userName}`: Deletes the basket for the specified user and returns a `204 No Content` response.

Together, these endpoints form the HTTP interface for the basket operations implemented in `BasketService`.

With the endpoints defined, the final step is to register them with the application so that ASP.NET Core can expose the routes.

Registering the Basket endpoints

To complete the registration of your Basket endpoints in the HTTP pipeline, you need to update the `Program.cs` file of the Basket microservice so that your HTTP `GET`, `POST`, and `DELETE` routes are mapped and exposed properly.

Open `Program.cs` in the Basket microservice. The `app.MapDefaultEndpoints()` call provided through Aspire's `ServiceDefaults` is already present. Add the following call immediately after that line to register the Basket endpoints:

```
app.MapBasketEndpoints();
```

When you add this line, Visual Studio might automatically insert a `using` statement for `Basket.Endpoints` at the top of your `Program.cs` file. To keep your code organized and to make the `Basket.Endpoints` namespace available throughout the project, move this `using` statement into your `GlobalUsings.cs` file under the Basket microservice.

The updated `GlobalUsings.cs` file should look like this:

```
global using Basket.Models;
global using Basket.Services;
global using Basket.Endpoints;
```

You can then remove the corresponding local `using Basket.Endpoints;` statement from `Program.cs` if Visual Studio added one automatically.

Return to your `Program.cs` file and confirm that the Basket endpoints are now registered after building the application and before any other middleware. This step is critical; if you skip registering the Basket endpoints, none of your basket-related routes will be available, and you will receive `404 Not Found` errors when trying to access them.

You should also clean up your `Program.cs` file by removing any default code related to weather forecasts or summaries that the ASP.NET Core template generated. Delete these pieces of code up to the `app.Run()` line, because you do not need them for your Basket microservice.

The final version of your `Program.cs` file for the basket microservice should look like this:

```
using Basket.Endpoints;
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.AddServiceDefaults();
```

```
builder.AddRedisDistributedCache(connectionName: "cache");
builder.Services.AddScoped<BasketService>();
var app = builder.Build();
// Configure the HTTP request pipeline.
app.MapDefaultEndpoints();
app.MapBasketEndpoints();
app.UseHttpsRedirection();
app.Run();
```

By following these steps, your basket endpoints are now properly registered and exposed as part of the ASP.NET Core HTTP pipeline, ready to handle requests for shopping cart operations. You are now ready to test the Basket microservice and verify its integration with Redis as a distributed cache using Visual Studio.

Testing the Basket microservice operations with Redis

With the Basket service and its endpoints in place, you can now test the complete request flow and verify that shopping-cart data is persisted correctly in Redis. We will use the `Basket.http` file to define and invoke HTTP `GET`, `POST`, and `DELETE` operations against your Basket endpoints.

To begin testing the Basket microservice endpoints, open Visual Studio and navigate to the Basket microservice project. There you will find the `Basket.http` file under the Basket project in **Solution Explorer**.

First, we'll need to set the correct host address for the Basket microservice. To do this

1. Set the `AppHost` project as the **startup project**.
2. Make sure Docker Desktop is running.
3. Run the Aspire distributed application.
4. In the **Aspire dashboard**, confirm that the **Basket project** and the **cache** and **cache-insight** resources are running.
5. Copy the **HTTPS** endpoint for the Basket microservice and use it to set the base address in `Basket.http`, for example,

   ```
   @Basket_HostAddress = https://localhost:7055/basket
   ```

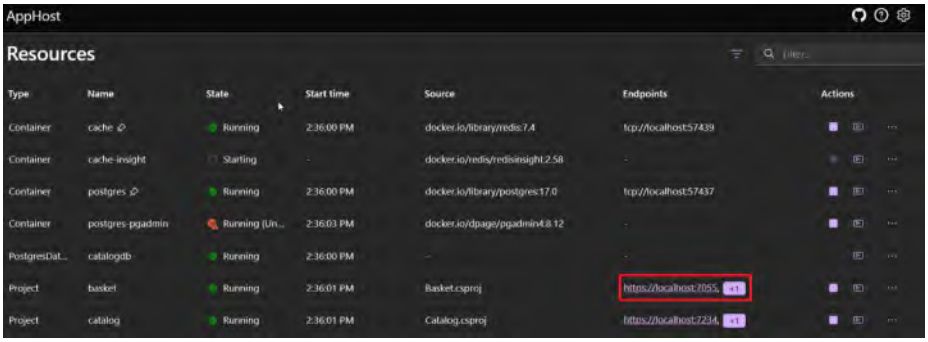


Figure 7.7 – Copy the Basket microservice endpoint

Next, we will group all endpoints under the `/basket` path by updating the host address variable:

```
@Basket_HostAddress = https://localhost:7055/basket
```

> **NOTE**
>
> The port assigned to your Basket microservice might differ, so use the endpoint shown in your **Aspire dashboard**.

Next, we'll define the requests that we'll use to exercise the Basket API. Remove the default weather forecast endpoint and add your own when first creating the `GET` endpoint.

```
@Basket_HostAddress = https://localhost:7055/basket
###
GET {{Basket_HostAddress}}/swn
Accept: application/json
###
POST {{Basket_HostAddress}}
Content-Type: application/json
{
  "UserName": "swn",
  "Items": [
    {
      "Quantity": 2,
      "Color": "Red",
      "Price": 500,
      "ProductId": 1,
```

```
      "ProductName": "Solar Powered Flashlight"
    },
    {
      "Quantity": 1,
      "Color": "Blue",
      "Price": 500,
      "ProductId": 2,
      "ProductName": "Hiking Poles"
    }
  ]
}
###
DELETE {{Basket_HostAddress}}/swnGET {{Basket_HostAddress}}/swn
Accept: application/json
```

After defining these endpoints, you are ready to invoke them one by one and verify that the Basket microservice reads from and writes to Redis as expected.

Set the AppHost project as the **startup project** in Visual Studio, make sure Docker Desktop is running, and run the Aspire distributed application.

Once the application starts, open the AppHost dashboard. You should see the `basket` and `catalog` projects along with the Redis and PostgreSQL resources running in the distributed environment.

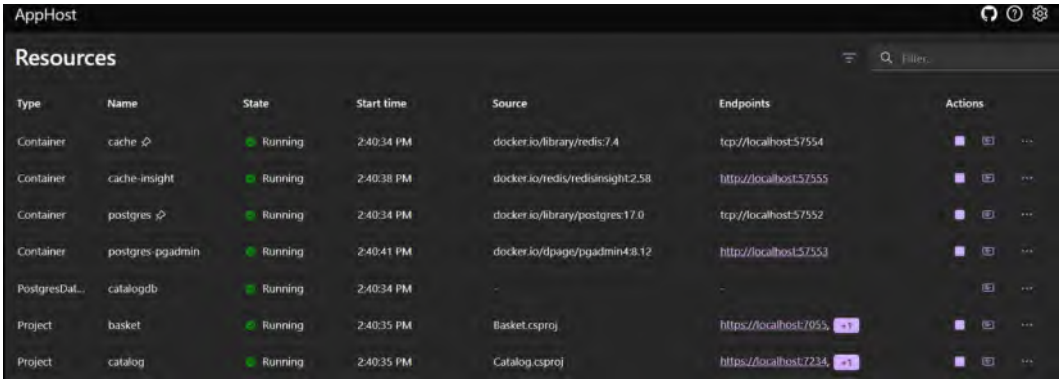


*Figure 7.8 – The AppHost **Resources** page with all services running, including the* `cache` *and* `cache-insight` *containers*

You will now use the `Basket.http` file to send requests to the Basket endpoints and **Redis Insight** to observe how the corresponding Basket data is stored in Redis

1. Open **Redis Insight** by selecting the `cache-insight` resource in the **AppHost** dashboard. This opens the Redis Insight interface, where you can inspect the keys stored in the Redis cache. At this point, the cache should not contain a basket for the user you are about to test.

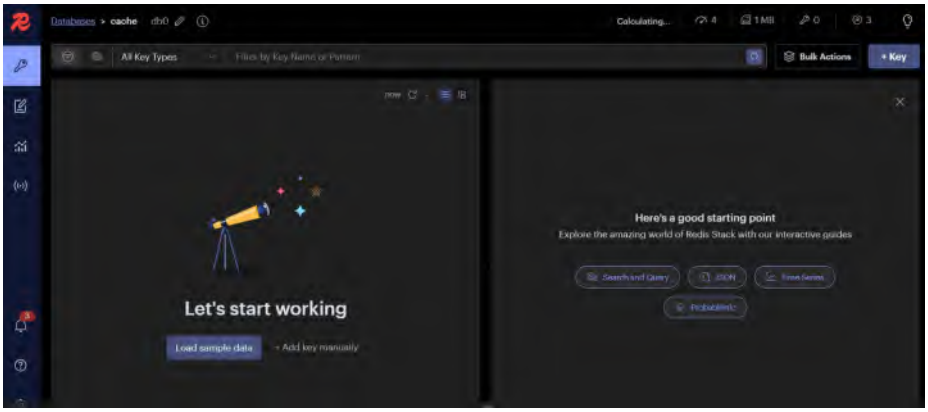


Figure 7.9 – Redis Insight interface, ready for you to browse or add keys

2. In Visual Studio, open the `Basket.http` file to prepare your HTTP requests for testing the Basket microservice endpoints.

```
@Basket_HostAddress = https://localhost:7055/basket
###
GET {{Basket_HostAddress}}/swn
Accept: application/json
###
POST {{Basket_HostAddress}}
Content-Type: application/json
{
  "UserName": "swn",
  "Items": [
    {
      "Quantity": 2,
      "Color": "Red",
      "Price": 500,
      "ProductId": 1,
      "ProductName": "Solar Powered Flashlight"
```

```
    },
    {
      "Quantity": 1,
      "Color": "Blue",
      "Price": 500,
      "ProductId": 2,
      "ProductName": "Hiking Poles"
    }
  ]
}
###
DELETE {{Basket_HostAddress}}/swn
```

The port assigned to the Basket microservice might differ, so use the HTTPS endpoint shown in your **AppHost** dashboard.

3. Set a breakpoint in the `BasketEndpoints.cs` file at the start of the `GET` endpoint logic to observe the request flow:

```
var shoppingCart = await service.GetBasket(userName);
if (shoppingCart is null)
{
    return Results.NotFound();
}
return Results.Ok(shoppingCart);
```

4. Send an HTTP `GET` request for the username `swn` using the prepared request in `Basket.http`. The debugger will hit your breakpoint, and because there is no cache entry yet, the result will be a `404 Not Found`.

5. Next, send an HTTP `POST` request to create a new shopping cart for the username `swn` with two items. Do this without debugging to let the request complete.
The request returns `201 Created` together with the newly created shopping cart:

```
{
    "userName": "swn",
    "items": [
    {
        "quantity": 2,
        "color": "Red",
        "productId": 1,
```

```
        "price": 500,
        "productName": "Solar Powered Flashlight"
    },
    {
        "quantity": 1,
        "color": "Blue",
        "productId": 2,
        "price": 500,
        "productName": "Hiking Poles"
    }],
    "totalPrice": 1500
}
```

6. Return to **Redis Insight** and refresh the **cache explorer**. You will see a new cache item with the key `swn`, representing the shopping cart **data** stored as a JSON object. You can expand the `data` **Field** to inspect the full JSON representation of the shopping cart, including its items and their properties.

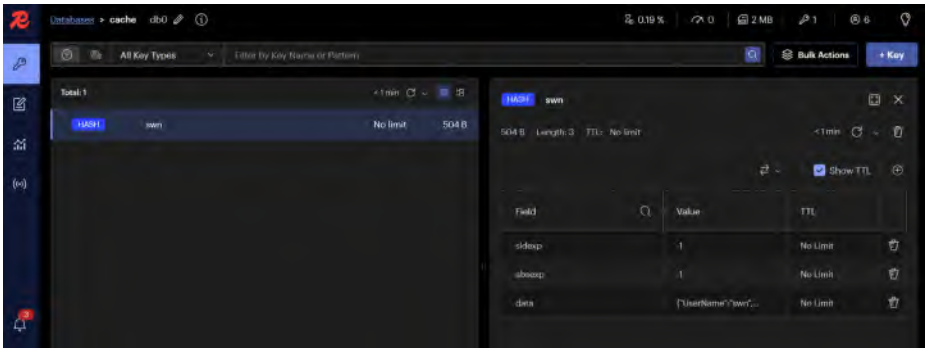


Figure 7.10 – Redis Insight showing the swn key and the shopping-cart data stored in Redis

7. Let's go back to Visual Studio and send the `GET` request for `swn` again:

```
GET {{Basket_HostAddress}}/swn
Accept: application/json
```

When you send this request, the basket item is retrieved from the cache as a JSON string, deserialized into a `ShoppingCart` object, and returned. You receive a `200 OK` response with the shopping cart data in JSON format.

8. Finally, send an HTTP `DELETE` request for the username `swn` to remove the shopping cart key from Redis:

```
DELETE {{Basket_HostAddress}}/swn
Content-Type: application/json
```

After sending the `DELETE` request, you receive a `204 No Content` response, indicating successful deletion.

To verify, refresh the Redis Insight cache explorer. The `swn` key should no longer be present, confirming that the shopping cart has been successfully deleted.

You have now tested the complete Basket API life cycle: the initial `GET` confirmed that no basket existed, the `POST` created and persisted a basket in Redis, the subsequent `GET` retrieved it, and the `DELETE` removed it.

Summary

In this chapter, we developed the Basket microservice and integrated it into the eShop distributed application using Aspire. We began by analyzing the Basket domain and defining the `ShoppingCart` and `ShoppingCartItem` models, along with the core use cases and REST API endpoints. We then created the Basket project, organized its responsibilities across the `Models`, `Services`, `Endpoints`, and `ApiClients` folders, and examined how `ServiceDefaults` and the AppHost project support orchestration across the application.

We then configured Redis as the Basket microservice's distributed cache. On the hosting side, we added a Redis resource to the AppHost project and connected the Basket project to it using `WithReference(cache)`. On the client side, we installed the Redis distributed caching integration and registered it with `AddRedisDistributedCache`. Using the injected `IDistributedCache`, we implemented `BasketService` to retrieve, create or update, and delete shopping carts, and exposed these operations through ASP.NET Core Minimal APIs.

Finally, we tested the Basket endpoints and used **Redis Insight** to verify that basket data was created, retrieved, and removed from Redis as expected. At this stage, however, product information such as the price and product name still comes from the basket request. In the next chapter, we will connect the Basket and Catalog microservices so that the Basket service can retrieve the latest product information directly from Catalog before storing the basket in Redis.

8

Synchronous Communication Between Catalog and Basket Using Aspire Service Discovery

In the previous chapter, we developed the Basket microservice, configured Redis as its distributed cache, and exposed endpoints for retrieving, updating, and deleting shopping carts. Although the Basket service can now persist shopping-cart data, the product price and name are still supplied as part of the basket request. In a distributed e-commerce application, this information should come from the Catalog microservice so that the basket uses the latest product data.

In this chapter, we will establish **synchronous communication** between the Basket and Catalog microservices using **Aspire Service Discovery**. When you create or update a basket, the Basket microservice will call Catalog over **HTTP** to retrieve the current price and product name for each item before persisting the updated basket to Redis.

We will begin by examining the communication flow and configuring a project-to-project reference from Basket to Catalog using `WithReference(catalog)`. This allows Aspire to provide service discovery information so that Basket can address Catalog by its logical service name rather than relying on hard-coded hostnames or ports.

Next, we will make the Product model available to the Basket microservice, create a typed `CatalogApiClient`, and register it using `AddHttpClient<CatalogApiClient>`. We will then inject this client into `BasketService` and update the `UpdateBasket` method so that it retrieves the latest product information from Catalog before writing the basket to Redis. Finally, we will test the complete flow and use the Aspire dashboard to trace the synchronous calls between the two microservices.

By the end of the chapter, you will have connected the Catalog and Basket microservices through Aspire Service Discovery, enabling the Basket microservice to enrich shopping-cart items with up-to-date product information before persisting them to Redis.

In this chapter, we'll be covering the following topics:

- Architecture of synchronous communication between Catalog and Basket microservices
- Leveraging Aspire Service Discovery for microservice communication
- Sharing the Product Entity between Catalog and Basket microservices
- Implementing the Catalog API client for HTTP communication
- Testing synchronous communication and end-to-end data flow

Architecture of synchronous communication between Catalog and Basket microservices

In this chapter, we are going to establish synchronous communication between the Catalog and Basket microservices using Aspire Service Discovery.

The Basket microservice needs product information owned by the Catalog microservice. Rather than relying on the product name and price supplied in the incoming API request, Basket retrieves these values directly from Catalog whenever you create or update a basket.

The flow begins when a client sends an HTTP `POST` request to the Basket microservice. Basket iterates through the shopping-cart items and, for each product ID, calls the Catalog microservice synchronously. Catalog returns the current product information, and Basket updates the item's price and product name before writing the completed shopping cart to Redis.

The following diagram shows the synchronous communication path between the Basket and Catalog microservices, with the Basket sending HTTP requests to the Catalog to retrieve product details.

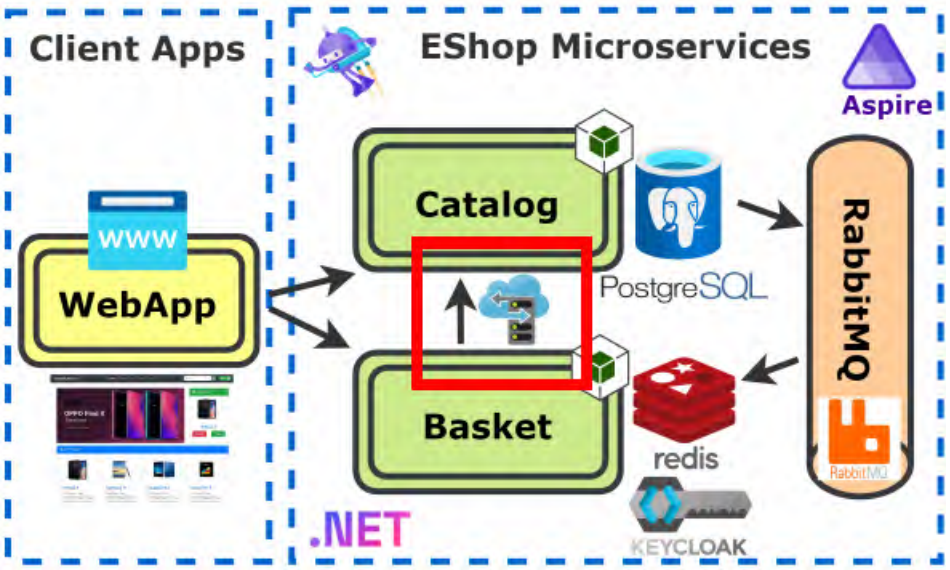


Figure 8.1 – Architecture diagram showing WebApp and Catalog and Basket microservices, with Basket connected to Redis, Catalog connected to PostgreSQL, and both orchestrated by Aspire

With the communication flow established, the next question is how Basket locates Catalog without relying on a fixed hostname or port.

Leveraging Aspire Service Discovery for microservice communication

With Aspire Service Discovery, services do not need to rely on hardcoded URLs. Instead, they can communicate using *logical service names*. Aspire provides the service discovery configuration that allows the Basket microservice to locate the Catalog microservice by name at runtime, without hardcoding its host address or port. To make Catalog discoverable from Basket, you first establish a project-to-project reference in the AppHost project.

Open `Program.cs` in `AppHost`. At this point, the application already defines PostgreSQL and Redis as backing services and registers the Catalog and Basket projects:

```
var builder = DistributedApplication.CreateBuilder(args);
// Backing Services
var postgres = builder
    .AddPostgres("postgres")
    .WithPgAdmin()
    .WithDataVolume()
```

```
    .WithLifetime(ContainerLifetime.Persistent);
var catalogDb = postgres.AddDatabase("catalogdb");
var cache = builder
    .AddRedis("cache")
    .WithRedisInsight()
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
// Projects
var catalog = builder
    .AddProject<Projects.Catalog>("catalog")
    .WithReference(catalogDb)
    .WaitFor(catalogDb);
var basket = builder
    .AddProject<Projects.Basket>("basket")
    .WithReference(cache)
    .WaitFor(cache);
builder.Build().Run();
```

To enable Basket to discover Catalog, add `.WithReference(catalog)` to the Basket project definition:

```
var basket = builder
    .AddProject<Projects.Basket>("basket")
    .WithReference(catalog)
    .WaitFor(catalog);
```

This makes the Basket microservice aware of the Catalog microservice and waits for it to be available.

After saving the change, set **AppHost** as the startup project and run the distri**b**uted application. Then, open the **Aspire dashboard** and select the b**asket** project.

In its **environment variables**, you should see HTTP and **HTTPS** endpoints for the Catalog service, such as

```
services__catalog__http__0
services__catalog__https__0
```

These values confirm that Aspire has supplied the information Basket needs to locate Catalog dynamically.

With service discovery in place, Basket can now locate and call the Catalog API. The next step is to handle the product data returned by that call. Because the Catalog endpoint returns product details such as the price and name, Basket needs a corresponding Product model to deserialize and use that response.

Sharing the Product entity between Catalog and Basket microservices

The Catalog **API** returns a `Product` object, so the Basket microservice needs access to the same model definition when deserializing the response. Rather than creating a duplicate `Product` class in Basket, you will link the existing `Product.cs` file from the Catalog project.

1. Open the Basket microservice in Visual Studio.
2. Go to the `Models` folder.
3. Right-click the folder and select **Add | Existing Item**.
4. Then, navigate to the Catalog project's `Models` folder and select `Product.cs`.
5. From the **Add** dropdown, select **Add As Link**.

By adding the `Product` entity as a link, you create a shortcut to the original file in the Catalog microservice. This approach guarantees that any updates to `Product.cs` in Catalog will be immediately available in Basket, maintaining a single, consistent `Product` entity across both microservices.

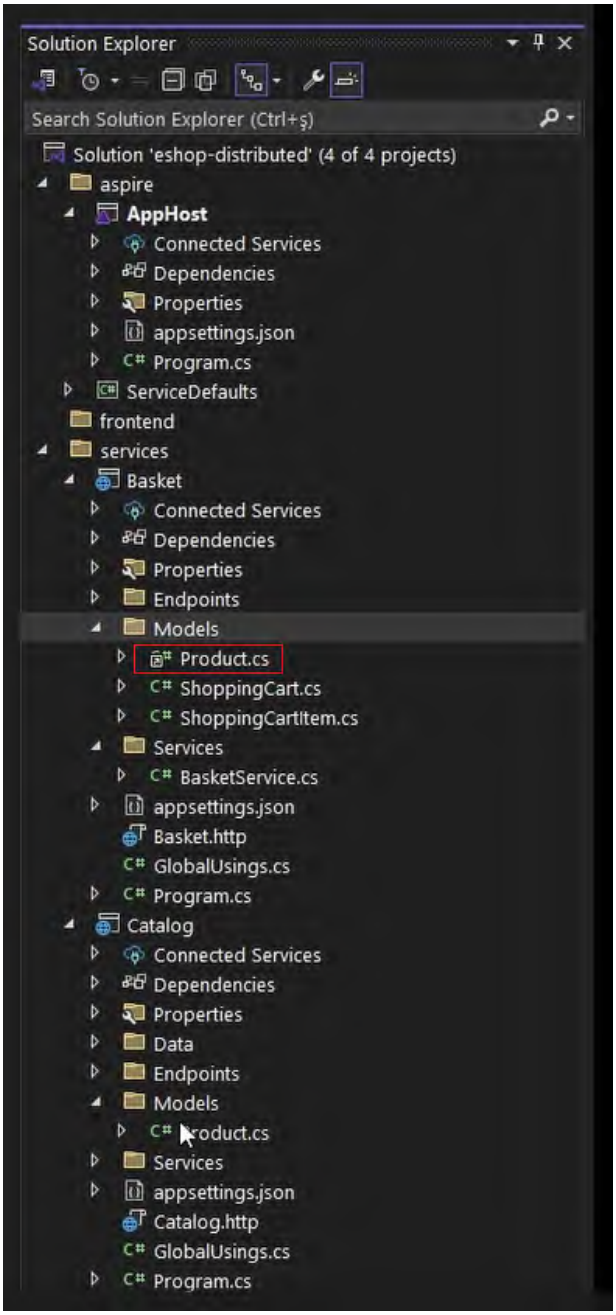


Figure 8.2 – The linked Product entity under the Basket microservice

With the shared model available, you can now create the HTTP client that will call the Catalog API.

Implementing the Catalog API client for HTTP communication

Create a new folder named `ApiClients` in the Basket project, and inside it create a class named `CatalogApiClient`. Start with the following class:

```
namespace Basket.ApiClients;
public class CatalogApiClient
{
}
```

The **API client** will encapsulate the HTTP communication with Catalog. Update the class so that it receives an `HttpClient` through dependency injection and exposes a method for retrieving a product by ID:

```
public class CatalogApiClient(HttpClient httpClient)
{
    public async Task<Product> GetProductById(int id)
    {
        var response =
            await httpClient.GetFromJsonAsync<Product>($"/products/{id}");
        return response!;
    }
}
```

Add the `Catalog` namespace at the start:

```
using Catalog.Models;
```

The `GetProductById` method sends an HTTP `GET` request to the Catalog microservice's `/products/{id}` endpoint and deserializes the response into the shared `Product` model. Keeping this logic inside `CatalogApiClient` also keeps HTTP-specific communication details out of the Basket business logic. That is, injecting the `HttpClient` into the `CatalogApiClient` encapsulates the Catalog endpoint logic in one place. This design allows you to add resilience patterns and enables automatic injection via ASP.NET dependency injection when registering services in `Program.cs`.

The next step is to register `CatalogApiClient` as a typed HTTP client in the Basket project's dependency injection container.

Registering the Catalog API client

To register `CatalogApiClient` in your Basket microservice using Visual Studio, you need to add it to the dependency injection container with the `AddHttpClient` extension method. This allows your application to inject `CatalogApiClient` wherever it is needed, targeting the **logical service name** for Aspire Service Discovery.

Open `Program.cs` in the Basket microservice project and add the following registration after `BasketService`:

```
builder.Services.AddHttpClient<CatalogApiClient>(client =>
{ client.BaseAddress = new("https+http://catalog");
});
```

Here, `catalog` is the logical service name defined when the Catalog project was registered in `AppHost`:

```
.AddProject<Projects.Catalog>("catalog")
```

Aspire service discovery resolves this logical name to the actual Catalog endpoint at runtime. This means the Basket microservice does not need to know the physical hostname or port on which Catalog is running.

Also when you add the registration line, Visual Studio will automatically add a `using` statement for `Basket.ApiClients` at the top of the file. Move this `using` statement to the `GlobalUsings.cs` file in the Basket microservice to make it available throughout your project.

The following code shows the updated `GlobalUsings.cs` file after adding the global using for `Basket.ApiClients`:

```
global using Basket.Models;
global using Basket.Services;
global using Basket.Endpoints;
global using Basket.ApiClients;
```

After saving the `GlobalUsings.cs` file, return to `Program.cs` and remove the now-unnecessary `using` statement for `Basket.ApiClients` at the top.

The completed service registrations in `Program.cs` should now look like this:

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.AddServiceDefaults();
builder.AddRedisDistributedCache(connectionName: "cache");
builder.Services.AddScoped<BasketService>();
builder.Services.AddHttpClient<CatalogApiClient>(client =>
{
    client.BaseAddress = new("https+http://catalog");
});
var app = builder.Build();
// Configure the HTTP request pipeline.
app.MapDefaultEndpoints();
app.MapBasketEndpoints();
app.UseHttpsRedirection();
app.Run();
```

With `CatalogApiClient` registered, it can now be injected into the Basket business layer.

Integrating Catalog into BasketService

To integrate the Catalog microservice into Basket, you will use `CatalogApiClient` to call Catalog synchronously from the `UpdateBasket` method in `BasketService`. Before the shopping cart is persisted to Redis, Basket will retrieve the latest product information from Catalog and use it to update each shopping-cart item.

Open `BasketService.cs` and inject `CatalogApiClient` alongside `IDistributedCache`:

```
public class BasketService(
    IDistributedCache cache,
    CatalogApiClient catalogApiClient)
```

Next, update the `UpdateBasket` method:

```
public async Task UpdateBasket(ShoppingCart basket)
{
    foreach (var item in basket.Items)
    {
        var product =
            await catalogApiClient.GetProductById(item.ProductId);
```

```
            item.Price = product.Price;
            item.ProductName = product.Name;
        }
        await cache.SetStringAsync(
            basket.UserName,
            JsonSerializer.Serialize(basket));
    }
```

For each basket item, `GetProductById` retrieves the corresponding product from Catalog. The returned price and product name replace the values supplied in the incoming basket request. Only after all items have been updated is the basket serialized and written to Redis. This ensures that each basket update uses the latest product information from Catalog, helping keep basket data consistent with the Catalog microservice.

You can now build the entire solution (right-click on the **Solution 'eshop-distributed'** and go to **Build Solution**) to ensure that the integration is successful and the build completes without errors.

Once the build completes without errors, you can test the complete flow to verify that Basket retrieves product information from Catalog and persists the updated basket to Redis.

Testing synchronous communication and end-to-end data flow

With the synchronous integration complete, you can now verify that the Basket microservice retrieves the latest product information from Catalog rather than relying on client-supplied values. To test this flow, you will use the `Basket.http` file in Visual Studio to invoke the Basket endpoints directly and observe how the request is processed across the two microservices.

Set the `AppHost` project as the startup project, make sure Docker Desktop is running, and launch the Aspire distributed application. Once the environment is running, open `Basket.http` in the Basket project.

To make the behavior easy to verify, set the price of each item in the `POST` request to `0`. When the request is processed, Basket should retrieve the current product prices and names from Catalog and replace these client-supplied values before persisting the basket to Redis.

```
POST {{Basket_HostAddress}}
Content-Type: application/json
{
  "UserName": "swn",
  "Items": [
```

```
    {
      "Quantity": 2,
      "Color": "Red",
      "Price": 0,
      "ProductId": 1,
      "ProductName": "Solar Powered Flashlight"
    },
    {
      "Quantity": 1,
      "Color": "Blue",
      "Price": 0,
      "ProductId": 2,
      "ProductName": "Hiking Poles"
    }
  ]
}
```

Next, set a breakpoint inside the `UpdateBasket` method in the `BasketService` class. Then, send the `POST` request from `Basket.http`. When the debugger stops at the breakpoint, step through the method. Basket iterates over each shopping-cart item and calls `CatalogApiClient.GetProductById` to retrieve the latest product price and name from Catalog.

The following code in `BasketService.cs` shows how each item is updated with data from the Catalog microservice:

```
public async Task UpdateBasket(ShoppingCart basket)
{
    // Before update(Add/remove Item) into SC, we should call Catalog ms
GetProductById method
    // Get latest product information and set Price and ProductName when adding
item into SC
    foreach (var item in basket.Items)
    {
        var product = await catalogApiClient.GetProductById(item.ProductId);
        item.Price = product.Price;
        item.ProductName = product.Name;
    }
    await cache.SetStringAsync(
```

```
        basket.UserName, JsonSerializer.Serialize(basket));
}
```

Each call to `CatalogApiClient.GetProductById` is handled by the following `GET` endpoint in `ProductEndpoints.cs`:

```
group.MapGet("/{id}", async (int id, ProductService service) =>
{
    var product = await service.GetProductByIdAsync(id);
    if (product is null)
        return Results.NotFound();
    return Results.Ok(product);
})
    .WithName("GetProductById")
    .Produces<Product>(StatusCodes.Status200OK)
    .Produces(StatusCodes.Status404NotFound);
```

The endpoint retrieves the requested product and returns its current details to Basket. `BasketService` then uses the returned price and product name to update the corresponding shopping-cart item.

If execution remains paused at a breakpoint for too long, the standard resilience handler configured in `ServiceDefaults` might cause the HTTP request to time out. If this happens, disable the breakpoints and resend the `POST` request.

When the request completes successfully, it returns `201 Created`. The response confirms that the prices supplied as `0` have been replaced with the current values returned by Catalog:

```
{
  "userName": "swn",
  "items": [
    {
      "quantity": 2,
      "color": "Red",
      "productId": 1,
      "price": 19.99,
      "productName": "Solar Powered Flashlight"
    },
    {
      "quantity": 1,
      "color": "Blue",
```

```
      "productId": 2,
      "price": 24.99,
      "productName": "Hiking Poles"
    }
  ],
  "totalPrice": 64.97
}
```

We can further verify the communication in the **Traces** view of the Aspire dashboard. Because the basket contains two items, you should see two synchronous calls from Basket to Catalog—one for each product.

To test the same flow with additional items, send another `POST` request containing three products in `Basket.http`. The resulting trace should show three calls to Catalog, after which the updated basket is persisted to Redis with the product information returned by Catalog.

Summary

In this chapter, we connected the Basket and Catalog microservices using Aspire Service Discovery. We began by establishing a project-to-project reference with `.WithReference(catalog)`, allowing Basket to discover Catalog through its logical service name rather than a hard-coded URL. We then linked the shared `Product` model into Basket, implemented `CatalogApiClient`, and registered it as a typed HTTP client using `AddHttpClient<CatalogApiClient>`.

Next, we integrated `CatalogApiClient` into `BasketService` and updated the `UpdateBasket` method so that each shopping-cart item retrieves the latest product price and name from Catalog before the basket is persisted to Redis. Finally, we tested the complete flow and used the Aspire dashboard to verify the synchronous calls between the two microservices.

At this point, Basket can request up-to-date product information directly from Catalog whenever you create or update a basket. However, this communication is synchronous: Basket must call Catalog and wait for its response as part of the request flow.

In the next chapter, we will introduce **asynchronous communication** using RabbitMQ and MassTransit. Instead of Basket calling Catalog directly when product data changes, Catalog will publish a `ProductPriceChangedIntegrationEvent` to RabbitMQ, and Basket will consume that event to update shopping-cart prices independently. This moves the application toward a more decoupled, event-driven architecture.

9

Asynchronous Communication with RabbitMQ and MassTransit in Aspire

In the previous chapter, we established **synchronous communication** between the Basket and Catalog microservices using **Aspire Service Discovery**. When a basket is created or updated, Basket calls Catalog directly to retrieve the latest product price and name before persisting the shopping cart to **Redis**.

That request/response flow keeps product data current when the basket is created or updated. However, it does not automatically update a basket that has already been stored in Redis. If a product price changes later in Catalog, the stored basket will still contain the previous price unless Basket calls Catalog again. To propagate these changes without requiring Basket to initiate another request, you need an **asynchronous communication** mechanism.

In this chapter, you will introduce asynchronous communication between the Catalog and Basket microservices. Instead of Basket requesting product information from Catalog, the Catalog microservice will publish a `ProductPriceChangedIntegrationEvent` whenever a product price changes. **RabbitMQ** will act as the **message broker**, while **MassTransit** will provide the messaging integration used to publish and consume the event. Basket can then react to the event and update the corresponding shopping-cart data in Redis.

You will begin by examining the asynchronous communication architecture and the product-price-change event flow. You will then add RabbitMQ to the Aspire environment, define the shared integration-event contract, and configure MassTransit in the Catalog and Basket microservices. With the messaging infrastructure in place, we will update Catalog to publish

the event and Basket to consume it. Finally, you will test the complete event-driven flow using the **RabbitMQ Management** interface and Aspire traces.

By the end of the chapter, you will have extended the eShop application with asynchronous messaging, allowing Catalog to notify Basket about product-price changes without requiring a direct request from Basket.

In this chapter, we'll be covering the following topics:

- Architecture of asynchronous communication between microservices
- Modeling the product-price-change event flow
- Adding RabbitMQ to the Aspire environment
- Connecting Catalog and Basket to RabbitMQ
- Defining shared integration events
- Configuring MassTransit for Catalog and Basket
- Publishing product-price-change events from Catalog
- Consuming product-price-change events in Basket
- Testing the asynchronous communication flow

Architecture of asynchronous communication between microservices

In this chapter, we are going to establish asynchronous communication between microservices using RabbitMQ and MassTransit, orchestrated with Aspire. In asynchronous communication, instead of relying on a direct request/response interaction for every update, services exchange information through messages and events. In our application, the Catalog microservice acts as the **publisher**, RabbitMQ as the message broker, and the Basket microservice as the **consumer**. MassTransit provides the messaging abstraction within the microservices, while Aspire orchestrates RabbitMQ and supplies the connection information required by the services.

The workflow begins when a product price changes in Catalog. Catalog publishes a `ProductPriceChangedIntegrationEvent` to RabbitMQ, and Basket consumes the event and updates matching shopping-cart items in Redis. Because the event is delivered through the broker, Catalog does not need to call Basket directly.

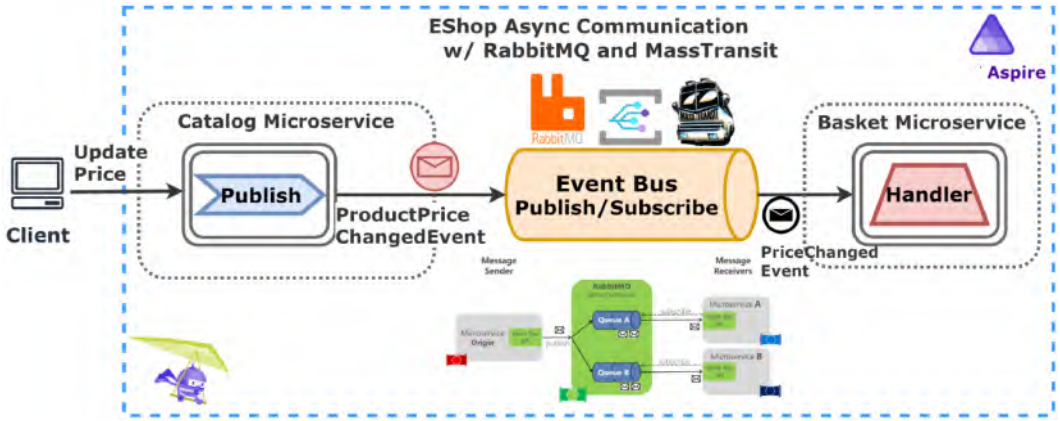


Figure 9.1 – Event-driven architecture diagram showing the Catalog microservice publishing ProductPriceChangedIntegrationEvent messages to RabbitMQ via MassTransit, and the Basket microservice handling these events asynchronously

This *publish/subscribe model* also makes the workflow extensible (extending system capabilities without rewriting existing code). Additional microservices, such as Inventory or Notification, can subscribe to relevant events without changing the Catalog-to-Basket interaction or introducing another direct dependency.

The implementation of async communication consists of four main parts:

1. Adding RabbitMQ as a backing service in the `AppHost` project so that Aspire can orchestrate the broker.
2. Referencing the RabbitMQ resource from Catalog and Basket so that both services receive the connection information they need.
3. Defining a shared `ProductPriceChangedIntegrationEvent` contract that both services understand.
4. Configuring MassTransit so that Catalog can publish the event and Basket can consume it.

Together, these elements establish the asynchronous communication path between the Catalog and Basket microservices. With the messaging infrastructure defined, the next step is to model the event that carries product-price changes between the two services.

Modeling the product-price-change event flow

The shared message at the center of this workflow is `ProductPriceChangedIntegrationEvent`. This event is central to how product price changes are propagated from the Catalog

microservice to the Basket microservice. In a typical scenario, you would raise a domain event for a product price change within the Catalog microservice, which would then trigger an integration event. However, for simplicity, we'll skip the domain event and directly publish the integration event from the Catalog microservice.

The following diagram illustrates the broader event model, including domain events such as `ProductCreatedEvent` and `ProductPriceChangedEvent` and the `ProductPriceChangedIntegrationEvent` that crosses the service boundary to Basket.

This integration event is consumed by the Basket microservice to update the shopping cart.

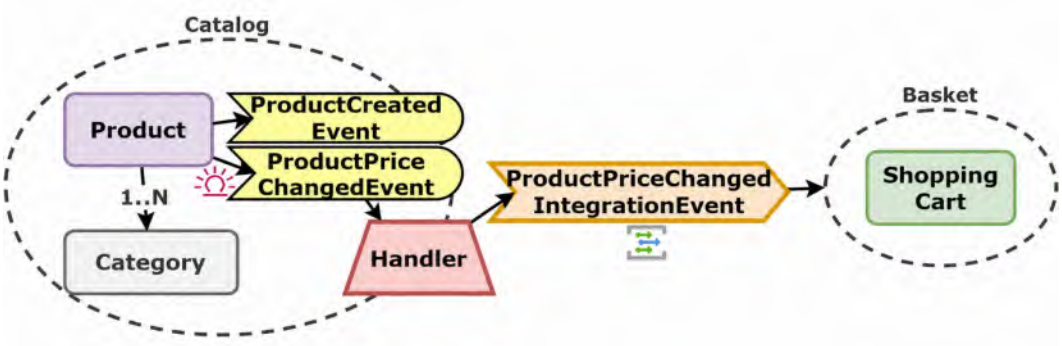


Figure 9.2 – Event model diagram showing ProductCreatedEvent and ProductPriceChangedEvent leading to ProductPriceChangedIntegrationEvent, which is consumed by the Basket microservice's Shopping Cart

At runtime, the workflow begins when a client updates a product price in the Catalog microservice through its **HTTP** `PUT` endpoint. Catalog updates the product and publishes a `ProductPriceChangedIntegrationEvent` containing the updated product information.

RabbitMQ then carries this integration event asynchronously using the **advanced message queuing protocol** (**AMQP**). The Basket microservice consumes the event through MassTransit, finds the corresponding item in the stored shopping cart, updates its product name and price, and writes the updated basket back to Redis.

To support this workflow, we will combine **REST**-based synchronous communication with AMQP-based asynchronous messaging. The client interacts with Catalog through its REST **API**, while RabbitMQ and MassTransit propagate the resulting price change to Basket without requiring a direct synchronous call between the two microservices.

With the event flow defined, the next step is to add the message broker that will carry events between Catalog and Basket.

Adding RabbitMQ to the Aspire environment

In this section, we will add RabbitMQ as a backing service in the AppHost project so that Aspire can create, configure, and orchestrate the broker alongside the other application resources.

Start by adding the RabbitMQ hosting integration to the AppHost project:

1. Open the solution in Visual Studio and navigate to the `AppHost` project.
2. Right-click **Dependencies** and select the option to add an Aspire package.
3. Choose `Aspire.Hosting.RabbitMQ` from the available hosting integrations and install it for the AppHost project.

After installation, verify that the package reference appears in the project file:

```
<PackageReference Include="Aspire.Hosting.RabbitMQ" Version="13.0.0" />
```

Next, configure RabbitMQ in AppHost `Program.cs` alongside the other backing services.

The following code adds a RabbitMQ resource named `rabbitmq` and enables the management interface and persistent storage:

```
var rabbitmq = builder
    .AddRabbitMQ("rabbitmq")
    .WithManagementPlugin()
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
```

This configuration tells Aspire to orchestrate a RabbitMQ container named `rabbitmq`. `WithManagementPlugin` enables the RabbitMQ management interface, `WithDataVolume` provides persistent storage, and `WithLifetime(ContainerLifetime.Persistent)` keeps the container available across application restarts.

By defining RabbitMQ as a backing service, you prepare your distributed environment for asynchronous messaging between microservices. This setup allows you to inject environment variables into your microservices so they can connect to the same RabbitMQ instance.

The next step is to connect Catalog and Basket to the broker so that both services receive the required connection information.

Connecting Catalog and Basket to RabbitMQ

Both Catalog and Basket microservices must reference the RabbitMQ resource. This allows Aspire to provide the broker connection information to each service and ensure RabbitMQ is ready before they start using it.

First, let's update AppHost `Program.cs` to add the RabbitMQ reference and `wait` condition to both projects.

For Catalog, keep the existing `catalogdb` reference and add `rabbitmq`:

```
var catalog = builder
    .AddProject<Projects.Catalog>("catalog")
    .WithReference(catalogDb)
    .WithReference(rabbitmq)
    .WaitFor(catalogDb)
    .WaitFor(rabbitmq);
```

For Basket, keep the existing `cache` and Catalog references and add `rabbitmq`:

```
var basket = builder
    .AddProject<Projects.Basket>("basket")
    .WithReference(cache)
    .WithReference(catalog)
    .WithReference(rabbitmq)
    .WaitFor(cache)
    .WaitFor(rabbitmq);
```

`WithReference(rabbitmq)` makes the RabbitMQ connection information available to each microservice, while `WaitFor(rabbitmq)` delays startup until the broker is ready.

Save the changes, make sure Docker Desktop is running, and start the AppHost project. The Aspire dashboard should now show RabbitMQ alongside the existing Catalog, Basket, PostgreSQL, and Redis resources.

Use the Aspire **Resources** dashboard to confirm that the RabbitMQ resource is running before moving on to the client-side messaging configuration.

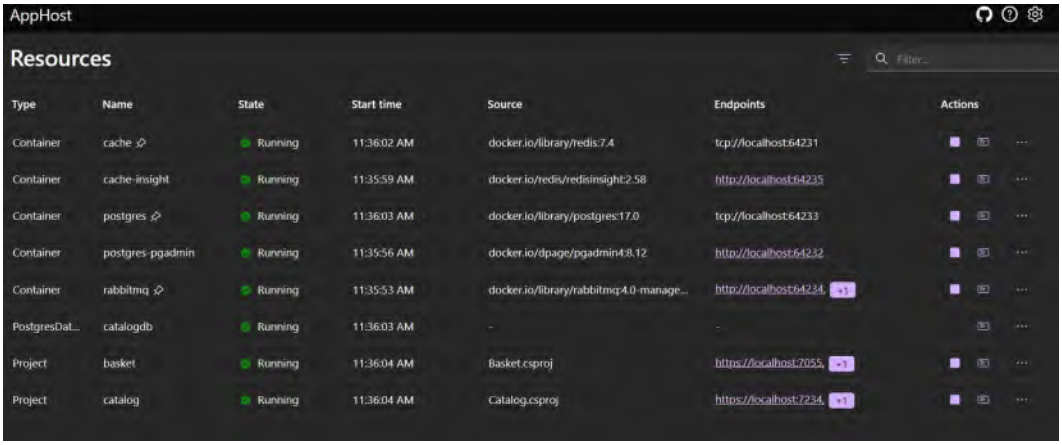


Figure 9.3 – AppHost resources dashboard showing running containers, including rabbitmq, with endpoints and state indicators

You can use the AppHost dashboard to verify the RabbitMQ credentials and the connection strings injected into the Catalog and Basket microservices. To do that, locate the **rabbitmq** container in the AppHost **Resources** list and open its resource details. Scroll down to the **Environment variables** section, where you can see variables such as `RABBITMQ_DEFAULT_USER` and `RABBITMQ_DEFAULT_PASS`.

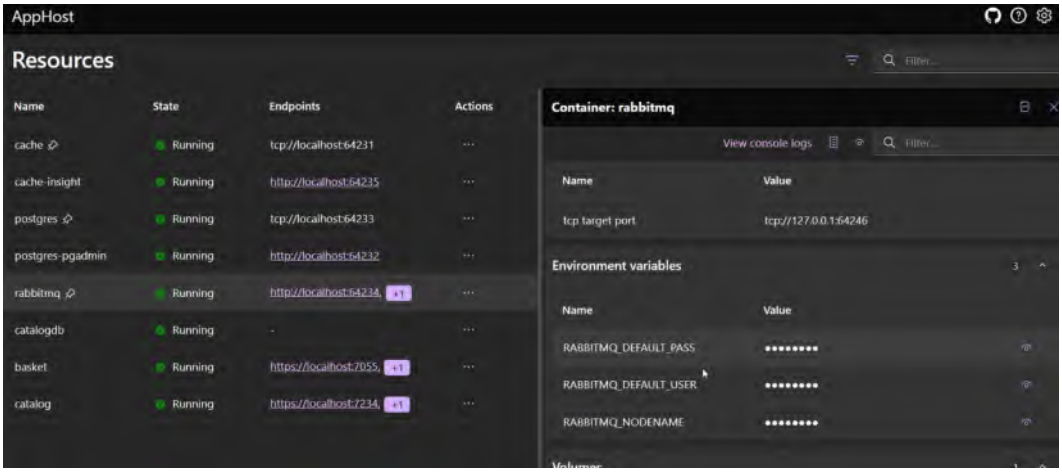


Figure 9.4 – rabbitmq environment variables

You can use these credentials to access the **RabbitMQ Management** interface. From the **rabbitmq** resource, select the provided management endpoint to open the **RabbitMQ**

Management login page. Enter `guest` as the **username** and copy the password from the `RABBITMQ_DEFAULT_PASS` environment variable in the AppHost dashboard.



Figure 9.5 – RabbitMQ Management login page

Next, verify that the RabbitMQ connection information has been injected into the microservices. Open the **Catalog** microservice in the AppHost dashboard and scroll down to the **Environment variables** section. You should see a RabbitMQ connection string, with a key containing `rabbitmq`. This connection string is injected through the `WithReference(rabbitmq)` method, allowing Catalog to connect to the RabbitMQ broker using AMQP.

Repeat the same check for the Basket microservice to confirm that it also has the RabbitMQ connection information.

With the environment variables and connection strings verified, you are ready to develop the shared messaging folders and classes for the `ProductPriceChanged` **integration event**.

Defining shared integration events

With RabbitMQ running and the connection information available to both services, you can define the **message contract** that Catalog will publish and Basket will consume. In a production system, shared integration events are typically placed in a dedicated messaging or contracts library referenced by the participating services.

For this chapter, however, the event classes will be placed in the ServiceDefaults project to keep the example compact. Both Catalog and Basket microservices already reference this project, so they can access the same event definitions without adding another project to the solution. Placing them there is a convenience for the sample rather than a recommended production boundary.

In the ServiceDefaults project, create a `Messaging` folder and, inside it, an `Events` folder for the integration-event types.

Both Catalog and Basket microservices already reference ServiceDefaults, as shown by the existing project reference (The following code is part of the Basket microservice's project file):

```
<ItemGroup>
<ProjectReference Include="..\ServiceDefaults\ServiceDefaults.csproj" />
</ItemGroup>
```

Inside `Messaging/Events`, first create `IntegrationEvent.cs`. This **base record** provides common metadata for integration events. Remove any unnecessary `using` statements. Define `IntegrationEvent` with the following properties, ensuring they are initialized exactly once upon creation:

```
public Guid EventId { get; } = Guid.NewGuid();
```

This property provides a stable, unique identifier for the event, which is critical for message tracking and **idempotency**.

```
public DateTime OccurredOn { get; } = DateTime.UtcNow;
```

This property records exactly when the event was created (in **UTC**) for accurate auditing and timeline diagnostics.

```
public string EventType => GetType().AssemblyQualifiedName;
```

This property exposes the fully qualified event type, which can be useful for diagnostics or routing scenarios. (**Note**: *It is safe to use an expression body here because the object's type does not change*).

These properties provide a common shape that domain-specific integration events can inherit.

The following code defines the base integration event record:

```
namespace ServiceDefaults.Messaging.Events;
public record IntegrationEvent
{
    public Guid EventId { get; } = Guid.NewGuid();
    public DateTime OccurredOn { get; } = DateTime.UtcNow;
```

```
    public string EventType => GetType().AssemblyQualifiedName!;
}
```

Next, create `ProductPriceChangedIntegrationEvent.cs` under the `Events` folder and inherit from `IntegrationEvent`. The event carries the product data the Basket microservice needs when reacting to a price change:

```
public int ProductId { get; set; } = default!;
```

The unique identifier for the product in the catalog.

```
public string Name { get; set; } = default!;
```

The product name.

```
public string Description { get; set; } = default!;
```

The product description.

```
public decimal Price { get; set; } = default!;
```

The current price of the product.

```
public string ImageUrl { get; set; } = default!;
```

The URL for the product image.

By publishing this event, the Catalog microservice can notify the Basket microservice that a product's price has changed, allowing the Basket service to update shopping carts accordingly.

The following code defines the `ProductPriceChangedIntegrationEvent` record:

```
namespace ServiceDefaults.Messaging.Events;
public record ProductPriceChangedIntegrationEvent : IntegrationEvent
{
    public int ProductId { get; set; } = default!;
    public string Name { get; set; } = default!;
    public string Description { get; set; } = default!;
    public decimal Price { get; set; } = default!;
    public string ImageUrl { get; set; } = default!;
}
```

As mentioned earlier, for a production architecture, consider moving these contracts to a dedicated shared messaging library so that ServiceDefaults remains focused on cross-cutting hosting concerns and the messaging boundary is explicit.

The next step is to develop client integration for each microservice that needs asynchronous communication.

Configuring MassTransit for Catalog and Basket

With the shared event contract in place, the next step is to configure the messaging layer. This involves creating MassTransit **extension methods** to configure the RabbitMQ connection and adding them to the ServiceDefaults project. Centralizing this configuration allows each microservice to integrate with RabbitMQ using the connection string supplied by Aspire, without duplicating messaging configuration across projects.

Rather than using the native RabbitMQ client integration provided by Aspire, you will use MassTransit for client-side messaging. MassTransit uses the RabbitMQ connection string supplied by Aspire to publish and consume integration events while also providing features such as consumer registration, endpoint configuration, retry policies, and message scheduling. These capabilities simplify the implementation of event-driven communication across the application.

Let's start by adding the `MassTransit.RabbitMQ` package to ServiceDefaults and creating a reusable registration extension:

1. Open Visual Studio and go to the **ServiceDefaults** project.
2. Right-click **Dependencies** and select **Manage NuGet Packages**.
3. Click the **Browse** tab and search for `MassTransit.RabbitMQ`.
4. Download and install the `MassTransit.RabbitMQ` NuGet package under the **ServiceDefaults** project.

Installing the package in ServiceDefaults allows you to define the MassTransit configuration once and reuse it from Catalog and Basket through their existing project references.

After installing the package, you can confirm its presence by double-clicking the **ServiceDefaults** project and checking the project file. The following code in the project file shows the `MassTransit.RabbitMQ` package reference:

```
<PackageReference Include="MassTransit.RabbitMQ" Version="8.3.6" />
```

Next, add a new class `MassTransitExtensions.cs` under the `Messaging` folder. This class will centralize the RabbitMQ and MassTransit registration shared by the microservices.

Create a static extension method named `AddMassTransitWithAssemblies` that accepts the assemblies MassTransit should scan for consumers and related components.

Inside this method, you use the `services.AddMassTransit` extension method, which comes from the MassTransit library and is the primary registration method. You then configure MassTransit as follows:

- Set the kebab-case **endpoint name formatter** to ensure endpoints follow a consistent naming schema, such as `product-price-changed`, by calling `config.SetKebabCaseEndpointNameFormatter()`.
- Set up **ephemeral saga state management** with `config.SetInMemorySagaRepositoryProvider()`.
- Configure consumers, saga state machines, sagas, and activities by passing the `assemblies` parameter to each respective method:

```
config.AddConsumers(assemblies);
config.AddSagaStateMachines(assemblies);
config.AddSagas(assemblies);
config.AddActivities(assemblies);
```

- Configure MassTransit to use RabbitMQ by calling `config.UsingRabbitMq((context, configurator) => { ... })`.

When configuring RabbitMQ, retrieve the `rabbitmq` connection string that Aspire injected into the microservice through `WithReference(rabbitmq)`.

Within the `UsingRabbitMq` configuration, retrieve the configuration and connection string as follows:

```
var configuration = context.GetRequiredService<IConfiguration>();
var connectionString = configuration.GetConnectionString("rabbitmq");
```

Then, configure the RabbitMQ host and endpoints:

```
configurator.Host(connectionString);
configurator.ConfigureEndpoints(context);
```

The following code shows the complete `AddMassTransitWithAssemblies` extension method:

```
using MassTransit;
using Microsoft.Extensions.DependencyInjection;
```

```
using System.Reflection;
namespace ServiceDefaults.Messaging
{
    public static class MassTransitExtensions
    {
        public static IServiceCollection AddMassTransitWithAssemblies
        (this IServiceCollection services, params Assembly[] assemblies)
        {
            services.AddMassTransit(config =>
            {
                config.SetKebabCaseEndpointNameFormatter();
                config.SetInMemorySagaRepositoryProvider();
                config.AddConsumers(assemblies);
                config.AddSagaStateMachines(assemblies);
                config.AddSagas(assemblies);
                config.AddActivities(assemblies);
                config.UsingRabbitMq((context, configurator) =>
                {
                    var configuration =
context.GetRequiredService<IConfiguration>();
                    var connectionString =
configuration.GetConnectionString("rabbitmq");
                    configurator.Host(connectionString);
                    configurator.ConfigureEndpoints(context);
                });
            });
            return services;
        }
    }
}
```

Save the file and build the solution to verify the shared MassTransit registration before applying it to the individual microservices.

Now, we need to register `AddMassTransitWithAssemblies` in both the Catalog and Basket microservices so that each service can participate in the RabbitMQ message flow.

First, we will add the MassTransit extension method to the Catalog microservice. Navigate to the `Program.cs` file of the Catalog project and, locate the section where you add services to the container. Add the MassTransit registration alongside the existing service registrations:

```
builder.Services.AddMassTransitWithAssemblies(
    Assembly.GetExecutingAssembly());
```

`Assembly.GetExecutingAssembly()` tells MassTransit to scan the Catalog assembly for consumers, sagas, and related messaging components.

To keep your `Program.cs` file clean, move the necessary `using` statements to your global `usings` file. For example, add the following lines to your global `usings` file:

```
global using ServiceDefaults.Messaging;
global using System.Reflection;
```

The resulting Catalog `Program.cs` includes the MassTransit registration alongside the existing database and `ProductService` registrations:

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.AddServiceDefaults();
builder.AddNpgsqlDbContext<ProductDbContext>(connectionName: "catalogdb");
builder.Services.AddScoped<ProductService>();
builder.Services.AddMassTransitWithAssemblies(Assembly.GetExecutingAssembly());
var app = builder.Build();
// Configure the HTTP request pipeline.
app.MapDefaultEndpoints();
app.UseMigration();
app.MapProductEndpoints();
app.UseHttpsRedirection();
app.Run();
```

Now, apply the same MassTransit registration to the Basket `Program.cs`:

```
builder.Services.AddMassTransitWithAssemblies(Assembly.GetExecutingAssembly());
```

Again, move the `using` statements to the global `usings` file for the Basket microservice:

```
global using ServiceDefaults.Messaging;
global using System.Reflection;
```

Your updated `Program.cs` for the Basket microservice should look like this:

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.AddServiceDefaults();
builder.AddRedisDistributedCache(connectionName: "cache");
builder.Services.AddScoped<BasketService>();
builder.Services.AddMassTransitWithAssemblies(Assembly.GetExecutingAssembly());
builder.Services.AddHttpClient<CatalogApiClient>(client =>
{
    client.BaseAddress = new("https+http://catalog");
});
var app = builder.Build();
// Configure the HTTP request pipeline.
app.MapDefaultEndpoints();
app.MapBasketEndpoints();
app.UseHttpsRedirection();
app.Run();
```

At this point, both services are configured to use the same RabbitMQ broker through MassTransit. Catalog will publish `ProductPriceChangedIntegrationEvent`, while Basket will provide the consumer that reacts to the event.

With the messaging infrastructure configured, you can now connect it to the product-update workflow.

Publishing product-price-change events from Catalog

In this section, we will implement asynchronous communication between the Catalog and Basket microservices using RabbitMQ. Whenever a product price changes, the Catalog microservice will publish a `ProductPriceChangedIntegrationEvent`, which the Basket microservice—and any future subscribers—can consume independently.

Because this workflow crosses service and messaging boundaries, it is useful to include MassTransit operations in the application's distributed traces. This allows you to follow a

message as it is published, sent through RabbitMQ, received, and consumed alongside the existing HTTP traces in the Aspire dashboard.

Before implementing the publisher, add MassTransit as an **OpenTelemetry** tracing source. In `Extensions.cs`, extend the existing tracing configuration by adding the MassTransit source alongside the ASP.NET Core and `HttpClient` instrumentation:

```
.WithTracing(tracing =>
{
    tracing.AddSource(builder.Environment.ApplicationName)
    .AddAspNetCoreInstrumentation()
    // Uncomment the following line to enable gRPC instrumentation (requires the
Grpc.Net.Client package)
    //.AddGrpcClientInstrumentation()
    .AddHttpClientInstrumentation()
    .AddSource("MassTransit");
});
```

Adding `.AddSource("MassTransit")` enables MassTransit tracing activities to be captured as part of the application's distributed traces.

With tracing configured, you can now implement the event-driven workflow. Open `Catalog/Services/ProductService.cs`. We will update the `UpdateProductAsync` method so that Catalog publishes a `ProductPriceChangedIntegrationEvent` whenever a product's price changes.

The current implementation of the `UpdateProductAsync` method updates the product fields and saves the changes:

```
public async Task UpdateProductAsync(Product updatedProduct, Product inputProduct)
{
    // update product with new values
    updatedProduct.Name = inputProduct.Name;
    updatedProduct.Description = inputProduct.Description;
    updatedProduct.ImageUrl = inputProduct.ImageUrl;
    updatedProduct.Price = inputProduct.Price;
    dbContext.Products.Update(updatedProduct);
    await dbContext.SaveChangesAsync();
}
```

To publish the integration event, inject MassTransit's `IBus` interface into `ProductService`:

```
public class ProductService(ProductDbContext dbContext, IBus bus)
{
    // ...
}
```

`IBus` is provided by the MassTransit registration and exposes the `Publish` operation used to publish integration events to RabbitMQ.

Next, update `UpdateProductAsync` to compare the existing product price with the incoming price. If the price has changed, create a `ProductPriceChangedIntegrationEvent` containing the relevant product information and publish it using the `bus.Publish` method before continuing with the update:

```
public async Task UpdateProductAsync(
    Product updatedProduct,
    Product inputProduct)
{
    // If the price has changed, publish the integration event
    if (updatedProduct.Price != inputProduct.Price)
    {
        var integrationEvent = new ProductPriceChangedIntegrationEvent
        {
            ProductId = updatedProduct.Id, // ID comes from the database entity
            Name = inputProduct.Name,
            Description = inputProduct.Description,
            Price = inputProduct.Price,
            ImageUrl = inputProduct.ImageUrl
        };
        await bus.Publish(integrationEvent);
    }
    // Update product with new values
    updatedProduct.Name = inputProduct.Name;
    updatedProduct.Description = inputProduct.Description;
    updatedProduct.ImageUrl = inputProduct.ImageUrl;
    updatedProduct.Price = inputProduct.Price;
    dbContext.Products.Update(updatedProduct);
    await dbContext.SaveChangesAsync();
}
```

Whenever the price changes, the Catalog microservice publishes the integration event to RabbitMQ, allowing interested consumers, such as the Basket microservice, to react independently.

This implementation introduces a **dual-write** concern. The dual-write problem occurs when you attempt to write to two separate distributed systems as part of the same operation. In this case, updating the Catalog database and publishing the event to RabbitMQ are separate operations. If one succeeds while the other fails, the two systems can become inconsistent. For example, if the database transaction commits successfully but the subsequent event publish fails (perhaps due to a network blip with RabbitMQ), the Catalog will reflect the new price, but downstream services will never be notified.

In production, an **outbox pattern** is commonly used to address this problem. The outgoing event is stored in the same database transaction as the business update and published only after that transaction commits successfully. To keep this sample focused on the messaging workflow, we will not add an outbox here.

Consuming product-price-change events in Basket

The Catalog microservice can now publish the event, so the next step is to handle it in the Basket outbox pattern. You will create a MassTransit consumer that receives `ProductPriceChangedIntegrationEvent` and delegates the basket update to `BasketService`. This will complete the end-to-end asynchronous flow.

In the Basket microservice project, create an `EventHandlers` folder and add a new class called `ProductPriceChangedIntegrationEventHandler.cs`.

The following code creates the initial class in the new folder:

```
namespace Basket.EventHandlers;
public class ProductPriceChangedIntegrationEventHandler
{
}
```

Next, we inject `BasketService` as a primary constructor and implement the `IConsumer<ProductPriceChangedIntegrationEvent>` interface from MassTransit. This interface requires you to implement the `Consume` method, which will handle the event when it is received.

The following code shows the updated class definition with the interface and constructor:

```
using MassTransit;
using ServiceDefaults.Messaging.Events;
namespace Basket.EventHandlers;
public class ProductPriceChangedIntegrationEventHandler(BasketService service)
: IConsumer<ProductPriceChangedIntegrationEvent>
{
}
```

Now, implement the `Consume` method. In this method, you retrieve the event context provided by MassTransit and RabbitMQ, which contains the product ID and new price. You then call the `UpdateBasketItemProductPrices` method on the injected `BasketService`, passing the product ID and price from the event.

The following code implements the `Consume` operation:

```
public async Task Consume(ConsumeContext<ProductPriceChangedIntegrationEvent>
context)
{
    // find products on basket and update price
    await service.UpdateBasketItemProductPrices(
        context.Message.ProductId, context.Message.Price);
}
```

When the Basket `microservice` receives a `ProductPriceChangedIntegrationEvent`, the consumer passes the affected product ID and new price to `BasketService`, which is responsible for updating the corresponding basket data stored in Redis.

To complete the consumer path, implement `UpdateBasketItemProductPrices` in `BasketService`. The service already uses `IDistributedCache` as an abstraction for accessing the Redis-backed basket cache. The new method will retrieve the user's basket, update the affected product price, and save the modified basket back to Redis.

One limitation of `IDistributedCache` is that it does not provide a way to enumerate all cache keys. For this reason, the sample updates the basket for the fixed user `swn` only. This keeps the example focused on the messaging flow. In a production implementation, you would need a storage strategy that can efficiently identify all baskets containing the affected product.

The following code implements the `UpdateBasketItemProductPrices` method in the `BasketService` class:

```
internal async Task UpdateBasketItemProductPrices(int productId, decimal price)
{
    // IDistributedCache not supported list of keys function
    // https://github.com/dotnet/runtime/issues/36402
    var basket = await GetBasket("swn");
    var item = basket!.Items.FirstOrDefault(
        x => x.ProductId == productId);
    if (item != null)
    {
        item.Price = price;
        await cache.SetStringAsync(
            basket.UserName, JsonSerializer.Serialize(basket));
        }
}
```

The method retrieves the `swn` basket, finds the item with the matching `productId`, updates its price, and writes the modified basket back to Redis using `SetStringAsync`.

With the publisher and consumer in place, you can now test the complete event-driven workflow.

Testing the asynchronous communication flow

In this test, you will create a basket, change a product price in Catalog, observe the resulting event in RabbitMQ, and confirm that Basket updates the product price stored in Redis.

Before sending the requests, we need to set breakpoints at the key points in the workflow so that you can follow the publish and consume path.

Let's start in `Catalog/ProductEndpoints.cs` at the `PUT` endpoint that updates a product:

```
// PUT (Update)
group.MapPut("/{id}", async
(int id, Product inputProduct, ProductService service) =>
{
    var updatedProduct = await service.GetProductByIdAsync(id);
    if (updatedProduct is null) return Results.NotFound();
    await service.UpdateProductAsync(updatedProduct, inputProduct);
    return Results.NoContent();
```

```
})
    .WithName("UpdateProduct")
    .Produces(StatusCodes.Status404NotFound)
    .Produces(StatusCodes.Status204NoContent);
```

Set a breakpoint on the call to `UpdateProductAsync`, which hands the update to the Catalog business layer.

```
await service.UpdateProductAsync(updatedProduct, inputProduct);
```

This lets you observe when the request leaves the endpoint and enters the Catalog business layer.

Next, open `ProductService.UpdateProductAsync` and set a breakpoint on the event publication. The method checks whether the product price has changed and, if so, creates and publishes a `ProductPriceChangedIntegrationEvent`:

```
public async Task UpdateProductAsync(
    Product updatedProduct,
    Product inputProduct)
{
    // 1. Determine whether the price has changed
    bool priceChanged = updatedProduct.Price != inputProduct.Price;
    // 2. Update the product with the new values
    updatedProduct.Name = inputProduct.Name;
    updatedProduct.Description = inputProduct.Description;
    updatedProduct.ImageUrl = inputProduct.ImageUrl;
    updatedProduct.Price = inputProduct.Price;
    dbContext.Products.Update(updatedProduct);
    await dbContext.SaveChangesAsync();
    // 3. Publish the integration event only after a successful save
    if (priceChanged)
    {
        var integrationEvent = new ProductPriceChangedIntegrationEvent
        {
            ProductId = updatedProduct.Id,
            Name = updatedProduct.Name,
            Description = updatedProduct.Description,
            Price = updatedProduct.Price,
            ImageUrl = updatedProduct.ImageUrl
        };
```

```
        await bus.Publish(integrationEvent);
    }
}
```

Set the breakpoint on:

```
await bus.Publish(integrationEvent);
```

This lets you observe when Catalog publishes the integration event through MassTransit.

On the Basket side, open `ProductPriceChangedIntegrationEventHandler.cs` and set a breakpoint inside `Consume`. This lets you observe when MassTransit delivers the event from RabbitMQ to the Basket consumer:

```
public class ProductPriceChangedIntegrationEventHandler :
IConsumer<ProductPriceChangedIntegrationEvent>
{
    public async Task Consume(ConsumeContext<ProductPriceChangedIntegrationEvent>
context)
    {
        // find products on basket and update price
        await service.UpdateBasketItemProductPrices(
    context.Message.ProductId, context.Message.Price);
    }
}
```

Finally, open `BasketService.UpdateBasketItemProductPrices` and set a breakpoint where the matching basket item is updated. This lets you confirm that the consumed event is applied to the Redis-backed basket:

```
internal async Task UpdateBasketItemProductPrices(int productId, decimal price)
{
    // IDistributedCache not supported list of keys function
    // https://github.com/dotnet/runtime/issues/36402
    var basket = await GetBasket("swn");
    var item = basket!.Items.FirstOrDefault(x => x.ProductId == productId);
    if (item != null)
    {
        item.Price = price;
        await cache.SetStringAsync(basket.UserName,
JsonSerializer.Serialize(basket));
```

```
    }
}
```

With these breakpoints in place, you are ready to exercise the workflow end to end.

First, create or update the basket for user `swn` by sending the `POST` request from `Basket.http`. This establishes the shopping-cart data in Redis before you change the product price in Catalog.

The following request creates the basket with two items:

```
POST {{Basket_HostAddress}}/swn
Content-Type: application/json
{
    "UserName": "swn",
    "Items": [
    {
        "Quantity": 2,
        "Color": "Red",
        "Price": 0,
        "ProductId": 1,
        "ProductName": "Solar Powered Flashlight"
    },
    {
        "Quantity": 1,
        "Color": "Blue",
        "Price": 0,
        "ProductId": 2,
        "ProductName": "Hiking Poles"
    }]
}
```

The request returns `201 Created`, and the basket is stored in Redis. The response shows the current Catalog prices applied when the basket was created:

```
{
    "userName": "swn",
    "items": [
    {
        "quantity": 2,
        "color": "Red",
        "productId": 1,
```

```
        "price": 19.99,
        "productName": "Solar Powered Flashlight"
    },
    {
        "quantity": 1,
        "color": "Blue",
        "productId": 2,
        "price": 24.99,
        "productName": "Hiking Poles"
    }],
    "totalPrice": 64.97
}
```

This establishes `19.99` as the initial price of the Solar Powered Flashlight before the asynchronous update.

Next, switch to `Catalog.http` and send a `PUT` request that changes the price of product 1:

```
PUT {{Catalog_HostAddress}}/1
Content-Type: application/json
{
    "name": "Solar Powered Flashlight",
    "description": "A fantastic product for outdoor enthusiasts",
    "price": 5.99,
    "imageUrl": "product1.png"
}
```

This request changes the Solar Powered Flashlight price from `19.99` to `5.99`. That price change triggers the asynchronous workflow.

Execution should first stop at the breakpoint in the Catalog `PUT` endpoint. When you continue, `ProductService.UpdateProductAsync` detects the price change and reaches the breakpoint on:

```
await bus.Publish(integrationEvent);
```

Continue execution again. MassTransit publishes the `ProductPriceChangedIntegrationEvent` to RabbitMQ, which routes it to the Basket consumer.

Execution should then stop inside `ProductPriceChangedIntegrationEventHandler.Consume`, confirming that Basket has received the event. The consumer passes the product ID and new price to:

```
await service.UpdateBasketItemProductPrices(
    context.Message.ProductId,
    context.Message.Price);
```

Continue to the breakpoint in `BasketService.UpdateBasketItemProductPrices`. Here, the Basket service finds product `1` in the swn basket, changes its price to 5.99, and writes the updated basket back to Redis.

The complete path is therefore:

```
Catalog PUT request
  ↓
ProductService
  ↓
MassTransit publishes event
  ↓
RabbitMQ
  ↓
Basket consumer
  ↓
BasketService
          ↓
Redis
```

While stepping through these breakpoints, the HTTP request might exceed the configured timeout before the workflow finishes. If this happens, disable the breakpoints and resend the request so that the complete workflow can execute without interruption.

For example, you might see an error similar to the following:

```
The request was canceled due to the configured timeout of 20 seconds elapsing.
```

Once the workflow has completed, you can verify the messaging path through the **RabbitMQ Management** interface and the **Aspire dashboard**.

Open the **RabbitMQ Management dashboard** and inspect the overall message rates, queue status, and connection activity.

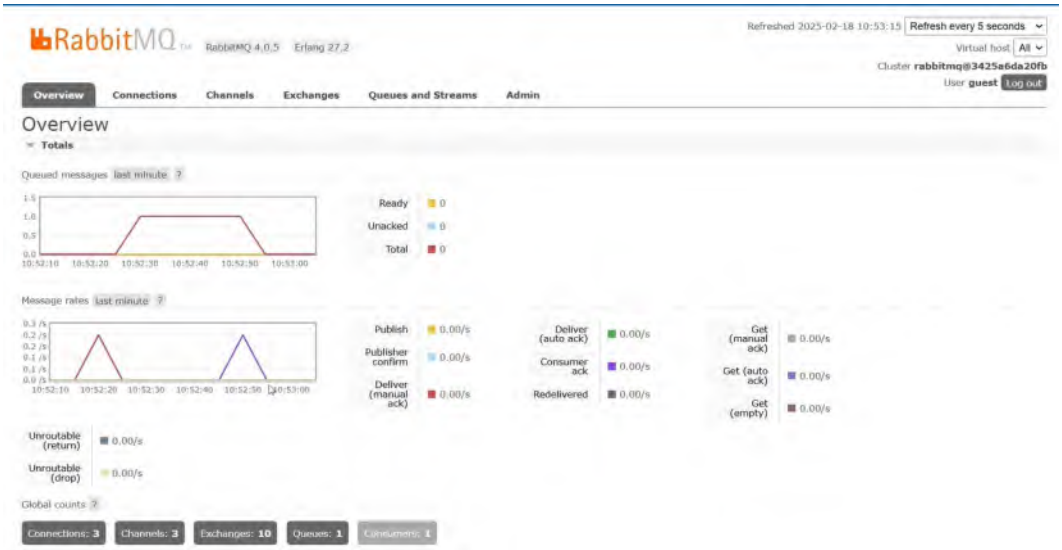


Figure 9.6 – RabbitMQ Management dashboard showing message rates, queue status, and connection statistics for the integration event workflow

Next, open the **Connections** tab. You should see active connections for the Catalog and Basket microservices.

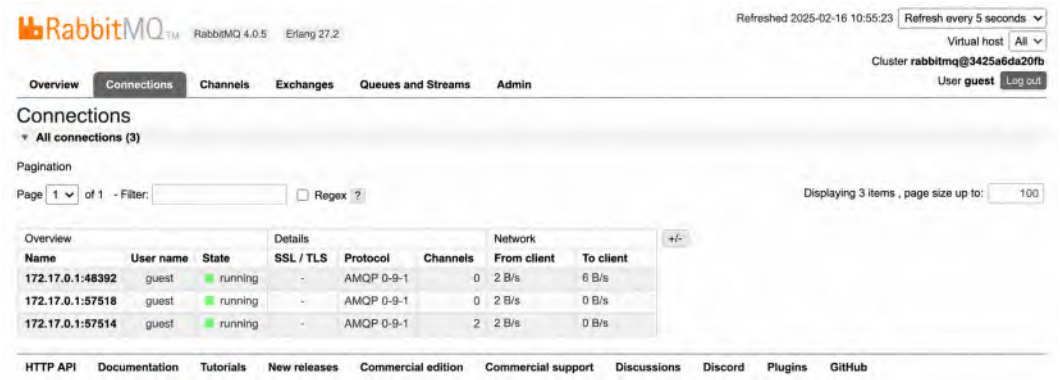


Figure 9.7 – RabbitMQ Management dashboard Connections tab showing Basket and Catalog microservices as active connections

Then, open the **Queues and Streams** tab and locate the `product-price-changed-integration-event-handler` queue used by the Basket consumer.



Figure 9.8 – RabbitMQ Management dashboard Queues and Streams tab showing the product-price-changed-integration-event-handler queue

Open the queue to inspect its consumer status and message activity.

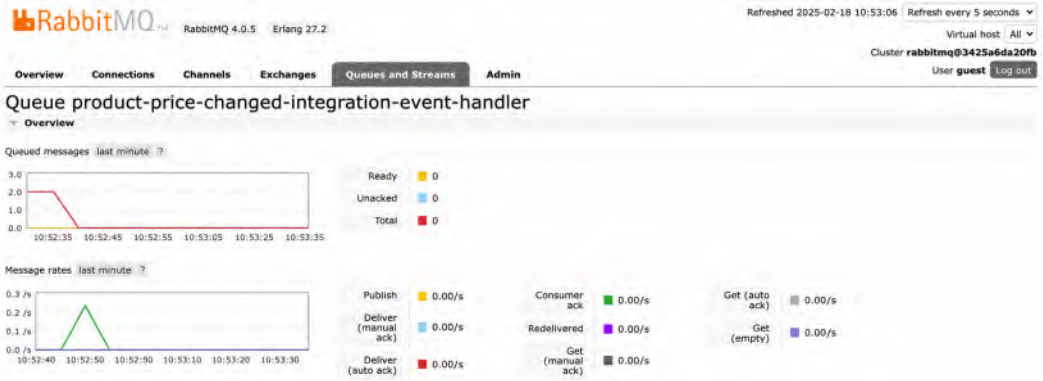


Figure 9.9 – RabbitMQ Management dashboard showing details for the product-price-changed-integration-event-handler queue

Finally, return to `Basket.http` and send a `GET` request for the `swn` basket. The first product price should now be `5.99` instead of `19.99`, confirming that the asynchronous event updated the shopping cart.

```
GET {{Basket_HostAddress}}/swn
Accept: application/json
```

After the event has been consumed, the price of product 1 should now be `5.99` instead of `19.99`:

```
{
    "userName": "swn",
    "items": [
    {
        "quantity": 2,
        "color": "Red",
        "productId": 1,
        "price": 5.99,
        "productName": "Solar Powered Flashlight" },
     ...
    ],
    "totalPrice": 36.97
}
```

This confirms that the product-price-change event published by Catalog was consumed by Basket and applied to the shopping cart stored in Redis.

You can also inspect the **Catalog** trace in the Aspire dashboard. From the **Resources** view, open the Catalog resource and select **Traces**. The trace shows the HTTP `PUT` request that starts the asynchronous workflow, the update to the Postgres database, the publishing of the `ProductPriceChangedIntegrationEvent` to RabbitMQ, and the consumption of the event by the Basket microservice, which updates the shopping cart items in the cache accordingly.

Together, the updated basket, RabbitMQ activity, and Aspire trace confirm that the event moved from Catalog to Basket asynchronously, without requiring a direct synchronous call between the two services.

Figure 9.10 shows the AppHost trace for the product update, including the MassTransit send and Basket event-consumption spans.

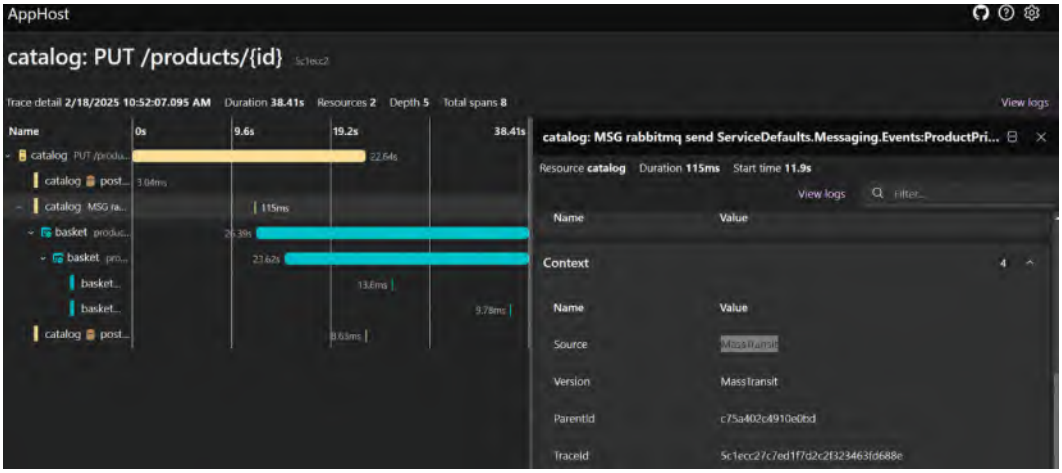


Figure 9.10 – AppHost trace showing the Catalog PUT /products/{id} operation, MassTransit RabbitMQ send, and event consumption by the Basket microservice, with detailed span and messaging metadata

Summary

In this chapter, we implemented asynchronous communication between the Catalog and Basket microservices using Aspire, RabbitMQ, and MassTransit. We began by defining the product-price-change workflow, adding RabbitMQ as an AppHost backing service, and connecting both microservices to the broker. We then created the shared `ProductPriceChangedIntegrationEvent` contract and configured MassTransit to support the publish/subscribe flow between the services.

Next, we updated the Catalog microservice to publish an integration event whenever a product price changes and created a Basket consumer to receive that event and update the corresponding shopping-cart price stored in Redis. Along the way, we also examined the dual-write concern that arises when publishing an event and updating the Catalog database as separate operations, and suggested the outbox pattern as a production-oriented approach for keeping these operations consistent.

Finally, we tested the workflow end to end by changing a product price in Catalog, tracing the event through RabbitMQ and MassTransit, confirming its consumption in Basket, and verifying the resulting update in Redis. With this event-driven communication path in place, Catalog can now propagate product-price changes to Basket asynchronously, without requiring a direct request from the consuming service.

With communication between the microservices now established, in the next chapter, we will turn to securing the application. We will introduce Keycloak as an **identity provider** and protect the Basket microservice so that its endpoints can be accessed only by authenticated users.

10

Securing the Basket Microservice with Keycloak Authentication

In the previous chapter, we established asynchronous communication between the Catalog and Basket microservices using RabbitMQ and MassTransit. With the communication paths between the services now in place, the next step is to secure access to the application.

In this chapter, we will introduce **Keycloak** as the **identity and access management** provider for the distributed application and integrate it with Aspire. We will first look at the role of Keycloak and how it supports authentication and authorization through OAuth 2.0 and OpenID Connect. We will then add Keycloak as a backing service in AppHost and configure the identity settings required by the application.

Next, we will create a realm, register a client, and add a test user in Keycloak before configuring the Basket microservice to validate **JWT** bearer tokens. Finally, we will protect the Basket endpoints and test the authentication flow by comparing requests made with and without a valid access token.

By the end of the chapter, the Basket microservice will be protected by centralized authentication, providing a secure foundation for controlling access to the application as the distributed architecture continues to grow.

In this chapter, we'll be covering the following topics:

- Understanding Keycloak identity and access management
- Integrating Keycloak Hosting with Aspire
- Configuring Keycloak for the eShop application
- Securing the Basket microservice with Keycloak
- Testing and observing the secured Basket **API**

Understanding Keycloak identity and access management

After completing both synchronous and asynchronous communication between the Catalog and Basket microservices, our next step is to develop identity operations using Keycloak authentication. Keycloak is an open-source identity and access management platform for modern applications and services.

Keycloak supports features such as **single sign-on**, allowing users to authenticate once and access multiple applications; **identity brokering** and **social login** through external identity providers; **user federation** with existing identity stores; and centralized management of users, roles, clients, and permissions. Instead of implementing authentication, credential management, token issuance, and related identity functionality independently in each microservice, you can centralize those responsibilities in Keycloak.

Centralizing these concerns reduces the amount of authentication code that individual application services need to maintain. It also gives you a consistent location from which to manage identities and access policies as the application grows.

For the eShop application, Keycloak will act as the identity provider for the Basket microservice.

The authentication flow relies on **OAuth 2.0** and **OpenID Connect**.

OAuth 2.0 is an authorization framework that allows applications to obtain access to protected resources on a user's behalf. OpenID Connect builds an identity layer on top of OAuth 2.0, allowing applications to authenticate users and obtain information about their identity.

In our application, Keycloak issues a signed access token after the user authenticates. The client then includes that token when calling the Basket API:

```
User
  |
  | Authenticate
  v
Keycloak
  |
  | Access token
  v
Client
  |
  | Authorization: Bearer <token>
```

```
  v
Basket API
```

The Basket microservice validates the token before allowing the request to reach an endpoint that requires authorization. Requests without an acceptable token are rejected with an `HTTP 401 Unauthorized` response.

With the role of Keycloak established, you can now add it to the distributed application.

Integrating Keycloak Hosting with Aspire

We will use Keycloak as our authentication and authorization provider by integrating it with Aspire. For this integration, we will configure both the hosting and client sides. The hosting integration adds and orchestrates the Keycloak resource in AppHost, while the client integration enables the Basket microservice to use Keycloak for authentication.

Let's start with the hosting integration.

To add Keycloak hosting integration to your Aspire solution:

1. Open Visual Studio and navigate to your **AppHost** project.
2. Right-click on the **Dependencies** node and select **Add Aspire package**.
3. In the **NuGet Package Manager**, filter for integration and hosting packages. Check the **Include prerelease** option, as the Keycloak package is a prerelease version.
4. Scroll down or use the **search bar** in the **Browse** section to find the `Aspire.Hosting.Keycloak` package.
5. Select the `Aspire.Hosting.Keycloak` package and install it under your AppHost project.

Once installed, verify that the package reference appears in your AppHost project file. The following line in the project file should confirms the addition of `Aspire.Hosting.Keycloak` as a package reference.

```
<PackageReference Include="Aspire.Hosting.Keycloak" Version="13.0.0-
preview.1.25560.3" />
```

Next, to configure Keycloak as a backing service in your infrastructure code, open the AppHost `Program.cs` file and define the Keycloak resource alongside PostgreSQL, Redis, and RabbitMQ

(under the *Backing Services* section). This ensures that Keycloak runs as a container alongside your other services and persists its data across restarts:

```
var keycloak = builder
    .AddKeycloak("keycloak", 8080)
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
```

`AddKeycloak` adds a Keycloak container named `keycloak` and exposes its HTTP service on port `8080`. `WithDataVolume()` provides persistent storage for its data, while the persistent container lifetime prevents the Keycloak configuration from being discarded unnecessarily between application runs.

After defining the Keycloak resource, you need to add a reference to it in the project that requires authentication. In this case, you are securing only the Basket microservice, so you add the reference specifically to the Basket project definition. This allows the Basket microservice to access environment variables such as the Keycloak connection string and automatically discover and communicate with Keycloak to secure its endpoints.

The following code shows how to add the Keycloak reference and ensure the Basket service waits for Keycloak to be ready:

```
var basket = builder
    .AddProject<Projects.Basket>("basket")
    .WithReference(cache)
    .WithReference(catalog)
    .WithReference(rabbitmq)
    .WithReference(keycloak)
    .WaitFor(cache)
    .WaitFor(rabbitmq)
    .WaitFor(keycloak);
```

The `WithReference(keycloak)` call makes the Keycloak connection information available to the Basket microservice through Aspire's configuration mechanism. `WaitFor(keycloak)` ensures that the Basket microservice does not start depending on Keycloak before the identity provider is ready.

NOTE

Why did we add the Keycloak reference and `wait` condition only to the Basket project? In a typical e-commerce architecture, the Catalog microservice remains public so anyone can browse products without logging in. The Basket microservice, however, handles user-specific data that must be protected from unauthenticated access. Therefore, we secure the Basket service and configure Aspire to wait for Keycloak, while intentionally leaving the Catalog open.

After adding these lines of code and including the Keycloak resources, let's set the AppHost project as the startup project and run it. To spin up the Keycloak container, make sure your Docker Desktop is running in the background. The Aspire dashboard should now display `keycloak` alongside the existing application resources.

Before we open the Keycloak management dashboard, let's verify that the Basket microservice has received the Keycloak service discovery information. Select the `basket` project in the **Resources** view and inspect its environment variables.

You should find a **service discovery endpoint** named `services__keycloak__http__0`, which points to `http://localhost:8080` in this example.

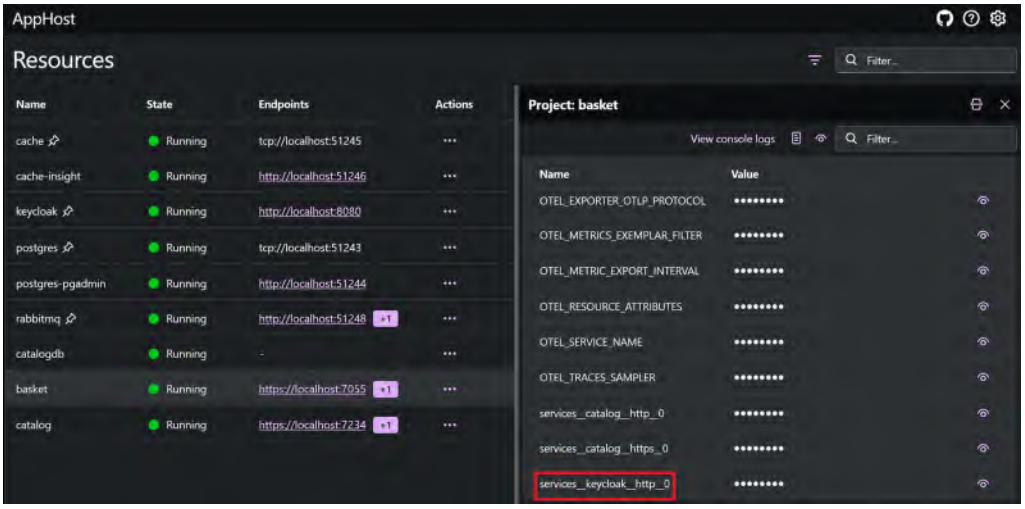


Figure 10.1 – Service discovery endpoint for Keycloak

Next, select the `keycloak` container and inspect its environment variables. Locate `KEYCLOAK_ADMIN` and `KEYCLOAK_ADMIN_PASSWORD`, which provide the administrator credentials for the Keycloak management dashboard. Copy the administrator password to your clipboard.

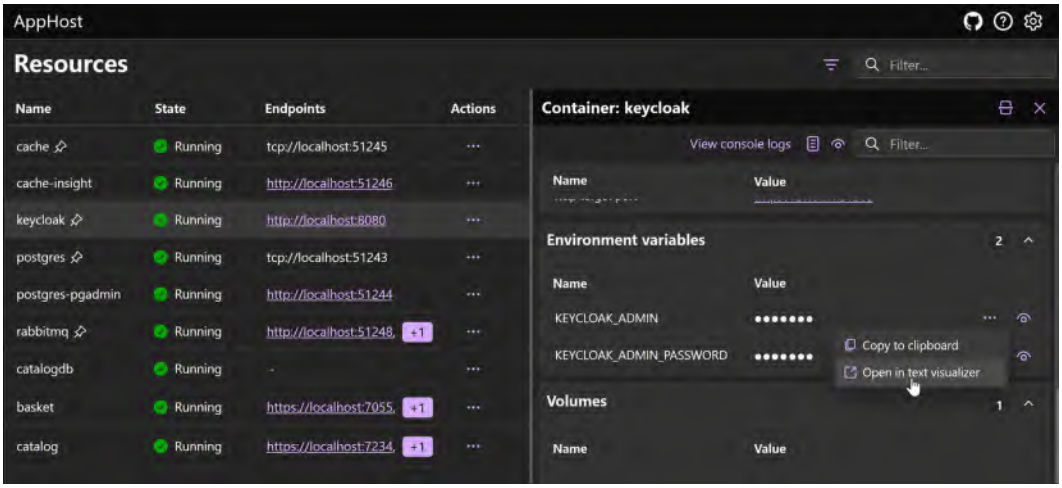


Figure 10.2 – `KEYCLOAK_ADMIN` *and* `KEYCLOAK_ADMIN_PASSWORD` *values for logging in to the Keycloak management dashboard*

Now, open the Keycloak HTTP endpoint from the Aspire dashboard. This takes you to the Keycloak management login page at `http://localhost:8080`.

Enter `admin` as the username, and paste the password you copied from the environment variables.

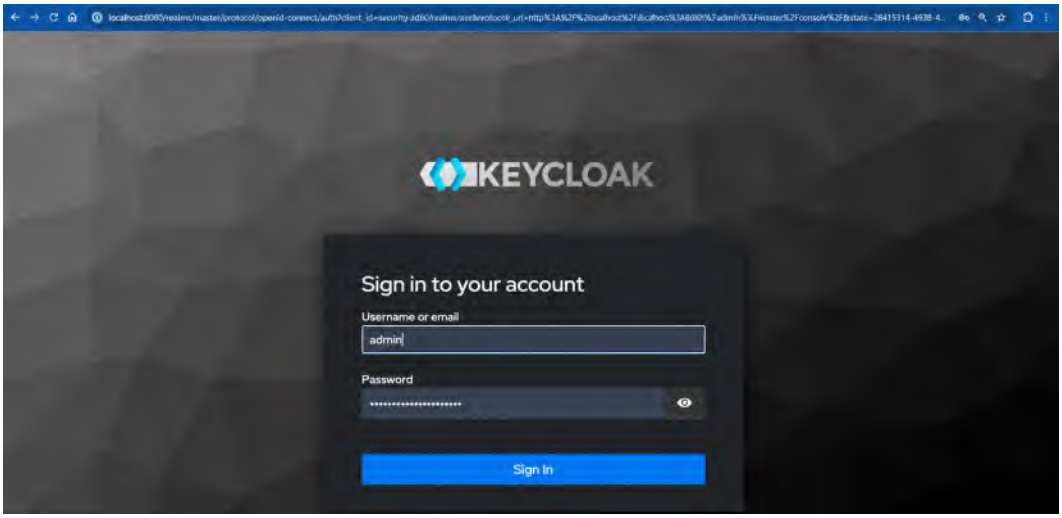


Figure 10.3 – Keycloak management login screen

Select **Sign In** to open the Keycloak management dashboard.

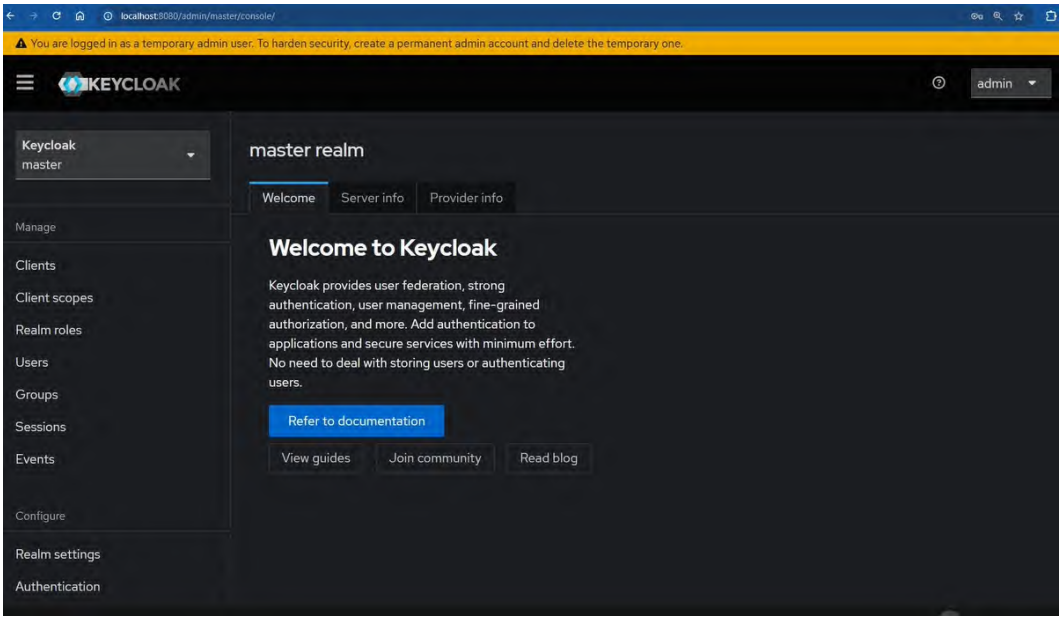


Figure 10.4 – Keycloak after signing in

From this dashboard, you can configure the realms, clients, users, and other identity settings required by the eShop application.

With Keycloak running and accessible, you can now configure the identity information required by our application.

Configuring Keycloak for the eShop application

With Keycloak running as part of the Aspire application, the next step is to configure it for the eShop authentication flow. In this section, we will create the `eshop` realm, register an OpenID Connect client, add a test user, and obtain an access token. We will then inspect the token to confirm that it contains the claims required by the Basket microservice when validating authenticated requests.

Creating a Keycloak realm

Keycloak organizes users, applications, roles, and authentication settings into **realms**. A realm provides an isolated identity-management boundary, allowing one Keycloak server to support multiple applications or environments independently.

We'll begin by creating a new realm definition:

1. Open the **Keycloak management dashboard**.
2. Select the drop-down menu at the top left to access realm options.

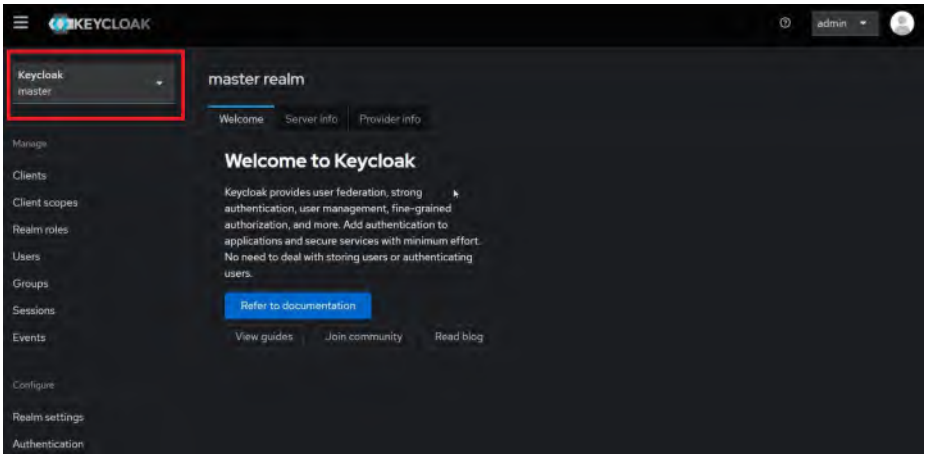


Figure 10.5 – Drop-down menu at the top left (labeled Keycloak master)

3. Select the **Create realm** button.
4. In the **Realm name** field, enter `eshop`.

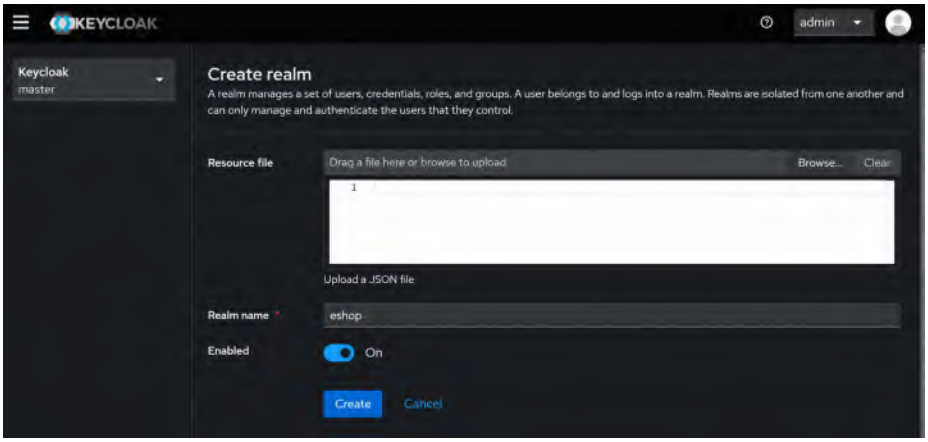


Figure 10.6 – The "Create realm" window

5. Ensure the **Enabled** toggle is set to **On**.
6. Select the **Create** button to create the realm.

Before continuing with the remaining configuration, make sure that the new realm is selected. The client and user that you create next should belong to the `eshop` realm rather than Keycloak's default `master` realm.

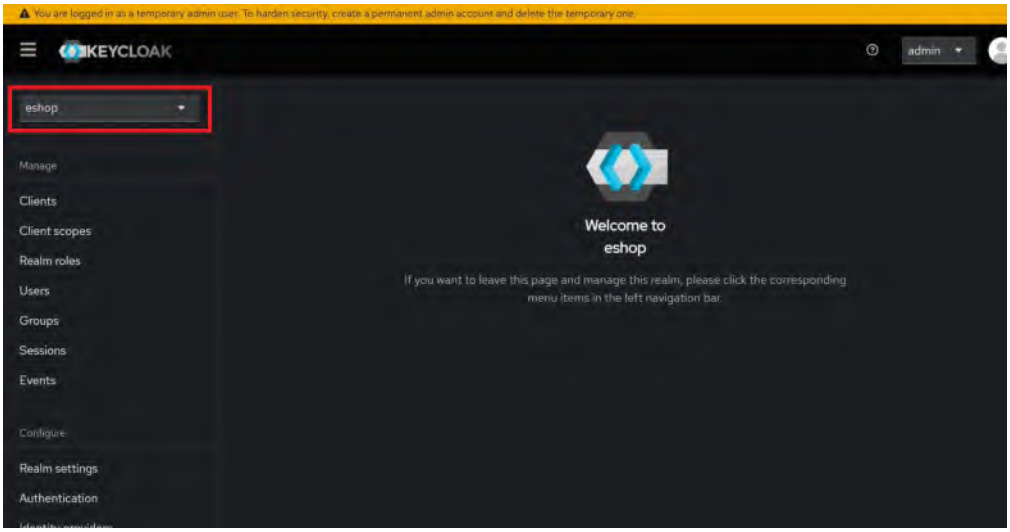


Figure 10.7 – eshop realm selected

Registering the OpenID Connect client

The next step is to register the application that will request authentication from Keycloak.

With the `eshop` realm selected, create a client definition:

1. Select the **Clients** button in the left sidebar.
2. Select the **Create client** button.

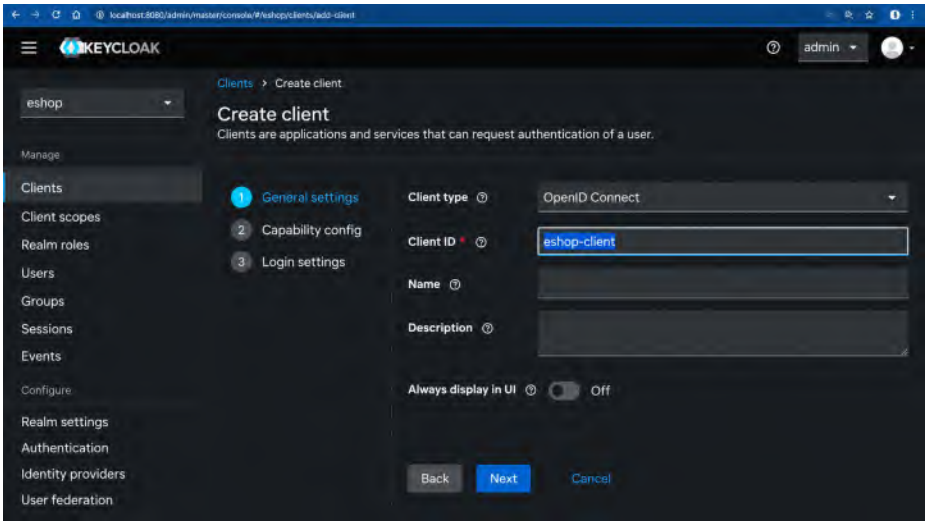


Figure 10.8 – Create client screen

3. In the **Client ID** field, enter `eshop-client`.
4. Leave the **Client type** as **OpenID Connect**.
5. Scroll down and leave the remaining configurations as default.
6. Select the **Next** button to proceed to **Capability config**.

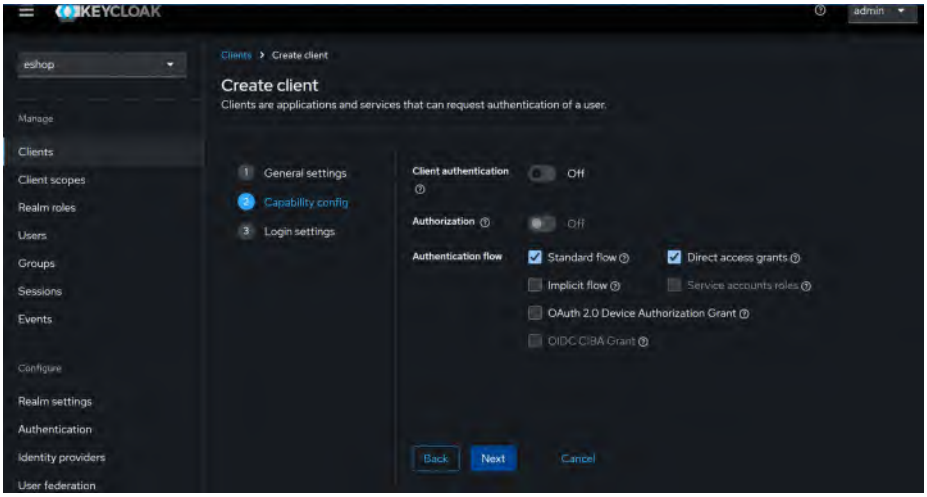


Figure 10.9 – Capability config screen

7. Leave the **Capability config** settings as default (**Standard flow** and **Direct access grants** enabled).
8. Select **Next** to reach login settings.
9. Do not change any login settings.
10. Scroll down and select the **Save** button to create the client.

This client represents the application requesting tokens from Keycloak.

Once you create the client, save the configuration.

Creating a test user

Next, we need to create a user that can authenticate against the `eshop` realm:

1. Select the **Users** button in the left sidebar.
2. Select the **Create new user** button.

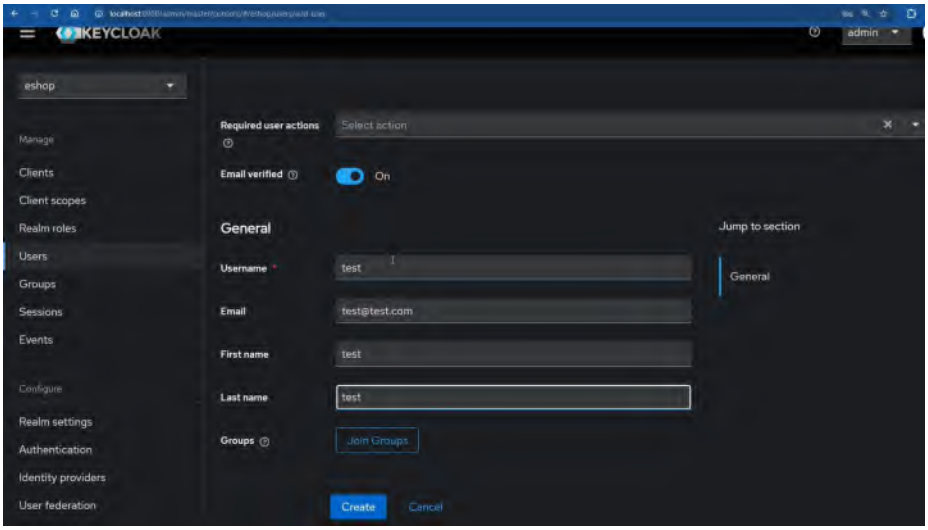


Figure 10.10 – Create new user screen

3. Set **Email verified** to **On**.
4. Enter `test` as the **Username**.
5. Enter `test@test.com` as the **Email**.
6. Enter `test` as the **First name** and `test` as the **Last name**.
7. Select the **Create** button to add the user.
8. Set a password for the new user.
    1. Go to the **Credentials** tab for the user.
    2. Select the **Set password** button.
    3. Enter `1234` as the password and confirm it.
    4. Ensure **Temporary** is set to **Off**.
    5. Select the **Save** button.

> **NOTE**
>
> This deliberately simple password is suitable only for the local sample. Use appropriately strong credentials and credential policies in real environments.

At this point, Keycloak has the three pieces required for the test authentication flow: the `eshop` realm, the `eshop-client` client, and the `test` user.

The next step is to verify that this configuration can successfully issue an **access token** for the user. To do that, you will send a request to the OpenID Connect token endpoint exposed by the `eshop` realm.

Requesting an access token

Keycloak exposes an OpenID Connect token endpoint for each realm. You need to use the Keycloak token endpoint with the configurations you just created. The endpoint format is:

```
https://your-keycloak-server/realms/your-realm-name/protocol/openid-connect/token
```

For your setup, the endpoint is:

```
http://localhost:8080/realms/eshop/protocol/openid-connect/token
```

You can call this endpoint from an `.http` file in Visual Studio.

You need the following configuration parameters for the token request:

```
grant_type = password
client_id = eshop-client
scope = email openid
username = test
password = 1234
```

With these definitions and configurations in place, you are ready to obtain an access token from Keycloak and use it to secure your Basket microservice endpoints.

To obtain a token from your Keycloak identity provider and use it to secure your Basket microservice, you need to configure and send an `HTTPPOST` request with the required parameters.

This request uses the **password grant type**, the previously created `eshop-client`, and the test user's credentials to request an access token from Keycloak.

Next, open Visual Studio and navigate to the `Basket.http` file in your Basket microservice project. At the end of this file, add a new HTTP `POST` request to obtain the token from Keycloak:

```
POST http://localhost:8080/realms/eshop/protocol/openid-connect/token
Content-Type: application/x-www-form-urlencoded
grant_type=password&client_id=eshop-client&scope=openid
email&username=test&password=1234
```

After defining this HTTP `POST` request, send the request from within Visual Studio. You will receive a response containing the access token, expiration time, refresh token, and other details.

The response body includes the following fields:

```
{
    "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...",
    "expires_in": 300,
    "refresh_expires_in": 1800,
    "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    "token_type": "Bearer",
    "id_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...",
    "not-before-policy": 0,
    "session_state": "5885c24a-ce58-4275-840c-cc1cbba06c85",
    "scope": "openid email profile"
}
```

You will use the `access_token` from this response to call secured endpoints in your Basket microservice.

Copy the access token and navigate to `https://jwt.io` to inspect its contents. Paste the access token into the decoder to view the decoded values and payloads.

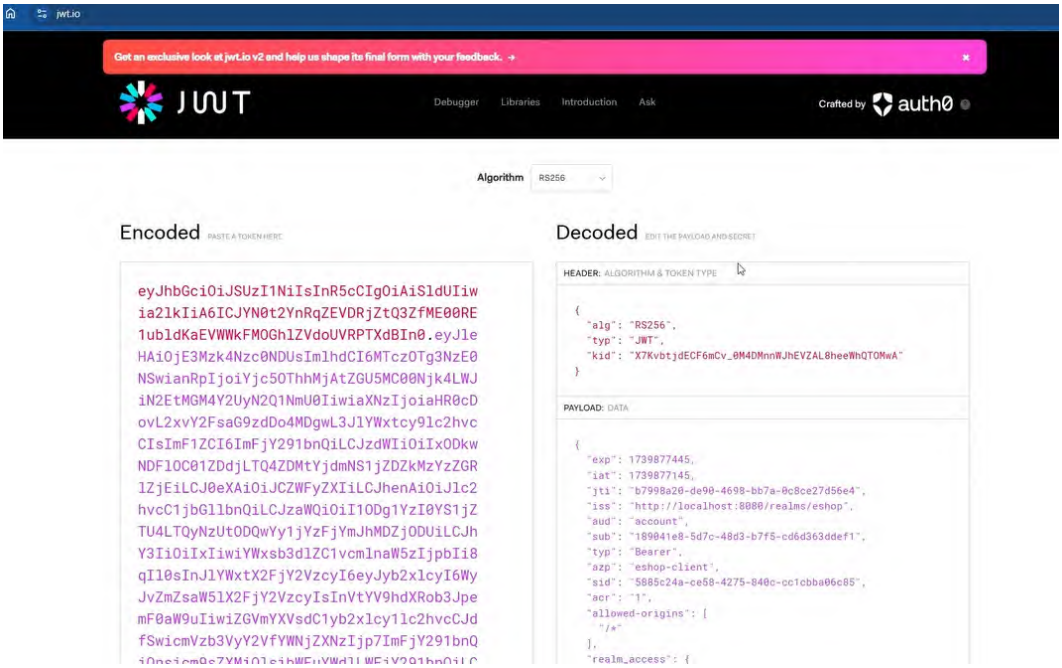


Figure 10.11 – Decoded values and payloads

The decoded JSON Web Token includes fields such as `realm_access`, `roles`, `resource_access`, `scope`, `email_verified`, `name`, `preferred_username`, and `email`. These values reflect the roles and claims assigned to your test user in the Keycloak realm.

This inspection step helps you verify the structure and claims of the token you will use to secure your Basket microservice endpoints.

The next step is to integrate Keycloak authentication into your Basket microservice and secure its endpoints using the access token you have obtained.

Securing the Basket microservice with Keycloak

Now that Keycloak is configured to issue tokens, we can integrate it with the Basket microservice. In this section, we will add the Keycloak client integration, configure authentication and authorization in the ASP.NET Core pipeline, and protect the Basket endpoints so that they require an authenticated caller.

Integrating Keycloak with the Basket microservice

After installing the required Keycloak client package, the next step is to configure the Basket microservice to use Keycloak for authentication and authorization. This involves registering the authentication services with the dependency injection container and adding the corresponding middleware to the HTTP request pipeline.

Right-click on the **Dependencies** node in the Basket microservice and select **Add Aspire package**. Search for the prerelease (check the **Include prerelease** option) `Aspire.Keycloak.Authentication` package. This is the client-side integration that connects the ASP.NET Core authentication system with the Keycloak resource orchestrated through Aspire. Install this package under the Basket microservice. Confirm the installation by checking the Basket project's `.csproj` file, where you should see the following line:

```
<PackageReference Include="Aspire.Keycloak.Authentication" Version="13.0.0-
preview.1.25560.3" />
```

Then, open the `Program.cs` file in the Basket microservice to register the required authentication and authorization services.

The Basket microservice already registers its Redis cache, `BasketService`, Catalog HTTP client, and MassTransit messaging configuration. Add authentication and authorization alongside these existing services. The following code registers Keycloak JWT bearer authentication with the dependency injection container, using the `keycloak` service, the `eshop` realm, and the required token-validation options:

```
builder.Services
    .AddAuthentication()
    .AddKeycloakJwtBearer(
        serviceName: "keycloak",
        realm: "eshop",
        configureOptions: options =>
        {
            options.RequireHttpsMetadata = false;
            options.Audience = "account";
        });
builder.Services.AddAuthorization();
```

In the preceding snippet, `AddAuthentication()` registers ASP.NET Core authentication services, while `AddKeycloakJwtBearer()` configures JWT bearer authentication against the

Keycloak realm. The `serviceName` is set to keycloak to match the Keycloak resource defined in **AppHost**, and the realm is set to eshop. The additional options specify how incoming tokens should be validated: `RequireHttpsMetadata` = false allows the local development setup to use HTTP, while Audience = "account" specifies the audience expected in accepted tokens. We use "account" here because it is Keycloak's default audience, but a production setup would configure a dedicated audience or client scope for the microservices. Finally, `AddAuthorization()` registers the services required to apply authorization policies to protected endpoints.

After registering these services, you must activate **authentication** and **authorization** in the **HTTP** request pipeline. Add the lines `app.UseAuthentication();` and `app.UseAuthorization();` after mapping your endpoints:

```
var app = builder.Build();
app.MapDefaultEndpoints();
app.UseAuthentication();
app.UseAuthorization();
app.MapBasketEndpoints();
app.UseHttpsRedirection();
app.Run();
```

`app.UseAuthentication()` ensures the authentication middleware checks JWT tokens on incoming requests, and `app.UseAuthorization()` enforces authorization policies and attributes.

Registering authentication tells the application how to verify a caller's identity, but it does not automatically restrict access to every API endpoint. You still need to specify which endpoints should be accessible only to authenticated users, for example by applying `RequireAuthorization()`.

Protecting the Basket endpoints

To secure our Basket microservice endpoints, you need to enforce authorization on each operation so that only requests with a valid token are allowed access. This ensures that every endpoint checks for a valid JWT token, and if a token is missing or invalid, the response will be either `401 Unauthorized` or `403 Forbidden`.

Open `BasketEndpoints.cs` in the `Endpoints` folder of the Basket project.

The Basket endpoints currently allow operations such as retrieving a basket, updating it, and deleting it. Add `RequireAuthorization()` to the endpoints that should be available only to

authenticated users. For each endpoint definition, add the `.RequireAuthorization()` extension method at the end of the endpoint configuration chain.

For example, to secure the `GET` operation by username, add `.RequireAuthorization()` after the `Produces` section:

```
group.MapGet("/{userName}", async (string userName, BasketService service) =>
{
    var shoppingCart = await service.GetBasket(userName);
    if (shoppingCart is null)
    {
        return Results.NotFound();
    }
    return Results.Ok(shoppingCart);
})
    .WithName("GetBasket")
    .Produces<ShoppingCart>(StatusCodes.Status200OK)
    .Produces(StatusCodes.Status404NotFound)
    .RequireAuthorization();
```

Repeat this process for the `POST (Upsert)` and `DELETE` operations. For each, add `.RequireAuthorization()` at the end of the endpoint definition to ensure that all routes require a valid JWT token.

The following code secures the `POST (Upsert)` endpoint:

```
group.MapPost("/", async (ShoppingCart shoppingCart, BasketService service) =>
{
    await service.UpdateBasket(shoppingCart);
    return Results.Created("GetBasket", shoppingCart);
})
    .WithName("UpdateBasket")
    .Produces<ShoppingCart>(StatusCodes.Status201Created)
    .RequireAuthorization();
```

The following code secures the `DELETE` endpoint:

```
group.MapDelete("/{userName}", async (string userName, BasketService service) =>
{
    await service.DeleteBasket(userName);
    return Results.NoContent();
```

```
})
    .WithName("DeleteBasket")
    .Produces(StatusCodes.Status204NoContent)
    .RequireAuthorization();
```

By adding `.RequireAuthorization()` to each endpoint, you ensure that every route in the Basket microservice is protected and only accessible with a valid Keycloak-issued JWT token.

We can move on to testing the authentication flow and observing the results in the Aspire dashboard.

Testing and observing the secured Basket API

With authentication enabled and the Basket endpoints protected, the next step is to verify that the security flow works as expected. We will first confirm that a request without an access token is rejected, then obtain a token from Keycloak and use it to access the protected endpoints. Finally, we will inspect the resulting requests in the Aspire dashboard.

Testing the secured Basket endpoints

We will use the `Basket.http` file in Visual Studio to send requests to the Basket microservice and verify the authentication behavior. As you know by now, this file contains HTTP requests for interacting with the Basket microservice, and you will update it to include the access token and authorization headers required for secure communication.

First, open `Basket.http` and send a request to one of the protected endpoints without including an access token:

```
GET {{Basket_HostAddress}}/swn
Accept: application/json
```

Because the endpoint requires an authenticated caller and the request does not contain a valid bearer token, Basket responds with:

```
401 Unauthorized
```

To authenticate the Basket requests, use the Keycloak token request introduced earlier in the *Requesting an access token* section. Send that request again from `Basket.http` to obtain a valid access token from Keycloak.

```
POST http://localhost:8080/realms/eshop/protocol/openid-connect/token
Content-Type: application/x-www-form-urlencoded
```

```
grant_type=password&client_id=eshop-client&scope=email
openid&username=test&password=1234
```

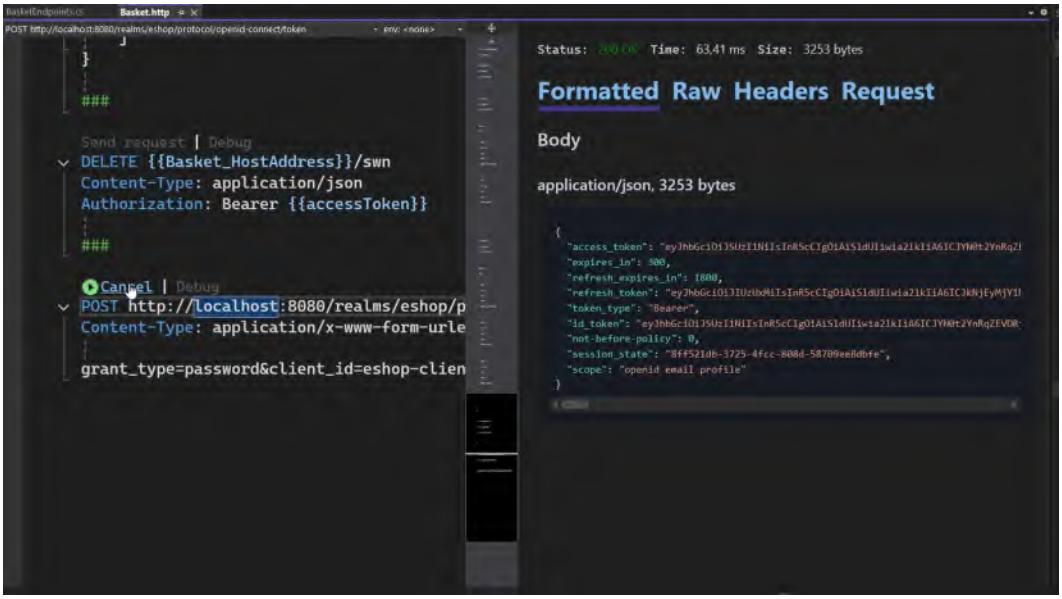


Figure 10.12 – Keycloak access token

After receiving the response, copy the access token and replace the current value of the `@accessToken` variable at the top of `Basket.http`.

```
@Basket_HostAddress = https://localhost:7055/basket
@accessToken = eyJhbGciOiJSUzI...
```

Next, for each endpoint request (`GET`, `POST`, `DELETE`), add an `Authorization` header that uses the `Bearer` scheme and passes the access token variable. This ensures that every request to the Basket microservice is authenticated. Without that header, the protected endpoint rejects the request; with a valid Keycloak token, the request is authenticated and can proceed.

```
GET {{Basket_HostAddress}}/swn
Accept: application/json
Authorization: Bearer {{accessToken}}
```

Send the request again. With a valid Keycloak-issued token, the request should succeed with a `200 OK` response and return the Basket data.

Apply the same authorization header to the `POST` request:

```
POST {{Basket_HostAddress}}
Content-Type: application/json
Authorization: Bearer {{accessToken}}
{
  "UserName": "swn",
  "Items": [
    {
      "Quantity": 2,
      "Color": "Red",
      "Price": 0,
      "ProductId": 1,
      "ProductName": "Solar Powered Flashlight"
    },
    {
      "Quantity": 1,
      "Color": "Blue",
      "Price": 0,
      "ProductId": 2,
      "ProductName": "Hiking Poles"
    }
  ]
}
```

A successful request returns `201 Created`. You can send another `POST` request with modified basket items to verify that an authenticated caller can also update the basket.

Finally, add the bearer token to the DELETE request:

```
DELETE {{Basket_HostAddress}}/swn
Content-Type: application/json
Authorization: Bearer {{accessToken}}
```

A successful deletion returns `204 No Content`. You can verify the result by sending the authenticated `GET` request again. If the basket has been deleted, the request returns `404 Not Found`.

These tests confirm that the Basket endpoints reject unauthenticated requests and allow requests carrying a valid Keycloak access token.

Observing authentication in the Aspire dashboard

You can also use the Aspire dashboard to observe how these requests are handled across the distributed application.

Open the **Traces** view and inspect the requests sent to the Basket microservice. For example, the request sent without a bearer token should show a `401 Unauthorized` response. A subsequent request containing a valid Keycloak access token should pass authentication and continue to the Basket endpoint, where the requested data is retrieved from the distributed cache.

Together, the HTTP responses and Aspire traces confirm that Keycloak authentication is being enforced for the protected Basket endpoints.

Summary

In this chapter, we secured the Basket microservice with Keycloak and integrated centralized identity and access management into the Aspire distributed application.

We began by examining the role of Keycloak and how OAuth 2.0 and OpenID Connect support token-based access to protected APIs. We then added Keycloak to AppHost as a backing service and connected it to the Basket microservice through Aspire.

Next, we configured Keycloak for the eShop application by creating the `eshop` realm, registering the `eshop-client` OpenID Connect client, and adding a test user. After confirming that Keycloak could issue a JWT access token, we configured the Basket microservice to validate Keycloak-issued bearer tokens and protected its endpoints with `RequireAuthorization()`.

Finally, we tested the secured API by comparing unauthenticated requests with requests that included a valid bearer token. This confirmed that the Basket endpoints reject unauthorized access and allow authenticated requests to proceed.

With communication, messaging, caching, and authentication now in place across the backend services, the next step is to provide users with an interface for interacting with the application. In the next chapter, we will begin developing the Blazor web application, connect it to the Catalog microservice through Aspire service discovery, and build the first product-facing functionality for the distributed system.

11

Developing the Client Blazor Web Application

In the previous chapter, we secured the Basket microservice using Keycloak, configured **JWT** bearer authentication, and protected its endpoints from unauthenticated access. With the Catalog and Basket microservices now communicating through synchronous and asynchronous mechanisms and the Basket **API** secured, the backend foundation of our distributed application is in place.

In this chapter, we will turn our attention to the client-facing layer by developing a Blazor web application for the eShop. We will create the WebApp project and add it to the Aspire application, configure its references to the Catalog and Basket microservices, and use Aspire **service discovery** so that the frontend can communicate with these services without relying on hardcoded endpoints.

We will then integrate the Catalog microservice with the Blazor application by linking the shared `Product` model, creating and registering a typed `CatalogApiClient`, and using it to retrieve product data. Building on this integration, we will develop the `Products.razor` page with **streaming rendering**, add it to the application navigation, and verify the communication flow through the Aspire dashboard. Finally, we will configure Redis-backed **output caching** to reduce unnecessary calls to the Catalog microservice and improve the responsiveness of the `Products` page.

By the end of the chapter, we will have a functional Blazor frontend integrated with the distributed backend, providing the foundation for the additional client-facing and AI-powered capabilities introduced later in the book.

In this chapter, we'll be covering the following topics:

- Creating the Blazor web application and Aspire integration
- Integrating the Catalog microservice with the Blazor application
- Registering the Catalog API client with Aspire service discovery
- Developing the Products Razor page
- Testing the Products page
- Enabling output caching with Redis
- Applying and testing output caching on the Products page

Creating the Blazor Web Application and Aspire Integration

In this section, we are going to develop a client Blazor web application that integrates with our distributed microservices architecture. Using Visual Studio, we will create a new Blazor web application, orchestrate it with Aspire, and connect it to internal microservices such as Catalog and Basket. You will focus on configuring project references, setting up **service defaults**, and preparing the application for further integration.

To create the Blazor web application and add it to your solution:

1. Open Visual Studio.
2. Navigate to the `frontend` folder in your solution; right-click the folder and select **Add | New Project**.
3. In the **Add a new project** dialog, select **Blazor Web App** as the project template.
4. Name the project `WebApp` and choose the desired location.
5. Click the **Next** button.
6. Leave all configurations as default: **.NET 10** as the **Framework**, select **None** for **Authentication type**, enable **Configure for HTTPS**, and check the **Include sample pages** option.
7. Ensure you select the option **Enlist in Aspire orchestration**. Enlisting the application will add the client application to the Aspire environment and include AppHost and service default configurations.
8. Select the **Create** button to generate the Blazor web application under the `frontend` folder.

After creating the project, check the `AppHost` and ServiceDefaults to confirm that your new web application is included in the Aspire environment. Double-click the AppHost project to see that it references the newly created `WebApp` project, as shown here:

```
<ProjectReference Include="..\WebApp\WebApp.csproj" />
```

Then, open the AppHost `Program.cs` file to view the infrastructure as code, and scroll to the end of the project definition to find the line of code added by Aspire for enlisting the web application:

```
Builder.AddProject<Projects.WebApp>("webapp")
```

After confirming your Blazor Web App is included in the Aspire environment, you need to review and adjust the service defaults and project references to ensure proper orchestration and integration with other microservices.

The Aspire tooling automatically adds a reference to the `ServiceDefaults` project in your Blazor Web App's project file. The following line in the `WebApp.csproj` file is present to include the shared service defaults:

```
<ProjectReference Include="..\ServiceDefaults\ServiceDefaults.csproj" />
```

This reference ensures that your web application inherits the standard configuration and cross-cutting concerns defined for all services in the distributed system.

Next, open the `Program.cs` file in your `WebApp` project. Here, you will find that Aspire has inserted two important lines of code. The first is

```
builder.AddServiceDefaults();
```

This line adds the default service configuration to the dependency injection container.

The other important line is

```
app.MapDefaultEndpoints();
```

This line configures the default HTTP endpoints for the application, which includes endpoints for structured logging and health checks.

You might want to refactor the placement of these lines for clarity and maintainability. Move the `AddServiceDefaults()` call inside the section where you add services to the container, and ensure that `MapDefaultEndpoints()` is included under the HTTP request pipeline

configuration. This keeps your configuration organized and aligns with Aspire's conventions for cross-cutting concerns.

Here's the updated structure in the web app's `Program.cs` file:

```
using WebApp.Components;
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.AddServiceDefaults();
builder.Services.AddRazorComponents()
.AddInteractiveServerComponents();
var app = builder.Build();
// Configure the HTTP request pipeline.
app.MapDefaultEndpoints();
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.
    app.UseHsts();
}
app.UseHttpsRedirection();
app.UseAntiforgery();
app.MapStaticAssets();
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();
```

After understanding the Aspire structure and the integration of your web application, you need to ensure that your client application is properly connected to the Catalog **microservice**. To do this, modify the AppHost's `Program.cs` file to reference the Catalog and Basket microservices from the web application.

Update the project definition in AppHost's `Program.cs` as follows:

```
var catalog = builder
    .AddProject<Projects.Catalog>("catalog")
    .WithReference(catalogDb)
    .WithReference(rabbitmq)
    .WaitFor(catalogDb)
    .WaitFor(rabbitmq);
var basket = builder
```

```
    .AddProject<Projects.Basket>("basket")
    .WithReference(cache)
    .WithReference(catalog)
    .WithReference(rabbitmq)
    .WithReference(keycloak)
    .WaitFor(cache)
    .WaitFor(rabbitmq)
    .WaitFor(keycloak);
var webapp = builder
    .AddProject<Projects.WebApp>("webapp")
    .WithExternalHttpEndpoints()
    .WithReference(catalog)
    .WithReference(basket)
    .WaitFor(catalog)
    .WaitFor(basket);
builder.Build().Run();
```

This configuration does the following:

- Adds the Catalog and Basket microservices as projects, specifying their dependencies and wait conditions.
- Adds the WebApp project, exposes its HTTP endpoints for external access, and references both the Catalog and Basket microservices.
- Ensures that the web application waits for the Catalog and Basket services to be ready before launching.

To enable your Blazor web application to communicate with internal microservices, you configure it within Aspire so it can discover the routes and environment variables for the Catalog and Basket microservices. The setup allows the web application to connect and interact with these services. You also add a `WaitFor` extension method to ensure that the microservices are running (ready) before the client web application launches.

You have now created the Blazor web application and orchestrated it into the Aspire environment. With this foundation, you are ready to consume the Catalog microservice from the web application, allowing you to fetch product listings or details into your Blazor components.

Integrating the Catalog microservice with the Blazor application

You will now focus on integrating the client web application with the Catalog microservice using a Catalog API client. This involves linking the `Product` model from the Catalog project into your WebApp project and creating a Catalog API client class to fetch product data via HTTP calls.

To share the `Product` model from the Catalog microservice with your web application:

1. In Visual Studio, right-click the WebApp project and create a new folder named `Models`.
2. Instead of duplicating the `Product` class, right-click the `Models` folder and select **Add | Existing Item**.
3. Navigate to the Catalog microservice project, then to `Models/Product`, and select the `Product.cs` file.
4. At the file selection dialog, choose **Add As Link** from the **Add** dropdown.

By adding the file as a link, you ensure that any changes made to the `Product` domain model in the Catalog microservice will automatically propagate to all linked locations, including your web application. The source of this file remains in the Catalog `Models` folder, but it is now accessible in your web application for use in the Catalog API client class.

You can verify that the link has been added by checking the WebApp project file:

```
    <Compile Include="..\Catalog\Models\Product.cs" Link="Models\Product.cs" />
```

The following code shows the `Product` class as it appears after being linked to the WebApp `Models` folder:

```
namespace Catalog.Models;
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; } = default!;
    public string Description { get; set; } = default!;
    public decimal Price { get; set; }
    public string ImageUrl { get; set; } = default!;
}
```

This approach ensures consistency across your microservices and client application, as updates to the `Product` model are reflected everywhere the link is used.

To integrate with the Catalog microservice, you need to create a `CatalogApiClient` class in your Blazor Web App. This client will handle communication between your client application and the Catalog microservice.

Right-click your WebApp project and add a new folder named `ApiClients`, and inside this folder, add a new class named `CatalogApiClient`.

Begin by importing the `Catalog.Models` namespace at the top of your new class file to access the shared `Product` domain model.

The following code defines the `CatalogApiClient` class, which uses an `HttpClient` injected via its **primary constructor**:

```
using Catalog.Models;
namespace WebApp.ApiClients;
public class CatalogApiClient(HttpClient httpClient)
{
    public async Task<List<Product>> GetProducts()
    {
        var response = await httpClient.GetFromJsonAsync<List<Product>>($"/
products");
        return response!;
    }
    public async Task<Product> GetProductById(int id)
    {
        var response = await httpClient.GetFromJsonAsync<Product>($"/products/
{id}");
        return response!;
    }
}
```

This class provides two asynchronous methods for integration:

- `GetProducts()`: This sends an HTTP `GET` request to the `/products` endpoint of the Catalog microservice, retrieves a list of products, and returns it as a list of `Product` objects.
- `GetProductById(int id)`: This sends an HTTP `GET` request to `/products/{id}` to fetch a single product by its ID and returns the corresponding `Product` object.

By implementing these methods, your client application can retrieve product data from the Catalog microservice efficiently.

With the API client now defined, the next step is to make it available to the Blazor application and connect it to the Catalog service through Aspire service discovery.

Registering the Catalog API client with Aspire service discovery

Rather than hardcoding the Catalog service's hostname or port, the WebApp can address it by the logical service name defined in AppHost. Aspire then resolves that name to the appropriate running endpoint.

Open the WebApp project's `Program.cs` file and locate the section where services are registered with the **dependency injection** container. Add `CatalogApiClient` as a typed HTTP client:

```
builder.Services.AddHttpClient<CatalogApiClient>(client =>
{
    client.BaseAddress = new("https+http://catalog");
});
```

`AddHttpClient<CatalogApiClient>` registers `CatalogApiClient` with the dependency injection container and provides it with a configured `HttpClient`. The client can therefore be injected into Razor components or other application services that need to retrieve product information from Catalog.

The important part of this configuration is the base address:

```
https+http://catalog
```

Here, `catalog` is the **logical name** assigned to the `Catalog` microservice in `AppHost`. The `https+http` scheme allows service discovery to prefer HTTPS when it is available and fall back to HTTP otherwise.

Because the client uses a logical service name rather than a fixed network address, the web app does not need to know which hostname or port has been assigned to the Catalog microservice at runtime. This is particularly useful in a distributed application, where endpoints can vary between development, testing, and deployment environments.

For the logical name to be available to the web app, however, the relationship between the two projects must also be declared in `AppHost`.

The `Catalog` project is registered with the logical name `catalog`:

```
var catalog = builder
    .AddProject<Projects.Catalog>("catalog");
```

The web app then references the `Catalog` resource:

```
var webapp = builder
    .AddProject<Projects.WebApp>("webapp")
    .WithExternalHttpEndpoints()
    .WithReference(catalog)
    .WithReference(basket)
    .WaitFor(catalog)
    .WaitFor(basket);
var catalog = builder.AddProject<Projects.Catalog>("catalog");
```

The following two calls serve different purposes:

- `WithReference(catalog)` makes the `Catalog` service-discovery information available to `WebApp`.
- `WaitFor(catalog)` delays the `WebApp`'s startup until the `Catalog` resource is ready.

Together, these configurations establish the connection between `WebApp` and `Catalog`. `AppHost` defines the relationship between the resources, while the typed HTTP client in `WebApp` uses the logical name `catalog` when making requests

The communication path can therefore be viewed as follows:

```
WebApp
   |
   | CatalogApiClient
   | https+http://catalog
   v
Aspire service discovery
   |
   v
Catalog microservice
```

When `CatalogApiClient` sends a request such as

```
httpClient.GetFromJsonAsync<List<Product>>("/products");
```

the relative `/products` path is combined with the configured logical base address. Aspire service discovery resolves `catalog` to the running `Catalog` endpoint, allowing the request to reach the microservice without a hardcoded hostname or port.

This separation is useful because each part has a clear responsibility: `AppHost` describes which resources can communicate, while `CatalogApiClient` defines how the web app communicates with `Catalog` over **HTTP**.

With `CatalogApiClient` registered and service discovery configured, the web app is ready to consume product data from the `Catalog` microservice. In the next section, we will inject this client into the `Products` Razor page and use it to retrieve and display the product catalog.

Developing the Products Razor page

In this section, we will create a Razor component for the `Products` page, inject `CatalogApiClient`, retrieve the product collection asynchronously, and present the results in a table containing each product's image, name, description, and price.

Let's begin:

In the WebApp project, navigate to the `Components/Pages` folder. Right-click the folder, select **Add** | **Razor Component**, name the component `Products.razor`, and add it to the project.

Remove all the existing code in this file. Begin the component with the page route, required namespaces, the injected API client, and streaming rendering:

```
@page "/products"
@using Catalog.Models
@using WebApp.ApiClients
@inject CatalogApiClient CatalogApiClient
@attribute [StreamRendering(true)]
```

The `@page` directive exposes the component through the `/products` route. The `@inject` directive provides the component with the `CatalogApiClient` that we registered earlier.

The `[StreamRendering(true)]` attribute allows the page to render its initial content while the asynchronous product request is still being completed. This is useful because the page does not need to remain blank while it waits for `Catalog` to return its data.

Next, add the page title and heading:

```
<PageTitle>Products</PageTitle>
<h1>Products</h1>
<p>Here are some of our amazing outdoor products that you can purchase.</p>
```

The page needs to handle three possible states: the products are still loading, no products are returned, or products are available to display. The following conditional rendering handles these states:

```
@if (products == null)
{
    <p><em>Loading...</em></p>
}
else if (products.Count == 0)
{
    <p>
        <em>There is a problem loading our products. Please try again later.</em>
    </p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Image</th>
                <th>Name</th>
                <th>Description</th>
                <th>Price</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var product in products)
            {
                <tr>
                    <td>
                        <img height="80"
                             width="80"
                             src="https://raw.githubusercontent.com/
MicrosoftLearning/AI-102-AIEngineer/main/data/images/@product.ImageFile" />
                    </td>
```

```
                    <td>@product.Name</td>
                    <td>@product.Description</td>
                    <td>@product.Price.ToString("C2")</td>
                </tr>
            }
        </tbody>
    </table>
}
```

When `products` is null, the page displays a loading message. If the request completes but returns no `products`, the page displays an error message. If products are available, the page generates a table listing each product's image, name, description, and price.

We now need to populate the `products` collection.

The following code in the `@code` block declares the `products` variable and loads the products asynchronously when the page initializes:

```
@code {
    private List<Product>? products;
    protected override async Task OnInitializedAsync()
    {
        // Simulate asynchronous loading to demonstrate streaming rendering
        products = await CatalogApiClient.GetProducts();
    }
}
```

When the component initializes, `OnInitializedAsync()` calls `CatalogApiClient.GetProducts()`. This method sends an HTTP `GET` request to the `/products` endpoint of the `Catalog` microservice and retrieves the available `products`. The returned `Product` objects are then assigned to the products collection.

Once that asynchronous operation completes, the page renders the resulting data in the table.

Now, if you look at the top of the page, you will see two `using` statements. You can move these `using` statements to the `_Imports.razor` file to simplify your code and make these imports available globally to all pages in your Blazor application.

After editing your `using` statements and developing the `Products.razor` page, you need to adjust the layout to make the **Products** page accessible from the main navigation menu. To do this, update the navigation menu component so users can easily find and access the `Products` page from the Blazor application's **UI**.

Open the `NavMenu.razor` file located under `Components/Layout` in your Blazor project. At the end of the existing navigation links, copy one of the existing `<NavLink>` elements. Add a new `<NavLink>` for the **Products** page, setting the `href` to `"products"` and the link text to `Products`:

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="products">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true">&lt;/span>
        Products
    </NavLink>
</div>
```

By adding this link, you will see a **"Products"** section in your main menu, allowing you to select and navigate directly to the `Products.razor` page.

With the **Products** page implemented, the application now has a frontend component capable of retrieving data through `CatalogApiClient` and presenting it to the user. The next step is to run the distributed application and verify that the entire communication path works as expected.

Testing the Products page

Now that the **Products** page is implemented, we can verify that the web app communicates with the Catalog microservice through the typed HTTP client and correctly displays the returned product data.

To test the **Products** page and verify integration with the Catalog microservice:

Open Visual Studio, right-click the `AppHost` project and select **Set as Startup Project**. Run the application.

Running the project will launch your web application, which serves as the UI client for your **distributed system**. Once the application is running, you can access the Blazor web application through the provided HTTP link.

When the Aspire dashboard opens, locate the **webapp** resource. Selecting it allows you to inspect its endpoints and environment variables. Under **Environment variables**, you will find service discovery links for both the Basket and Catalog microservices. These links enable the web application to discover and call the microservices dynamically.

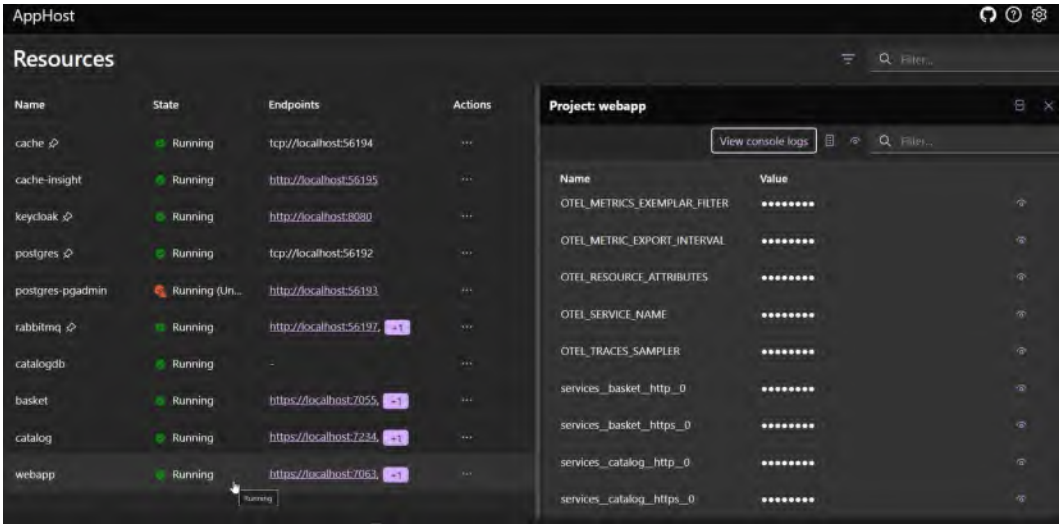


Figure 11.1 – Service discovery links for the Basket and Catalog microservices

To access the Blazor web application, select the HTTP link associated with the `webapp` project. In the running application, you will see the **Products** page listed in the navigation menu.

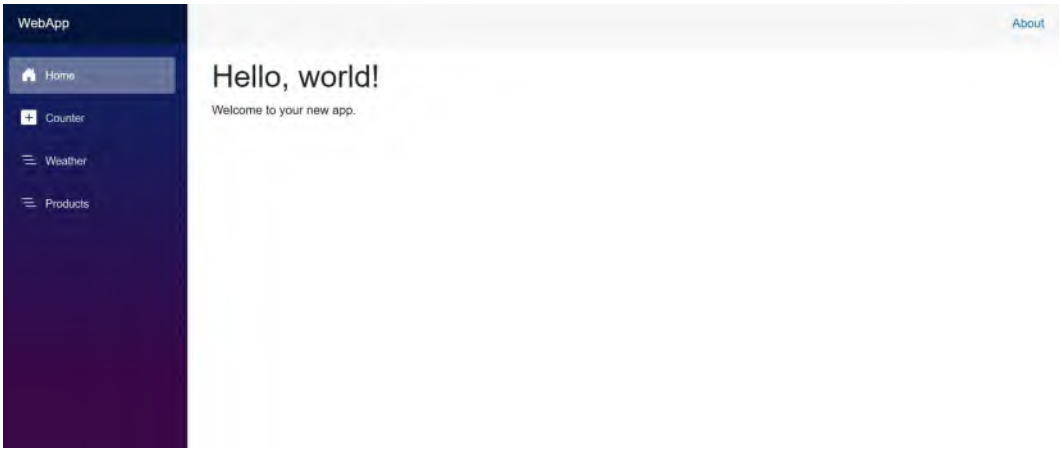


Figure 11.2 – WebApp UI

Selecting the **Products** option in the navigation menu will redirect you to the **Products** page, where the Catalog microservice returns product data.

Before testing, you may want to set a breakpoint in the `Products.razor` page. In Visual Studio, scroll to the `OnInitializedAsync` method in `Products.razor` and set a breakpoint. This allows

you to debug and verify that the page is correctly calling the `CatalogApiClient` and retrieving product data from the Catalog microservice.

```
protected override async Task OnInitializedAsync()
  {
      // Simulate asynchronous loading to demonstrate streaming rendering
      products = await CatalogApiClient.GetProducts();
  }
```

After placing breakpoints, return to your Blazor web application and select the **Products** menu item. When you do this, your breakpoint will be triggered, confirming that the application is retrieving products by communicating with the Catalog microservice and returning a list of products.

You will see that there are nine products being fetched. Press *F5* to continue, and these nine products will be displayed on the **Products** Razor page in a table format. This confirms that the integration with the Catalog microservice is functioning as intended.

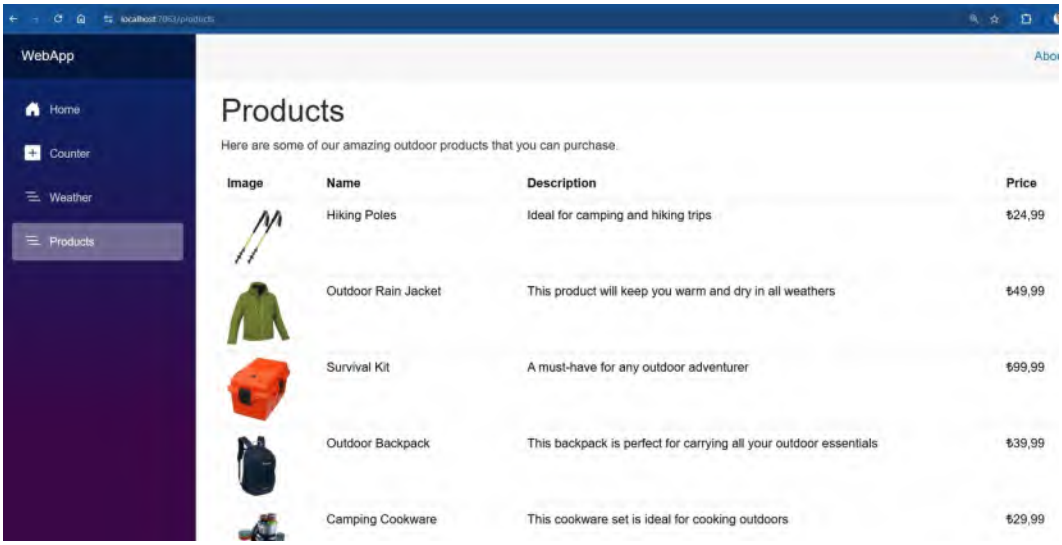


Figure 11.3 – Products page

The following code in your `Products.razor` page demonstrates how the products are loaded asynchronously using the CatalogApiClient:

```
@code {
    private List<Product>? products;
    protected override async Task OnInitializedAsync()
    {
        // Simulate asynchronous loading to demonstrate streaming rendering
        products = await CatalogApiClient.GetProducts();
    }
}
```

You can further verify this integration by inspecting the AppHost dashboard and navigating to the **Traces** section and opening the **Products** page again. The trace should show an HTTP `GET` request to `/products`, with the web app calling the Catalog microservice.

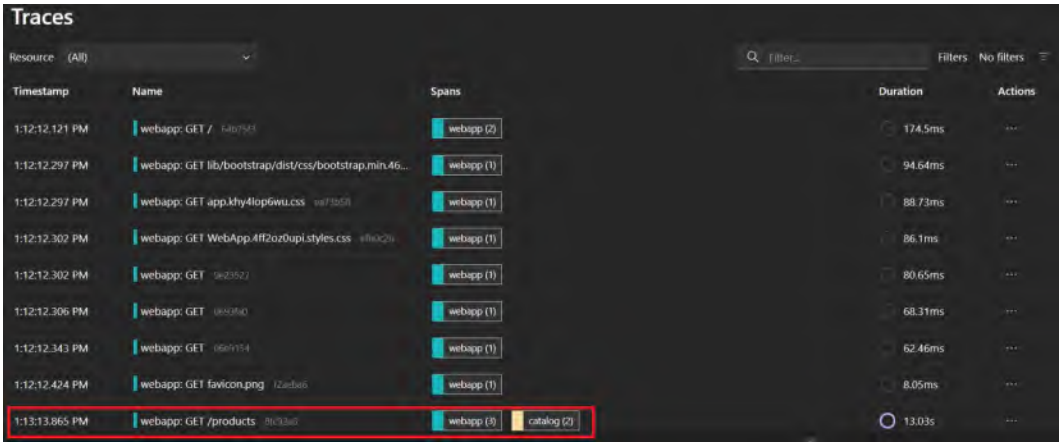


Figure 11.4 – Traces view showing HTTP `GET` *request to* `/products`

Clicking this entry reveals that the `GET` products request for the **Products** page is being made, and the web application is calling the Catalog microservice to retrieve products from the Catalog PostgreSQL database. The data is then returned to your application and rendered in the table.

This confirms that your Blazor web application's **Products** page is successfully fetching data from the Catalog microservice using the Aspire environment and the typed HTTP client object.

The **Products** page is now functional, but every request can still result in the page being generated again and the Catalog microservice being called. For requests that can tolerate a short period of cached data, we can reduce this repeated work by introducing output caching.

Enabling output caching with Redis

The **Products** page currently retrieves product information from the Catalog microservice whenever the page needs to be generated. When several equivalent requests arrive within a short period, repeatedly executing the same application logic and making the same downstream request can be unnecessary.

Output caching helps avoid this repeated work by temporarily storing the generated response. Subsequent equivalent requests can then be served directly from the cache instead of executing the full request path again.

For the **Products** page, this means that a cached response can avoid repeated calls to Catalog during the configured cache period.

The cache duration is also important. If product data changes frequently, a longer cache duration might cause users to see stale information. The cache policy should therefore balance improved performance against the need to keep the displayed data up to date.

In this application, you will use **Redis** as the backing store for output caching.

Open the `AppHost` project and confirm that Redis is already installed as a backing service. The application already defines a Redis resource named `cache` in `Program.cs`:

```
var cache = builder
    .AddRedis("cache")
    .WithRedisInsight()
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
```

Next, update the existing `WebApp` configuration to add a reference to this Redis resource:

```
var webapp = builder
    .AddProject<Projects.WebApp>("webapp")
    .WithExternalHttpEndpoints()
    .WithReference(cache)
    .WithReference(catalog)
    .WithReference(basket)
    .WaitFor(catalog)
    .WaitFor(basket);
```

The `WithReference(cache)` call makes the Redis connection information available to the `WebApp`. The application can then use this Redis instance as the backing store for output caching.

With Redis now added and referenced by the `WebApp`, you are ready to configure output caching for the **Products** page.

Next, integrate Redis output caching into the `WebApp` project. Right-click **Dependencies**, select **Add Aspire package**, and search for the `Aspire.StackExchange.Redis.OutputCaching` NuGet package. Install the package in the `WebApp` project.

The following line in your `WebApp`'s `.csproj` file confirms the package installation:

```
<PackageReference Include="Aspire.StackExchange.Redis.OutputCaching"
Version="13.5.4" />
```

Next, configure output caching in the `Program.cs` file of your `WebApp` project. Start by adding Redis output cache registration to the dependency injection container, referencing the Redis resource named `cache` that you defined in the `AppHost`:

```
builder.AddRedisOutputCache("cache");
```

The connection name must match the Redis resource name defined in the AppHost. Because the `WebApp` already references this resource through `WithReference(cache)`, Aspire makes the Redis connection information available to the `WebApp` at runtime.

Next, add the output caching **middleware** to the HTTP request pipeline:

```
app.UseOutputCache();
```

Your updated `Program.cs` file in the `WebApp` project should now include these lines to enable Redis-backed output caching:

```
builder.Services.AddHttpClient<CatalogApiClient>(client =>
{
    client.BaseAddress = new("https+http://catalog");
});
builder.AddRedisOutputCache("cache");
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();
var app = builder.Build();
```

```
app.MapDefaultEndpoints();
app.UseOutputCache();
```

With these changes, your Blazor web application is now configured to use Redis for output caching, allowing you to apply caching policies such as duration to specific pages or the entire site. This setup helps you balance fresh data with improved performance by reducing unnecessary calls to the microservices.

Applying and testing output caching on the Products page

To enable output caching for the **Products** page, add an output caching attribute near the top of the page. After the attribute used for stream rendering, add another attribute that configures output caching with a duration of five seconds.

For this example, the page output will be cached for five seconds. This short interval makes the caching behavior easy to observe while still demonstrating the trade-off between reducing repeated work and keeping product information reasonably fresh.

The following code in `Products.razor` demonstrates where to add the output cache attribute:

```
@page "/products"
@inject CatalogApiClient CatalogApiClient
@attribute [StreamRendering(true)]
@attribute [OutputCache(Duration = 5)]
<PageTitle>Products</PageTitle>
<h1>Products</h1>
<p>Here are some of our amazing outdoor products that you can purchase.</p>
@if (products == null)
{
    <p><em>Loading...</em></p>
}
else if (products.Count == 0)
{
    <p><em>There is a problem loading our products. Please try again later.</em></
p>
}
else
{
    <table class="table">
    <thead>
```

```
    <tr>
    <th>Image</th>
    <th>Name</th>
    <th>Description</th>
    <th>Price</th>
    </tr>
    </thead>
    <tbody>
    @* Product rows rendered here *@
    </tbody>
    </table>
}
```

You need to import the required `using` statement for ASP.NET Core output caching. Move the following `using` statement to your `Imports.razor` file so it is available throughout your application:

```
@using Microsoft.AspNetCore.OutputCaching
```

This ensures that output caching is activated for the entire page. With the `[OutputCache(Duration = 5)]` attribute, ASP.NET Core will cache the rendered HTML for 5 seconds. When you send an HTTP `GET` request to the `/products` page, your Redis-backed output cache stores the generated HTML. After five seconds, the cache expires and the next request fetches fresh data from the Catalog microservice.

To test output caching in your Blazor web application, set `AppHost` as the startup project and run the distributed application. Make sure Docker Desktop is running in the background.

The Aspire dashboard opens and shows the projects and backing services starting. Once the `WebApp` and Redis cache are running successfully, select the **HTTPS endpoint** for the client application.

Navigate to the **Products** page in your client application. On the first load, the web app sends a `GET` request to the Catalog microservice to retrieve the product data. This response is then stored in the Redis-backed output cache.

Refresh the page within five seconds. The response should be returned immediately from the output cache, without another call to the Catalog microservice. If you have placed a breakpoint in the code that retrieves products from Catalog, the breakpoint should not be triggered for this cached request.

After the five-second cache duration expires, refresh the page again. The cached response is no longer valid, so the web app sends a new request to the Catalog microservice to retrieve fresh product data. At this point, the breakpoint should be triggered again, and the new response is cached for the next five seconds.

You can also observe this behavior in the Aspire dashboard under **Traces**. The first HTTP `GET` request shows the web app calling the Catalog microservice to retrieve the product data. A subsequent request made within the cache duration is served from the Redis-backed output cache, so no corresponding downstream call to Catalog is required. Once the cache expires, the next request again includes a call to the Catalog microservice.

Summary

In this chapter, you completed the client-side integration of the eShop application by connecting the web app to the Catalog and Basket microservices. You used Aspire service discovery to reference the microservices without hardcoding their endpoints. You also configured startup dependencies so that the web app waits for the required services to become available before starting.

You also configured typed HTTP clients to communicate with the backend services and introduced Redis-backed output caching for the `Products` page. By caching generated responses for a short period, the application can avoid unnecessary repeated calls to the Catalog microservice while still keeping product information reasonably fresh.

At this point, the eShop application is running as a complete distributed system. Aspire orchestrates the WebApp, Catalog and Basket microservices, and supporting services such as PostgreSQL, Redis, RabbitMQ, and Keycloak. With the application working locally, the next step is to move beyond the development environment and deploy the complete solution to the cloud.

Chapter 12 takes that next step by deploying the eShop Aspire application to Azure Container Apps, where you will provision the required Azure resources, deploy the microservices and backing services, and verify the running application using the Azure portal and Aspire dashboard.

12

Deploying the eShop Aspire Application to Azure Container Apps

You are now ready to take the eShop application beyond local development and deploy it to Azure. So far, you have built and integrated the application as a distributed Aspire solution, bringing together the WebApp, Catalog and Basket microservices, and supporting services such as PostgreSQL, Redis, RabbitMQ, and Keycloak. With the application working locally under Aspire orchestration, the next step is to move the complete solution into the **Azure Container Apps** environment.

In this chapter, you will learn how Azure Container Apps can host your distributed application and how to deploy the Aspire project using the **Azure Developer CLI** (**azd**). You will follow the deployment workflow from initialization through provisioning and deployment, including creating an **Azure Resource Group** and **Azure Container Registry**, publishing the application containers, provisioning backing services such as PostgreSQL and Redis, and deploying the WebApp and microservices to Azure Container Apps.

You will also examine how the cloud environment differs from local development, particularly when working with backing services and persistent data volumes. You will adjust the Aspire configuration where necessary to make the application suitable for deployment.

Finally, you will verify the deployed services through the **Azure portal** and **Aspire dashboard**, inspect endpoints and configuration, and use application logs to troubleshoot the deployment. By the end of the chapter, you will have taken the eShop application from a locally orchestrated Aspire solution to a distributed application running in Azure Container Apps.



Figure 12.1 – Architecture diagram showing the eShop microservices (WebApp, Catalog, Basket, RabbitMQ, PostgreSQL, Redis, Keycloak) being deployed to Azure Container Apps, with Azure Container Apps icons and deployment arrow

In this chapter, we'll be covering the following topics:

- Overview of Azure Container Apps
- Managing data volumes and backing services in Azure Container Apps
- Deploying our Aspire application to Azure Container Apps
- Verifying the deployment in the **Azure portal**
- Testing the deployment
- Cleaning up Azure resources with the `azd down` command

Overview of Azure Container Apps

In this section, we are going to learn what Azure Container Apps is and how to deploy our Aspire project to it.

Azure Container Apps is Microsoft's *managed serverless platform* for hosting containerized applications and microservices. It is well suited to distributed solutions built with Aspire because it handles much of the underlying container orchestration, allowing you to focus on application development rather than infrastructure management. It also provides built-in scaling and integrated observability, making it a practical deployment target for Aspire-based microservices.

Azure Container Apps offers several advantages for deploying Aspire-based microservices. Each microservice can scale independently according to demand, and the platform supports event-driven scaling through **KEDA** (`https://keda.sh/`), allowing services to scale in response to triggers such as queue messages. It also provides built-in revision management for deploying and managing application updates. Optional Dapr integration (`https://`

`aspire.dev/integrations/frameworks/dapr/dapr-get-started/`) can further support distributed application patterns such as service invocation and pub/sub messaging.

The Azure Developer CLI (`azd`) also simplifies deployment. Rather than manually managing complex orchestration files, you can use `azd` from your local development environment or **CI/CD** pipeline to provision Azure resources and deploy the application.

When you deploy an Aspire application, its microservices are packaged as container images, while the backing services and other dependencies defined in AppHost are provisioned or connected according to the deployment configuration. Azure Container Apps then runs the application containers in a managed environment, while Azure services such as Azure Container Registry provide the supporting infrastructure required for deployment.

To move your Aspire application to Azure Container Apps, you will use three main Azure Developer CLI commands: `azd init`, `azd up`, and `azd down`:

- `azd init`: Initializes the Aspire application for Azure deployment. It prompts for details such as the environment name and Azure region. It also creates the local configuration required by Azure Developer CLI, including the `.azure` environment folder and, where applicable, an `azure.yaml` project configuration file.
- `azd up`: Provisions the Azure resources required by the application and deploys the Aspire solution. It builds the application containers, pushes the images to Azure Container Registry, creates resources such as the Azure Container Apps environment and other configured dependencies, and deploys the microservices with the required environment configuration and service-discovery settings.
- `azd down`: Removes the Azure environment and its provisioned resources, including the deployed container apps and associated dependencies. The `azd down` command is useful for cleaning up development, test, or prototype environments and avoiding unnecessary Azure costs.

With the deployment workflow established, the next step is to consider how the backing services in your Aspire application handle persistent data when they move from local development to Azure Container Apps.

Managing data volumes and backing services in Azure Container Apps

A **data volume** is storage mounted to a container so that application data can persist independently of the container itself. This is important for stateful backing services such as PostgreSQL, Redis, RabbitMQ, and Keycloak, which might need to retain databases, messages, configuration, or other state even when a container is restarted or recreated.

During local development with Aspire, you can use the `WithDataVolume()` extension method to attach persistent storage to these backing-service containers. However, this local Docker-volume approach does not translate directly to Azure Container Apps. Azure Container Apps uses its own storage mechanisms, such as Azure Files. The behavior and permissions of these mounted file systems can differ from those of local Docker volumes.

The following code in your AppHost `Program.cs` file how these backing services might initially be configured with data volumes during local development:

```
var builder = DistributedApplication.CreateBuilder(args);
// Backing Services
var postgres = builder
    .AddPostgres("postgres")
    .WithPgAdmin()
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
var catalogDb = postgres.AddDatabase("catalogdb");
var cache = builder
    .AddRedis("cache")
    .WithRedisInsight()
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
var rabbitmq = builder
    .AddRabbitMQ("rabbitmq")
    .WithManagementPlugin()
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
var keycloak = builder
    .AddKeycloak("keycloak", 8080)
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
```

When the same configuration is deployed to Azure Container Apps, some stateful containers might encounter compatibility issues with the mounted storage. For example, a discussion of this issue on GitHub (`https://github.com/microsoft/aspire/issues/6671`) shared by the Aspire community documents this problem with PostgreSQL, showing that using `WithDataVolume()` can cause the container to fail during startup because PostgreSQL cannot apply the required permissions to the mounted data directory.

This means that `WithDataVolume()` should not be assumed to provide the same persistence behavior in Azure Container Apps that it provides during local development. For production workloads, a managed service such as Azure Database for PostgreSQL is generally a better choice for stateful data. If a containerized backing service must retain its own state, the persistent-storage configuration should be designed and tested specifically for Azure Container Apps rather than relying on the local Docker-volume setup.

To address this, we can wrap the `WithDataVolume` calls in a conditional statement that checks if the execution context is in run mode.

In the `AppHost Program.cs` file, first, comment out the direct use of `WithDataVolume` for each backing service:

```
var postgres = builder
    .AddPostgres("postgres")
    .WithPgAdmin()
    //.WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
```

Then, add an `if` statement to conditionally apply `WithDataVolume` only when running locally:

```
if (builder.ExecutionContext.IsRunMode)
{
    // Data volumes don't work on ACA for Postgres so only add when running
locally
    postgres.WithDataVolume();
}
```

You should apply this pattern to all backing services that previously used `WithDataVolume`. The following code demonstrates the adjustment for all relevant services:

```
var builder = DistributedApplication.CreateBuilder(args);
// Backing Services
var postgres = builder
```

```
    .AddPostgres("postgres")
    .WithPgAdmin()
    //.WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
var catalogDb = postgres.AddDatabase("catalogdb");
var cache = builder
    .AddRedis("cache")
    .WithRedisInsight()
    //.WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
var rabbitmq = builder
    .AddRabbitMQ("rabbitmq")
    .WithManagementPlugin()
    //.WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
var keycloak = builder
    .AddKeycloak("keycloak", 8080)
    //.WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent);
if (builder.ExecutionContext.IsRunMode)
{
    // Data volumes don't work on ACA for backing services, so only add when
running locally
    postgres.WithDataVolume();
    cache.WithDataVolume();
    rabbitmq.WithDataVolume();
    keycloak.WithDataVolume();
}
```

This approach allows you to avoid exceptions when deploying to Azure Container Apps, while still supporting local development scenarios where data volumes are needed.

If you require production-grade backing services, consider using Azure-managed alternatives such as Azure Database for PostgreSQL Flexible Server instead of a containerized PostgreSQL, Azure Service Bus instead of RabbitMQ, and Azure Identity instead of Keycloak. For now, you can proceed with the conditional logic in your `Program.cs` file to ensure compatibility with Azure Container Apps.

You are now ready to deploy your application to Azure Container Apps.

Deploying our Aspire application to Azure Container Apps

Now that you understand how Azure Container Apps hosts Aspire-based applications and how backing services should be handled in Azure, you are ready to deploy the eShop application. In this section, you will prepare your local environment, initialize the Aspire project for Azure deployment, provision the required Azure resources, and deploy the application using Azure Developer CLI (azd). You will then verify the deployed resources and services in the Azure portal.

Prerequisites and setup verification

Before starting the deployment, make sure you have an Azure account, tools, and Aspire application available.

To get started, you need to meet the following prerequisites:

1. Have an Azure subscription that allows you to provision resources such as resource groups and Azure Container Apps. If you do not already have an account, you can create one using the following link: `https://azure.microsoft.com/en-in/pricing/purchase-options/azure-account`. The Azure account creation page provides options to start with a free account or choose a pay-as-you-go plan.
2. Install the Azure Developer CLI (azd) on your computer (`https://learn.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest`).
3. Ensure that the Aspire application you want to deploy is available locally.
4. After completing these prerequisites, verify that the Azure Developer CLI is installed and that you can authenticate with Azure.

Next, open a terminal and check the installed version:

```
azd version
```

Then authenticate with your Azure account:

```
azd auth login
```

After successful authentication, you will see confirmation that you are logged in.

You can also visit the Azure portal at `portal.azure.com` to verify the current state of your subscription and confirm that the resources for this deployment have not yet been created.

Once authentication is complete, navigate to the root directory of your Aspire solution. For example, if your project is in `C:\Users\PC\source\repos\eshop-distributed`, use the following command:

```
cd C:\Users\PC\source\repos\eshop-distributed
```

Running the deployment commands from the solution root allows `azd` to locate the Aspire AppHost project and the configuration associated with the application.

Initializing the Aspire application for Azure deployment

With the prerequisites verified and the terminal positioned at the solution root, the next step is to initialize the application for Azure deployment. For example, after navigating to the solution root, your command prompt might look like this:

```
C:\Users\PC\source\repos\eshop-distributed> azd init
```

`The azd init` command scans the current directory for an Aspire application, detects the available services, and guides you through creating an Azure Developer CLI environment. During initialization, you confirm the detected application and provide an environment name, such as `myaspire`. The command then generates the files required for the deployment workflow, including `azure.yaml`, which defines how azd builds and deploys the application to Azure.

Provisioning and deploying the application

Once initialization is complete, the application is ready for provisioning and deployment. Run the following command in the same directory:

```
azd up
```

`The azd up` command provisions the Azure resources required by the application and deploys the services. You will be prompted to select your Azure subscription and region.

After you make these selections, Azure Developer CLI begins provisioning the required infrastructure. For this application, the deployment creates resources such as a resource group, an Azure Container Registry, a Log Analytics workspace, and an Azure Container Apps environment.

During provisioning, `azd` creates the required Azure resources, including a resource group. In this example, the resource group is named `rg-myaspire`.

Once the infrastructure is available, `azd` proceeds with deploying the application services. Each microservice and backing service is packaged and deployed as a separate container app.

During deployment, the terminal displays progress information similar to the following:

```
azd up
Select an Azure Subscription to use: 1. Free Trial
Retrieving locations...
Provisioning Azure resources
Subscription: Free Trial
Location: East US 2
Creating/Updating resources
Done: Resource group: rg-myaspire
Done: Container Registry: acrdee5rx7z6rbzw
Done: Log Analytics workspace: law-dee5rx7z6rbzw
Creating/Updating resources (Container Apps Environment)
Deploying services (azd deploy)
Done: Deploying service basket
Endpoint:
https://basket.internal.jollyglacier-9497df4a.eastus2.azurecontainerapps.io/
Done: Deploying service cache
Endpoint:
https://cache.internal.jollyglacier-9497df4a.eastus2.azurecontainerapps.io/
Done: Deploying service catalog
Endpoint:
https://catalog.internal.jollyglacier-9497df4a.eastus2.azurecontainerapps.io/
Done: Deploying service keycloak
Endpoint:
https://keycloak.internal.jollyglacier-9497df4a.eastus2.azurecontainerapps.io/
Done: Deploying service postgres
Endpoint:
https://postgres.internal.jollyglacier-9497df4a.eastus2.azurecontainerapps.io/
Done: Deploying service rabbitmq
Endpoint:
https://rabbitmq.internal.jollyglacier-9497df4a.eastus2.azurecontainerapps.io/
Done: Deploying service webapp
Endpoint:
https://webapp.jollyglacier-9497df4a.eastus2.azurecontainerapps.io/
Aspire Dashboard:
https://aspire-dashboard.ext.jollyglacier-9497df4a.eastus2.azurecontainerapps.io/
```

The generated names and URLs will vary depending on your Azure environment and deployment.

In this deployment, the backing services and internal microservices use endpoints containing `.internal`, indicating that they are accessible only from within the Azure Container Apps environment. The web app does not use an internal endpoint because it is configured for external access.

After the deployment completes successfully, `azd` displays the endpoints for the deployed services together with a link to the **Aspire dashboard**. You can use these endpoints to confirm that the application has been deployed and that the services are running.

Verifying the deployment in the Azure portal

After deployment, open the Azure portal and navigate to the resource group created for the environment, such as `rg-myaspire`.

The resource group should contain the Azure resources provisioned during deployment, including the Container Apps environment, individual container apps, Container Registry, Log Analytics workspace, and other supporting resources.

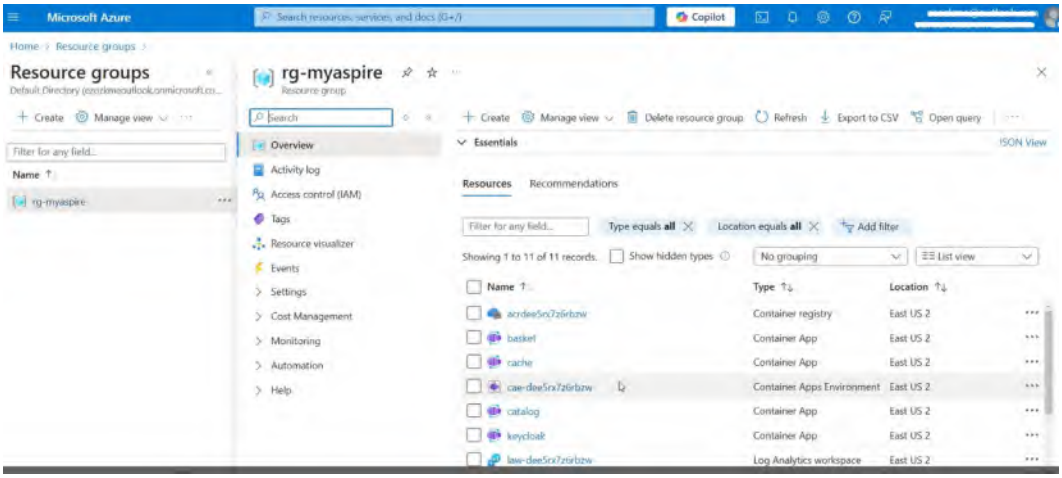


Figure 12.2 – Azure portal with the resource group `rg-myaspire` *selected, listing resources such as the container registry, container apps for each microservice and backing service, and the container apps environment*

You can inspect the individual container apps to verify their status, endpoints, configuration, and logs. The externally accessible web app endpoint allows you to open the deployed eShop application and confirm that the frontend is running.

You can also open the **Aspire dashboard** using the URL displayed at the end of the deployment output. The dashboard provides a consolidated view of the distributed application and helps you inspect the status and behavior of its services.

With the application successfully deployed and the resources verified, you can proceed to test the communication between the deployed microservices and review application logs for troubleshooting.

Testing the deployment

With the deployment verified at the environment level, you can now examine individual services to confirm that they are running correctly and configured as expected.

Let's inspect the configuration of a specific microservice. Select the **basket container app** from the list. This opens the overview page for the Basket microservice, where you can review its status, environment, application URL, and link to the Aspire dashboard.

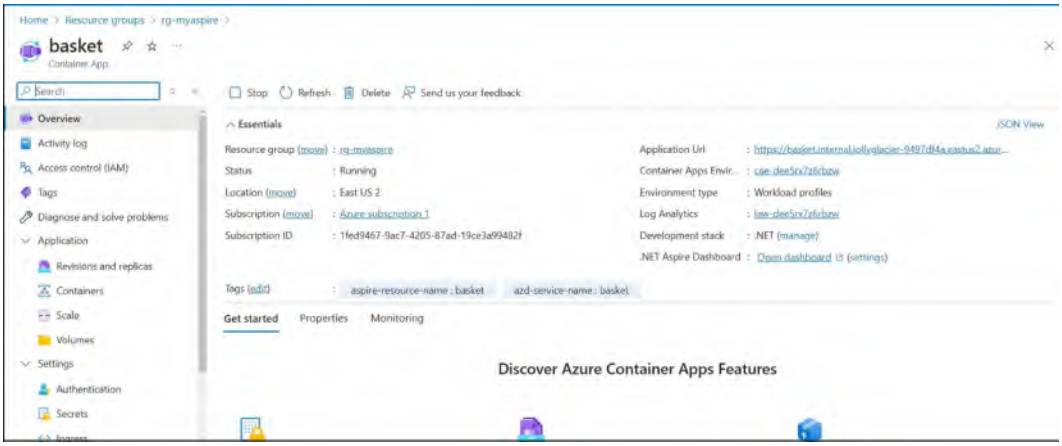


Figure 12.3 – Overview page of the basket container app, including its status, environment, and related links

To examine the environment variables injected into the basket microservice, in the left sidebar, select the **Containers** section. Then, select the **Environment variables** tab.

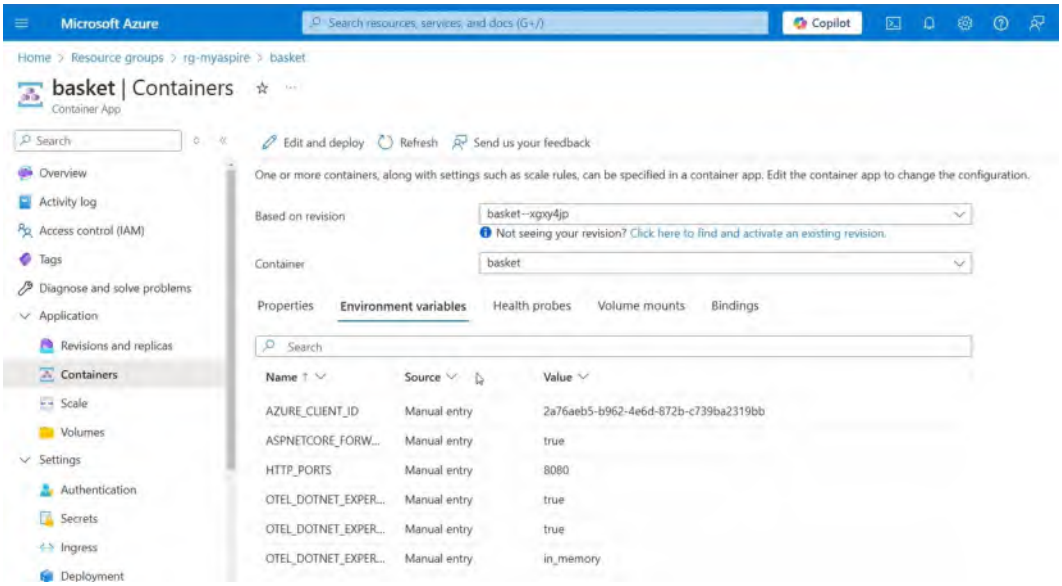


Figure 12.4 – Basket microservice environment variables

Here, you will see all environment variables related to the Basket microservice. The variables include service discovery endpoints for the Catalog and keycloak integrations, with values generated by Azure Container Apps. The endpoints use internal links such as `catalog.internal` and `keycloak.internal`, reflecting service discovery values injected by Aspire.

You will also find two connection strings for the distributed cache and RabbitMQ backing services. Azure Container Apps stores these as secrets, ensuring secure configuration management. All of these settings are handled automatically by the deployment process.

You can repeat this process to inspect the configuration of other container apps in the deployment.

To further explore your deployment, open the **Aspire dashboard** using the **URL** displayed in the terminal at the end of the `azd up` deployment output. The dashboard lists the microservices and backing services in the application together with their endpoints and runtime information.

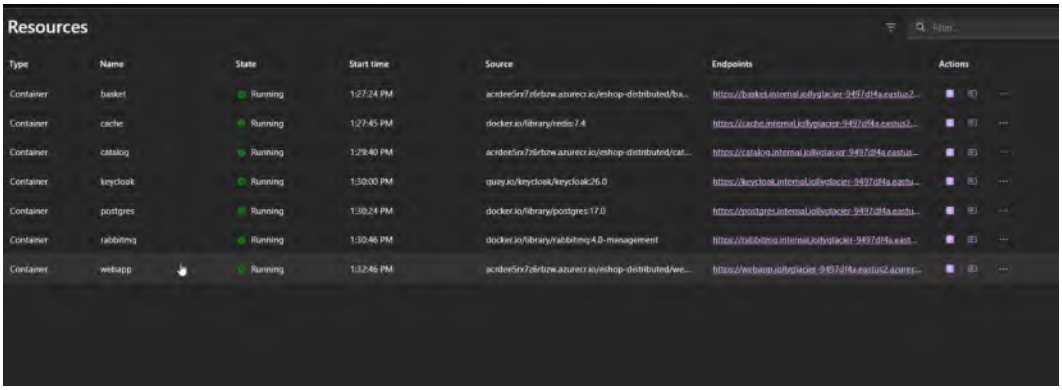


Figure 12.5 – Aspire dashboard listing the microservices and backing services in the application

From the Aspire dashboard, open the webapp frontend. Because the web application is configured for external access, you can open it directly in the browser and interact with the deployed Blazor application. When you open the web application, you will see the Blazor web interface.

To test the integration with the Catalog microservice, select the **Products** link in the Blazor web application. The application will communicate with the Catalog microservice, retrieve the product list, and display it in **HTML** format.

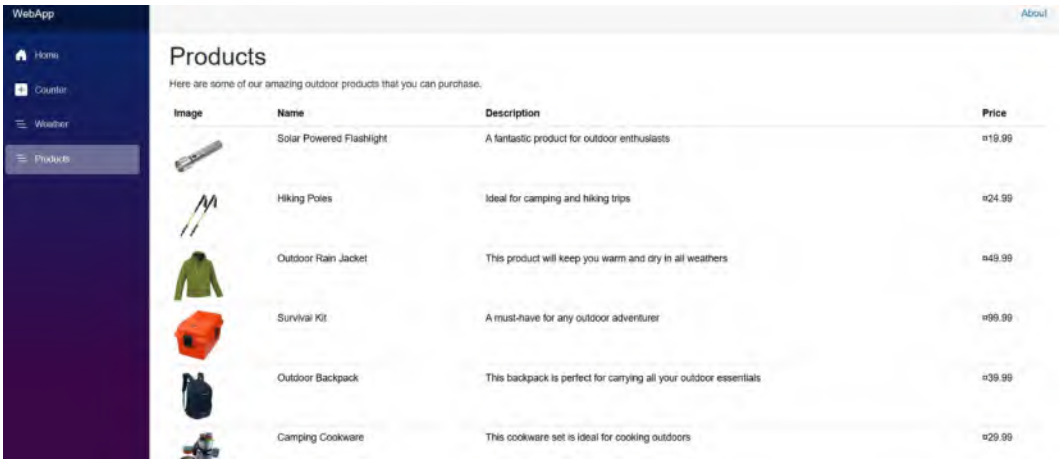


Figure 12.6 – Retrieved product list

If you refresh the page, the response will come from the output cache, demonstrating successful integration with Redis for output caching.

You can also use the **Traces** view in the **Aspire dashboard** to inspect the request flow. When the Products page requests data, the trace shows the call from the WebApp to the Catalog microservice. If a subsequent request is served from the output cache, the WebApp does not repeat the downstream call to Catalog, allowing you to confirm that output caching is working as expected.

These checks confirm that the application has been deployed successfully to Azure Container Apps and that the web app, microservices, backing services, and application configuration are working together as expected.

With the deployment now tested and the application behavior verified, the final step is to remove the Azure resources created for this environment when they are no longer needed.

Cleaning up Azure resources with the azd down command

After deploying your Aspire application to Azure Container Apps, it is important to clean up all Azure resources to avoid unnecessary costs and maintain a tidy environment. You can use the `azd down` command to remove all deployed resources, including container apps, resource groups, and supporting infrastructure.

To delete all resources associated with your deployment, open your Windows terminal and navigate to the solution folder where you previously ran the `azd up` command. Run the following command to initiate the cleanup process:

```
azd down
```

When prompted, review the confirmation message listing the resource group and all associated services that will be deleted. Type `y` or `yes` and press *Enter* to confirm and proceed with the deletion.

The command removes the Azure resources provisioned for the environment, including the Container Apps, Container Apps environment, registry, and other supporting infrastructure associated with the deployment.

The deletion process can take up to 15 minutes. Once the process is complete, refresh the Azure portal to confirm that the resource group no longer exists. You should see that the `rg-myaspire` resource group is no longer present, and there are no remaining resources associated with your application.

Summary

In this chapter, we established a complete workflow for deploying the eShop Aspire application to Azure Container Apps using the Azure Developer CLI (azd). We began by exploring Azure Container Apps as a deployment target for Aspire-based distributed applications, then initialized the application with `azd init` and used `azd up` to provision the required Azure resources and deploy the application. We also examined how data persistence and backing services need to be handled differently in the cloud, and verified the deployed services, configuration, endpoints, and request flows through the Azure portal and Aspire dashboard.

We also distinguished between externally accessible components, such as the WebApp, and services that communicate through internal endpoints within the Azure Container Apps environment. Finally, after testing the deployed application and its integrations, we used `azd down` to remove the provisioned Azure resources when they were no longer required. Together, these steps gave us an end-to-end workflow for deploying, inspecting, testing, and cleaning up an Aspire-based distributed application in Azure.

With the eShop microservices, communication patterns, and cloud deployment now in place, we are ready to extend the application with intelligent capabilities. In *Chapter 13*, we will introduce generative **AI** into the eShop application using `Microsoft.Extensions.AI`, explore local and cloud-based language models, and begin building AI-powered features such as customer-support chat and semantic search.

13

Building Generative AI Features with Microsoft.Extensions.AI, Microsoft Agent Framework, and Ollama

In the previous chapter, we deployed the eShop Aspire application to Azure Container Apps and verified that its microservices, backing services, and communication flows worked successfully in a cloud environment. With the core distributed application now in place, we are ready to extend it with **generative AI** capabilities powered by **large language models** (**LLMs**).

In this chapter, we will use **Microsoft.Extensions.AI** to integrate LLM-based chat and semantic search capabilities into the eShop microservices. `Microsoft.Extensions.AI` provides a unified abstraction layer for working with AI services through interfaces such as `IChatClient` and `IEmbeddingGenerator`, allowing the application to remain largely independent of a specific model provider. We will also use **Microsoft Agent Framework**, the successor to Semantic Kernel, together with Aspire integrations such as `CommunityToolkit.Aspire.OllamaSharp` to connect LLM capabilities to the existing microservice architecture.

For local development, we will integrate **Ollama** and run an LLM such as Llama 3.2 locally, while the same abstractions can support cloud-hosted models when required. This provider-agnostic approach allows us to change the underlying LLM provider without rewriting the application's business logic, while giving us greater flexibility over cost, privacy, and deployment.

Building on this foundation, we will develop a customer-support **Q&A** capability that allows users to ask product-related questions and receive LLM-generated responses. We will also introduce **embeddings** and **vector storage** to support semantic product search and related retrieval scenarios. Finally, we will integrate these AI capabilities into the Blazor frontend so that users can interact with them directly from the eShop application.

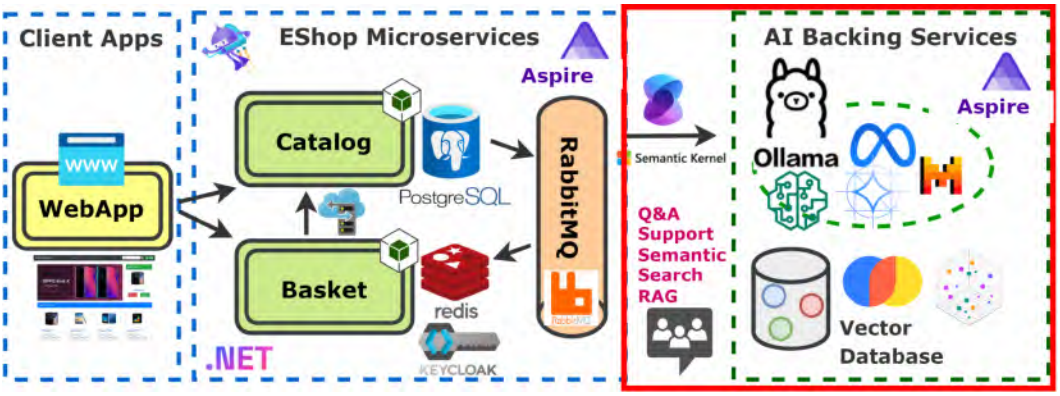


Figure 13.1 – Architecture diagram showing WebApp, Catalog, and Basket microservices with AI backing services (Ollama, Microsoft Agent Framework, vector database) for QnA, support, and semantic search scenarios

In this chapter, we'll be covering the following topics:

- Fundamentals and capabilities of large language models
- Running LLMs locally with Ollama
- Using AI services as cloud-native backing services
- Ollama integration in Aspire
- Downloading and managing LLMs in Ollama containers
- Consuming Ollama from the Catalog microservice
- Using Microsoft.Extensions.AI for provider-independent AI integration
- Registering the Ollama chat client
- Developing the AI-powered product support service
- Exposing and testing the AI support endpoint
- Integrating AI-powered support into the Blazor application
- Testing the Blazor AI support feature

Fundamentals and capabilities of large language models

LLMs represent some of the most advanced AI systems available today. They are trained on massive amounts of text data from books, websites, research papers, conversations, and so on, enabling them to understand and generate human-like text. They can also interact with users in natural language, making them central to applications such as AI chatbots, content creation tools, translation services, and other intelligent language systems.

At their core, LLMs are specialized machine learning models designed to work with human language at a very large scale. Some models contain billions or even trillions of **parameters**, which allow them to capture complex patterns in language and perform a wide range of tasks. These tasks include text generation, summarization, question answering, translation, and complex dialogue management.

So, why are these models considered "large"?

The term "large" in large language models refers to both the size of the model in terms of the number of parameters and the breadth of the training data. These models are trained using billions of parameters, which are internal settings adjusted during training to improve language understanding. With billions or trillions of parameters, LLMs have an extensive capacity to store knowledge and understand context. This enormous scale allows them to produce accurate, contextually aware responses, making them highly effective for language-based tasks.

For example, GPT-3, one of the most well-known LLMs, has 175 billion parameters. The larger the number of parameters, the more complex and nuanced the model's understanding of language becomes. However, there is a trade-off: larger models require more computational resources to train and run, making them more expensive to operate. This trade-off makes the choice of where and how to run an LLM especially important. For development and experimentation, one option is to run models locally, which brings us to Ollama.

Running LLMs locally with Ollama

Ollama enables you to run LLMs and **small language models** (SLMs) directly on your local machine without relying on cloud-hosted inference. By keeping model execution local, Ollama allows you to build AI applications without sending data to external cloud services. This can improve privacy, support offline use, reduce dependence on cloud infrastructure, and give you greater control over how your AI models are hosted and managed.

Ollama provides a wide selection of pre-built models for tasks such as conversational AI, code generation, creative work, and domain-specific applications. You can browse available models such as Llama 3.2, Llama 3.1, Gemma 2, Qwen, Phi, and Mistral, each offering different parameter sizes and capabilities.

For conversational applications, models such as Llama, Gemma, Qwen, Phi, and Mistral can support chat-based interactions. For code generation, models such as CodeGemma and Code Llama can assist with tasks such as generating code snippets and debugging. Ollama also supports models intended for creative, multimodal, and domain-specific use cases. Its model library therefore gives you the flexibility to select a model according to the requirements of your application.

Running models locally with Ollama can provide several benefits, including offline capability, greater control over data privacy, reduced reliance on cloud-based inference, and potentially lower cloud costs and latency. These characteristics make Ollama useful for scenarios such as software development, customer-support applications, educational tools, and creative workloads.

Using AI services as cloud-native backing services

Running models locally is useful during development, but in a distributed application, we also need to consider how AI capabilities fit into the broader architecture. In cloud-native systems, traditional *backing services* such as databases, message brokers, and distributed caches provide storage, communication, and scalability. AI-powered applications can extend this backing-service pattern by treating LLMs and **vector databases** as additional services that provide intelligence, semantic understanding, and retrieval capabilities.

By integrating LLMs and vector databases in this way, you can introduce capabilities such as summarization, classification, chat-based Q&A, **similarity search**, and knowledge retrieval into your microservices architecture. LLMs provide language-based understanding and generation, while vector databases support **semantic search** and retrieval based on similarity. Together, these services can support scenarios such as automated customer support, recommendations, semantic search, and **retrieval-augmented generation**.

Ollama integration in Aspire

With Ollama established as an AI backing service, the next step is to integrate it with the Aspire application so that its capabilities can be consumed by the existing microservices.

Ollama enables you to host LLMs such as Llama 3.2 locally within your Aspire microservices architecture. By integrating Ollama as a backing service, you can make AI capabilities such as

Q&A and embedding generation available to your microservices through Aspire-managed configuration, references, and environment variables.

To integrate Ollama hosting with your Aspire solution, begin by installing the required package into your AppHost project. The package is `CommunityToolkit.Aspire.Hosting.Ollama`, and it provides the necessary extension methods for Ollama **container orchestration**.

1. Open your solution and navigate to the AppHost project.
2. Right-click on **Dependencies** and select **Manage NuGet Packages**.
3. Search for `CommunityToolkit.Aspire.Hosting.Ollama` and install it into your AppHost project.

After installation, verify the package reference by opening your project file. You should see the following line included:

```
<PackageReference Include="CommunityToolkit.Aspire.Hosting.Ollama"
Version="13.4.0" />
```

This package extends Aspire with Ollama hosting support, allowing you to use the `builder.AddOllama` extension method to add an Ollama resource to your distributed application.

Before configuring the resource, choose the model you want to run. You can browse the available models on the Ollama website and review their sizes and capabilities. In this example, we will use Llama 3.2, which is a small model requiring only 2 GB of space.

Next, open the AppHost project's `Program.cs` file and add the Ollama configuration in the *backing-services* section, before the project definitions. The following code adds the Ollama resource and configures the Llama 3.2 model:

```
var ollama = builder
    .AddOllama("ollama", 11434)
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent)
    .WithOpenWebUI();
var llama = ollama.AddModel("llama3.2");
```

- The `AddOllama` method declares an Ollama resource named `"ollama"` on port `11434`, which is the default port for Ollama. Specifying the port number ensures that you know exactly where the Ollama service is listening. If you omit this, Aspire might assign a random port, which can complicate integration.

- The `WithDataVolume` extension method provides persistent storage for Ollama data, allowing you to store the Llama 3.2 model inside the Ollama container.
- The `WithLifetime(ContainerLifetime.Persistent)` option ensures the container's data persists across restarts.
- The `WithOpenWebUI` extension method adds an Open WebUI container alongside the Ollama resource, providing a browser-based interface for interacting with and managing the available models.
- The `AddModel("llama3.2")` method adds Llama 3.2 as a model resource and ensures that the model is downloaded and made available to the Ollama resource.

Once Ollama and the Llama model are configured, you can reference them from other microservices. For example, to connect the Catalog microservice to the Llama model, add a reference using the `WithReference` keyword and the `llama` variable:

```
var catalog = builder
    .AddProject<Projects.Catalog>("catalog")
    .WithReference(catalogDb)
    .WithReference(rabbitmq)
    .WithReference(llama)
    .WaitFor(catalogDb)
    .WaitFor(rabbitmq)
    .WaitFor(llama);
```

Adding the reference to the Llama model automatically injects environment variables, including the connection string for the Ollama Llama model, into the Catalog microservice. The `WaitFor(llama)` ensures that the Ollama container with the Llama model is fully up and running before the Catalog microservice starts.

This integration pattern allows your microservices to access LLMs as **backing services**, making AI-powered capabilities available throughout your distributed application. With the Ollama resource and Llama 3.2 model now defined in `AppHost`, the next step is to start the environment and allow Ollama to download the model locally.

Downloading and managing LLMs in Ollama containers

When you start the Aspire application for the first time, Ollama downloads the Llama 3.2 model into its container. You can monitor the download and model status through the **Aspire dashboard**. Once the download completes, the model becomes available for use by the microservices that reference it.

To start the download and setup process, Right-click the `AppHost` project in your solution and set it as the startup project. **Run** the application, making sure that Docker Desktop is running in the background.

Once the application starts, the Aspire dashboard displays the resources and projects defined in `AppHost`. The Ollama container starts first, followed by the download of the Llama 3.2 model.

You can monitor the Ollama container and model download status in the **Resources** view of the Aspire dashboard.

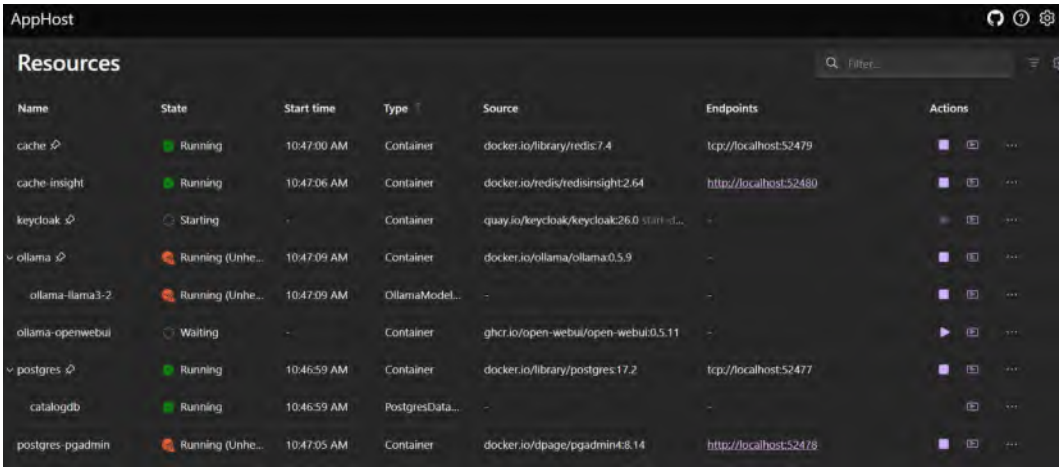


Figure 13.2 – Ollama container and model download status

To view the download progress in more detail, select the `ollama-llama3-2` resource. The resource details show the current download status and percentage.

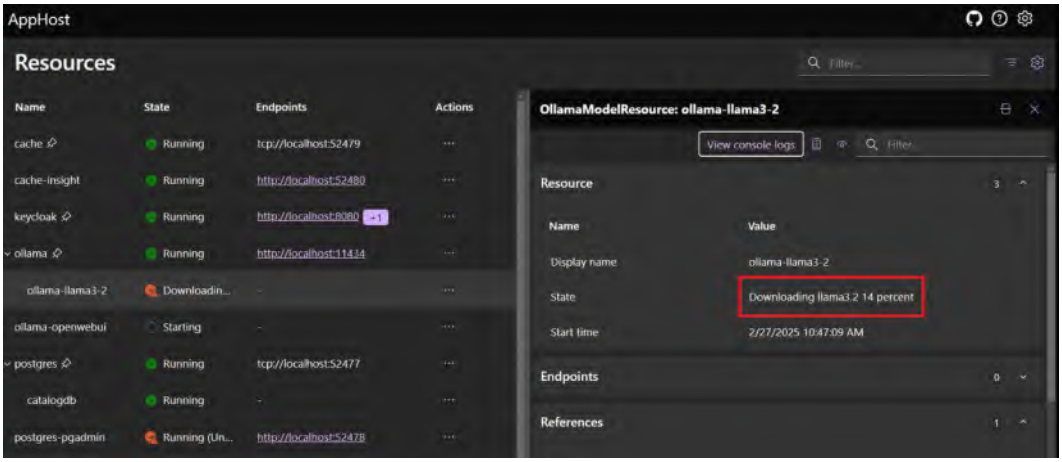


Figure 13.3 – ollama-llama3-2 download progress

As the download continues, the displayed percentage increases until the model has been downloaded completely.

The first download might take several minutes, depending on your network connection. Because the Ollama resource is configured with `WithDataVolume()`, the downloaded model data is persisted locally. On subsequent starts, the model can be reused from the **persistent volume** instead of being downloaded again.

After the download completes, you can connect to the Llama model from your Catalog microservice. To do this, check the environment variables for the Catalog microservice, where you will find the connection string for Ollama, including the model name `llama3.2`. This connection string is injected by Aspire when you added the reference for the Llama 3.2 model.

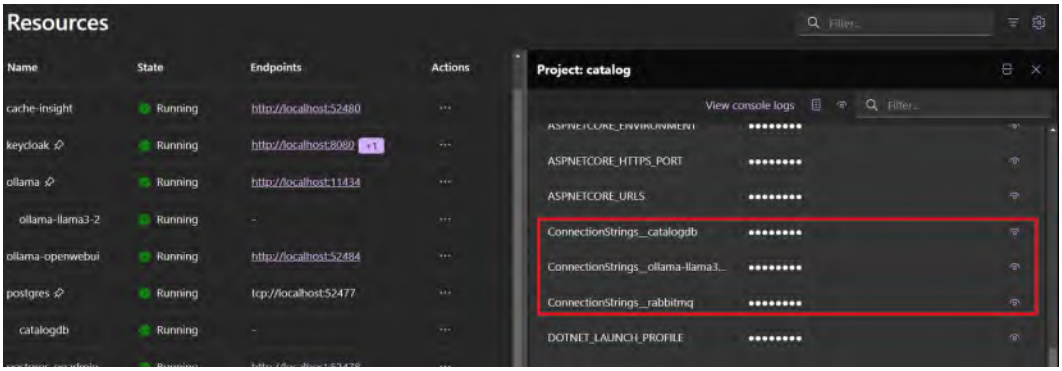


Figure 13.4 – Ollama connection string

The value of the connection string includes an **endpoint** definition, which specifies the Ollama host address (`localhost`) and the port number defined in the AppHost, as well as the model name (`llama3.2`). This endpoint allows your Catalog microservice to communicate with the Llama model to send prompts or queries and receive responses for AI-driven features.



Figure 13.5 – Value of the connection string

Because the Ollama resource also uses `WithOpenWebUI()`, the Aspire application includes a browser-based Open WebUI (`ollama-openwebui`) for interacting with the available models. You can open this interface from the corresponding endpoint in the Aspire dashboard. From the interface, select the latest model (**Llama 3.2**), and send a test prompt such as:

```
What are the microservices?
```

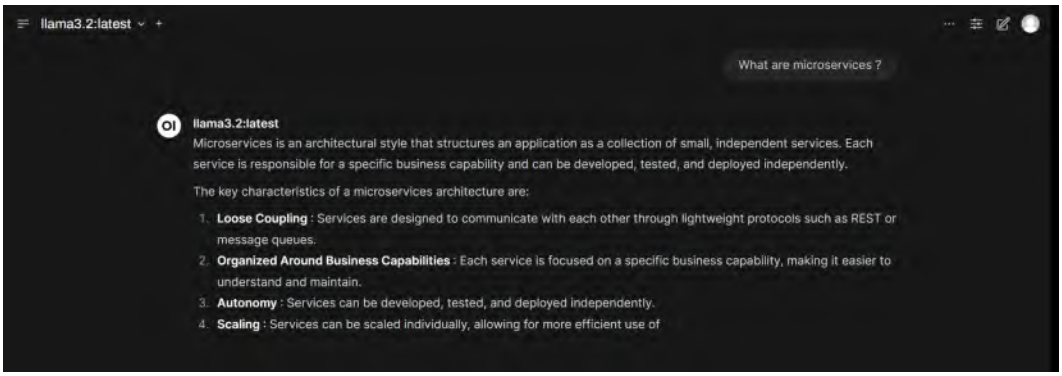


Figure 13. 6 – WebUI showing test prompt with response

Receiving a response confirms that the Llama 3.2 model has been downloaded successfully and is running through Ollama.

With the model running and available to the Catalog microservice, the next step is to add the client-side integration that allows Catalog to consume the Ollama service from application code.

Consuming Ollama from the Catalog microservice

To integrate the Ollama client into your Aspire solution, you will need to add the necessary package to your Catalog microservice so they can consume AI services for tasks like semantic search and chat-based interactions with the LLM.

Open Visual Studio and navigate to your Catalog microservice project, go to **Dependencies** and select **Manage NuGet Packages. Search** for `CommunityToolkit.Aspire.OllamaSharp` and install the package. You may choose to install a pre-release version if available, but here you proceed with the stable version.

After installation, the package simplifies communication with the Ollama service by using the connection information provided by Aspire for the configured Llama 3.2 model.

The following code shows the updated `.csproj` file for your Catalog microservice, including the new package reference:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
<PropertyGroup>
<TargetFramework>net10.0</TargetFramework>
<Nullable>enable</Nullable>
<ImplicitUsings>enable</ImplicitUsings>
</PropertyGroup>
<ItemGroup>
<PackageReference Include="Aspire.Npgsql.EntityFrameworkCore.PostgreSQL"
Version="13.5.4" />
<PackageReference Include="CommunityToolkit.Aspire.OllamaSharp" Version="13.4.0" /
>
<PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="10.0.12">
<PrivateAssets>all</PrivateAssets>
<IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</
IncludeAssets>
</PackageReference>
</ItemGroup>
<ItemGroup>
<ProjectReference Include="..\ServiceDefaults\ServiceDefaults.csproj" />
</ItemGroup>
</Project>
```

The OllamaSharp integration works with `Microsoft.Extensions.AI`, which provides common **abstractions** for interacting with AI services. Before registering the Ollama chat client, it is

useful to understand how these abstractions help separate your application logic from a specific AI provider.

Using Microsoft.Extensions.AI for provider-independent AI integration

To understand how you can integrate generative AI into your Aspire application, you need to become familiar with the `Microsoft.Extensions.AI` library. This library provides a unified set of abstractions that allow you to build AI-powered solutions without being tied to a specific service provider.

`Microsoft.Extensions.AI` acts as an *abstraction layer* between your Aspire application and different AI services. Instead of writing application code directly against a provider-specific **API**, you can work with common interfaces such as `IChatClient` and `IEmbeddingGenerator`.

The following diagram illustrates the structure of `Microsoft.Extensions.AI` and how it fits into your application architecture.

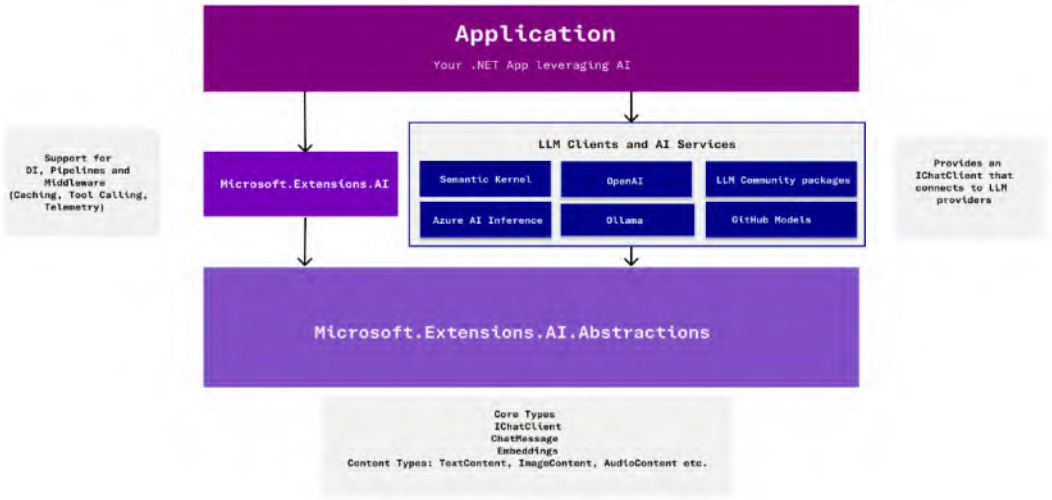


Figure 13.7 – Architecture of Microsoft.Extensions.AI, showing application code working through common AI abstractions with different underlying AI services and providers

By using these abstractions, you can implement features such as chat-based question answering, embedding generation for semantic search, and **tool calling** for AI-driven API integration in a consistent way across different LLM providers.

For example, the `IChatClient` interface allows you to select a chat client according to the environment. A local environment might use an Ollama client, while another environment

could use a cloud-hosted AI client, with the calling code continuing to work through the same interface:

```
IChatClient client =
environment.IsDevelopment
? new OllamaChatClient(...)
: new AzureAIInferenceChatClient(...);
var response = await client.GetResponseAsync(messages, options,
cancellationToken);
```

This approach allows you to keep your application flexible and future-proof, making it easy to change providers as your requirements evolve.

For this application, we will use the Ollama chat client during local development and register it with the Catalog microservice.

Registering the Ollama chat client

To complete the client integration for your Catalog microservice, you need to register the `OllamaSharp` chat client using `Microsoft.Extensions.AI`. This step connects your Catalog microservice to the LLM module, allowing you to use the `IChatClient` interface in your business classes for AI-driven features.

First, make sure that you have added the required NuGet package to the Catalog microservice project. The following entry in the `.csproj` file confirms that the `CommunityToolkit.Aspire.OllamaSharp` package is installed:

```
<PackageReference Include="CommunityToolkit.Aspire.OllamaSharp" Version="13.4.0" /
>
```

In the Catalog microservice, register the `OllamaSharp` chat client in `Program.cs` before building the application:

```
builder.AddOllamaSharpChatClient("ollama-llama3-2");
```

The connection name must exactly match the Ollama model resource name exposed by the AppHost. In this application, that name is `ollama-llama3-2`, which identifies both the Ollama resource and the `Llama 3.2` model. Using only `ollama` will not resolve the model-specific connection information and will result in an exception.

Your `Program.cs` file in the Catalog microservice should now include the chat client registration as shown here: :

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.AddServiceDefaults();
builder.AddNpgsqlDbContext<ProductDbContext>(connectionName: "catalogdb");
builder.Services.AddScoped<ProductService>();
builder.Services.AddMassTransitWithAssemblies(Assembly.GetExecutingAssembly());
builder.AddOllamaSharpChatClient("ollama-llama3-2");
var app = builder.Build();
// Configure the HTTP request pipeline.
app.MapDefaultEndpoints();
app.UseMigration();
app.MapProductEndpoints();
app.UseHttpsRedirection();
app.Run();
```

Once you have registered the OllamaSharp chat client, you can inject the `IChatClient` interface into your business service classes using dependency injection.

You are now ready to implement AI-driven business logic in your product AI service class using the registered chat client.

Developing the AI-powered product support service

To develop the business logic for AI-powered product support, create a `ProductAIService` class in the business layer. This class uses the `IChatClient` interface to provide chat-based customer support and Q&A capabilities while preserving the *layered architecture* that separates domain models, data access, business services, and presentation endpoints.

Within `ProductAIService`, inject `IChatClient`, send the user's query to the LLM, and return the generated response. Let's begin:

1. Create a new file named `ProductAIService.cs` under the `Services` folder in your Catalog microservice.
2. Define the `ProductAIService` class and inject `IChatClient` through its primary constructor:

```
using Microsoft.Extensions.AI;
namespace Catalog.Services;
```

```
public class ProductAIService(IChatClient chatClient)
{
}
```

3. Implement the `SupportAsync` method, which will handle incoming user queries and interact with the AI model:

```
public async Task<string> SupportAsync(string query)
{
    var systemPrompt = """
    You are a useful assistant.
    You always reply with a short and funny message.
    If you do not know an answer, you say 'I don't know that.'
    You only answer questions related to outdoor camping products.
    For any other type of questions, explain to the user that you only
answer outdoor camping products.
    At the end, Offer one of our products: Hiking Poles-$24, Outdoor Rain
Jacket-$12, Outdoor Camping Tent-$120.
    Do not store memory of the chat conversation.
    """;
    var chatHistory = new List<ChatMessage>
    {
        new ChatMessage(ChatRole.System, systemPrompt),
        new ChatMessage(ChatRole.User, query)
    };
    var resultPrompt = await chatClient.GetResponseAsync(chatHistory);
    return resultPrompt.Text;
}
```

The method begins by defining a *system prompt* that limits the assistant to questions about outdoor camping products and specifies that responses should be short and humorous. It then builds a chat history containing the system prompt and the user's query, and passes these messages to the injected `IChatClient` through `GetResponseAsync`. The generated response is returned as a string, ready to be exposed through an endpoint or minimal API.

The completed service therefore looks like this:

```
using Microsoft.Extensions.AI;
namespace Catalog.Services;
public class ProductAIService(IChatClient chatClient)
```

```
{
    public async Task<string> SupportAsync(string query)
    {
        var systemPrompt = """
        You are a useful assistant.
        You always reply with a short and funny message.
        If you do not know an answer, you say 'I don't know that.'
        You only answer questions related to outdoor camping products.
        For any other type of questions, explain to the user that you only answer
outdoor camping products.
        At the end, Offer one of our products: Hiking Poles-$24, Outdoor Rain
Jacket-$12, Outdoor Camping Tent-$120.
        Do not store memory of the chat conversation.
        """;
        var chatHistory = new List<ChatMessage>
        {
            new ChatMessage(ChatRole.System, systemPrompt),
            new ChatMessage(ChatRole.User, query)
        };
        var resultPrompt = await chatClient.GetResponseAsync(chatHistory);
        return resultPrompt.Text;
    }
}
```

This approach allows you to provide async AI-driven support logic in your microservice, using the abstractions provided by `Microsoft.Extensions.AI` and the Ollama-backed LLM.

Next, we have to register `ProductAIService` in the Catalog `Program.cs` file so that it can be resolved through dependency injection. Below the point where we registered `ProductService`, add the following:

```
builder.Services.AddScoped<ProductAIService>();
```

With the service registered, the next step is to expose its functionality through an HTTP endpoint.

Exposing and testing the AI support endpoint

To expose an **HTTP** `GET` endpoint for AI-driven support queries, you need to add a new route to your `ProductEndpoints` class that injects the product AI service and delegates the query to it.

At the end of `ProductEndpoints.cs`, after the existing product endpoints, add a route that accepts a user query and delegates it to `ProductAIService`:

```
// Support AI
group.MapGet("/support/{query}", async (string query, ProductAIService service) =>
{
    var response = await service.SupportAsync(query);
    return Results.Ok(response);
})
    .WithName("Support")
    .Produces(StatusCodes.Status200OK);
```

This code defines a `GET` endpoint at `/support/{query}`. The endpoint receives the user-provided query as a route parameter and injects an instance of `ProductAIService`. Inside the handler, it calls `service.SupportAsync(query)` to generate an AI-powered response and returns the result with a `200 OK` status.

For example, if the user submits a query such as `"give me 1 outdoor activity"`, the endpoint forwards the query to `ProductAIService`, which sends it to the configured LLM and returns the generated response to the client.

To test the new support AI endpoint, open the `Catalog.http` file and navigate to the section where you define HTTP requests for testing the Catalog endpoints. Add a new request for the support endpoint:

```
GET {{Catalog_HostAddress}}/support/give-me-1-outdoor-activity
Accept: application/json
```

Save the `Catalog.http` file after adding this request.

To run the test, follow these steps:

1. Right-click the `AppHost` project in **Solution Explorer** and select **Set as Startup Project**.
2. Run the project using **HTTPS** to start your Aspire environment.

3. Wait for all projects and backing services to start successfully.
4. In the `Catalog.http` file, send the `GET` request to the support endpoint.

The request is handled by the Catalog microservice, which passes the query to `ProductAIService`. The service then uses `IChatClient` to send the prompt to the Llama 3.2 model running through Ollama and returns the generated response. Pasted text

If the AI model takes longer than 20 seconds to respond, you might encounter a timeout exception. During local testing, you can remove or comment out the standard resilience configuration in your Service Defaults project to allow the request more time to complete.

You can also use the Aspire dashboard to verify that the integration is running correctly. In the **Resources** view, confirm that the Ollama resource, the Llama 3.2 model, and the Catalog microservice are all running.

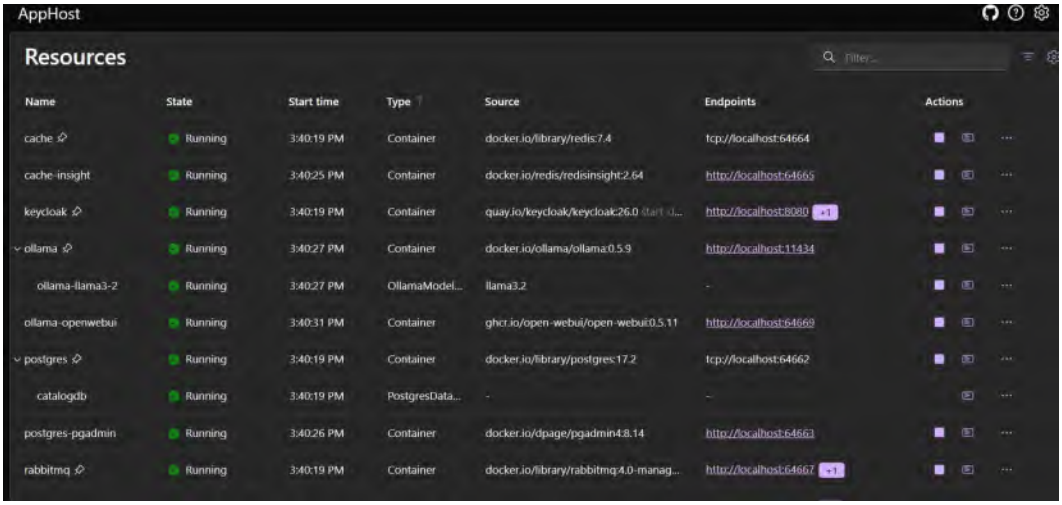


Figure 13.8 – Aspire dashboard Resources view showing Ollama, ollama-llama3-2, Catalog, and the other application resources in the running state

After sending the request from `Catalog.http`, the generated response appears in the HTTP client. For example, the model might return:

```
"How about 'Rock Climbing'? It's an outdoor activity that gets your adrenaline
pumping and offers a great workout."
```

You can further inspect the **Traces** view to examine the request flow from Catalog to Ollama and review the duration of the operation.

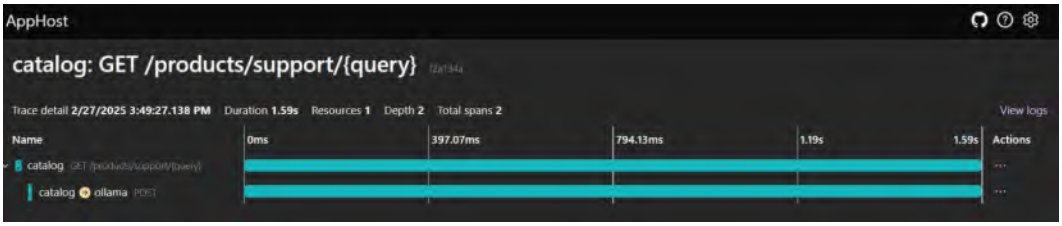


Figure 13.9 – Aspire dashboard Traces view showing a request to /products/support/{query} and the downstream call from Catalog to Ollama

This confirms that the Catalog microservice can send requests to the locally hosted Llama 3.2 model and receive AI-generated responses.

With the backend integration working, we can now expose the same capability through the Blazor frontend.

Integrating AI-powered support into the Blazor application

To make the AI support capability available to users, extend the Blazor WebApp with a dedicated support page. From this page, users can submit questions to the Catalog microservice, which forwards them to the Llama 3.2 model through `ProductAIService`.

This creates an end-to-end flow in which the user enters a question in the Blazor interface, the Catalog microservice processes it through the AI service, and the generated response is returned and displayed in the web app.

Begin by focusing on the `frontend` folder of your project and opening the web application. The first step is to update the API client to integrate with the Catalog microservice using an HTTP client. You will add a new method to the `CatalogApiClient` class to handle support queries. The existing client already retrieves products and product details:

```
using Catalog.Models;
namespace WebApp.ApiClients;
public class CatalogApiClient(HttpClient httpClient)
{
    public async Task<List<Product>> GetProducts()
    {
        var response = await httpClient.GetFromJsonAsync<List<Product>>($"/
products");
        return response!;
    }
```

```
    public async Task<Product> GetProductById(int id)
    {
        var response = await httpClient.GetFromJsonAsync<Product>($"/products/
{id}");
        return response!;
    }
}
```

Next, add a new method called `SupportProducts` to the `CatalogApiClient` class. This method takes a string query as a parameter, sends an HTTP `GET` request to the `/products/support/{query}` endpoint of the Catalog microservice, and returns the AI-generated string response.

The following code shows the addition of the `SupportProducts` method:

```
public async Task<string> SupportProducts(string query)
{
    var response = await httpClient.GetFromJsonAsync<string>($"/products/support/
{query}");
    return response!;
}
```

This method enables your Blazor web application to send user queries to the Catalog microservice, which in turn interacts with the Ollama platform and returns the AI-generated support response.

Next, create a new Blazor component named `Support.razor` under the `Components/Pages` folder. The page defines the `/support` route, injects `CatalogApiClient`, accepts a question from the user, and displays the response returned by the Catalog microservice:

```
@page "/support"
@attribute [StreamRendering(true)]
@rendermode InteractiveServer
@inject CatalogApiClient CatalogApiClient
<PageTitle>Support</PageTitle>
<h1>Support</h1>
<p>Ask questions about our amazing outdoor products that you can purchase.</p>
<div class="form-group">
    <label for="query" class="form-label">Type your question:</label>
    <div class="input-group mb-3">
        <input type="text"
               id="query"
```

```
                class="form-control"
                @bind="queryTerm"
                placeholder="Enter your question" />
        <button id="btnSend"
                class="btn btn-primary"
                @onclick="DoSend"
                type="submit">
            Send
        </button>
    </div>
    <hr />
</div>
@if (response != null)
{
    <p><em>@response</em></p>
}
@code {
    private string queryTerm = default!;
    private string response = default!;
    private async Task DoSend(MouseEventArgs e)
    {
        response = "Loading...";
        await Task.Delay(500);
        response =
            await CatalogApiClient.SupportProducts(queryTerm);
    }
}
```

You begin building the support page by setting its route to `/support`, allowing users to navigate directly to it. The page also uses the `@attribute [StreamRendering(true)]` directive to enable streaming rendering and injects `CatalogApiClient` so that the client can call the `SupportProducts` method you created earlier.

The Razor markup defines the page title and provides a short description inviting users to ask questions about outdoor products. Within the main `div`, a form group contains a label and an input field bound to the `queryTerm` variable. When the user hits the **send** button, the `DoSend` method is triggered. Once the AI-generated answer is returned, it is displayed using the `response` variable below the form.

The code section defines two variables: `queryTerm`, which stores the user's question, and `response`, which stores the answer returned by the LLM. When `DoSend` runs, it first sets response to `"Loading..."`, briefly waits, and then calls `CatalogApiClient.SupportProducts(queryTerm)`. This sends the user's question to the Catalog microservice, which forwards it to the LLM through the AI service. The returned answer is then assigned to `response` and displayed on the page.

To make the support page accessible from the navigation menu, you update the `NavMenu.razor` file. You add a new navigation item that links to the support page, allowing users to easily find and use the AI-powered support feature:

```
<div class="nav-item px-3">
<NavLink class="nav-link" href="support">
<span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span> Support
</NavLink>
</div>
```

With this addition, the support page appears in the navigation menu, enabling users to interact with the LLM through the Catalog microservice endpoints.

The **Support** option now appears in the application navigation. With the frontend wired to the Catalog endpoint, you can test the complete request path from the browser to the LLM.

Testing the Blazor AI support feature

To test the integration of your Blazor web application's support page with the Catalog microservice for chat AI operations, you need to ensure that all required services are running and accessible.

To test the support page, follow these steps:

1. First, open Visual Studio.
2. Set the AppHost project as the startup project.
3. Run the application and ensure that Docker Desktop is running.
4. Wait until Ollama, the Llama 3.2 model, catalog, WebApp, and the required backing services are running.

You can monitor the startup process from the Aspire dashboard.

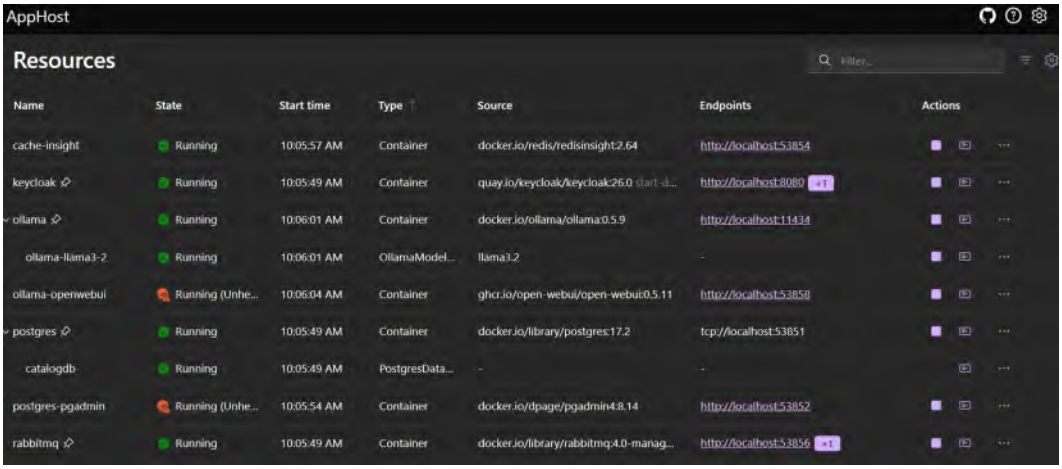


Figure 13.10 – Aspire dashboard showing Ollama, ollama-llama3-2, and other application resources starting

Once all services are running, including Ollama and the Catalog microservice, open the WebApp endpoint.

The Blazor application should now include a **Support** option in its sidebar.

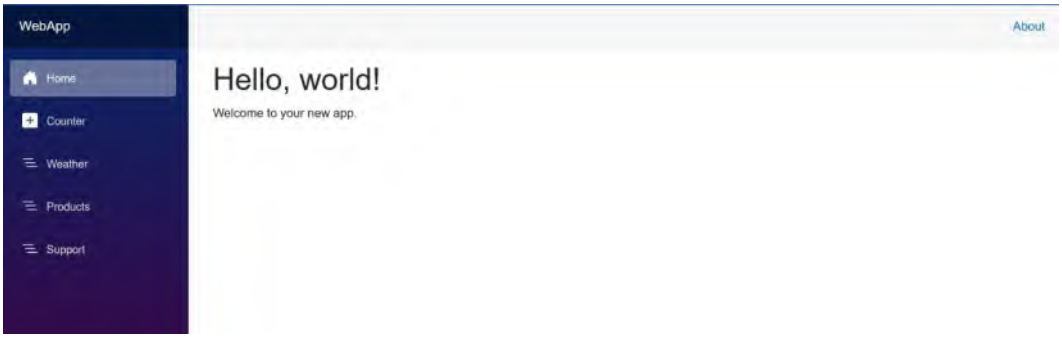


Figure 13.11 – Blazor web application home page with the Support menu in the left sidebar

Select the **Support** menu item to open the support page. This page allows users to ask questions about outdoor products and interact with the LLM in chat mode.

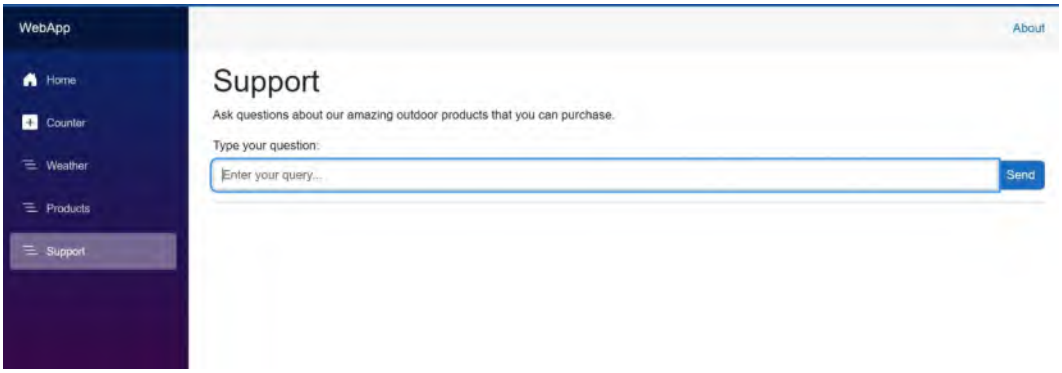


Figure 13.12 – Blazor web application support page with a text input for user queries and a "Send" button

To test the chat AI functionality, type a question in the input box, such as "`give me one outdoor activity.`" Then, select the **Send** button.

The first request might result in a timeout exception, but the system will automatically retry and eventually receive a response from the LLM. For example, you might see a response like "get ready to pitch yourself into the adventure," that the LLM generates as a recommendation for an outdoor activity.

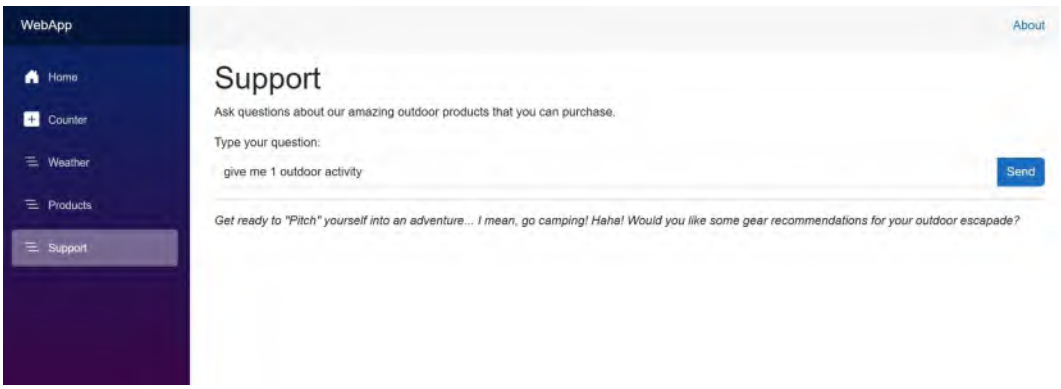


Figure 13.13 – LLM generating a response

You can also inspect the **Traces** view in the Aspire dashboard to follow this request across the application. There, you can confirm the interaction between the frontend, Catalog microservice, and Ollama-backed LLM. Pasted text

Summary

In this chapter, we established how to integrate generative AI capabilities into Aspire microservices using `Microsoft.Extensions.AI`, Ollama, and the `CommunityToolkit.Aspire.OllamaSharp` package. We began with the fundamentals of LLMs and explored how Ollama can be used to run models such as Llama 3.2 locally as AI backing services within the distributed application.

We then connected the Ollama-hosted model to the Catalog microservice and used the `IChatClient` abstraction from `Microsoft.Extensions.AI` to keep the application logic independent of a specific AI provider. Building on this integration, we created `ProductAIService`, defined a **system prompt** for product-related customer support, registered the service through **dependency injection**, and exposed an HTTP support endpoint that forwards user queries to the LLM and returns AI-generated responses.

Next, we integrated this chat support capability into the Blazor frontend by extending `CatalogApiClient`, creating a dedicated support page, and adding it to the application navigation. This gave users a complete end-to-end flow in which a question entered in the Blazor UI travels through the Catalog microservice to the Ollama-hosted Llama 3.2 model and the generated response is returned to the user. We also used the Aspire dashboard and traces to verify the interaction between the frontend, Catalog microservice, and Ollama service.

A key principle throughout the chapter was separating application logic from the underlying AI provider. By working through `Microsoft.Extensions.AI` abstractions such as `IChatClient`, the application can use a locally hosted Ollama model during development while retaining the flexibility to adopt other providers without rewriting the business logic.

With chat-based AI support now integrated into the eShop application, we have established the foundation for the next AI capability: semantic product search. In the next chapter, we will extend this architecture with vector embeddings and vector storage, use an **embedding model** to represent product information numerically, and compare traditional **keyword search** with semantic search that retrieves products based on the meaning of a user's query rather than exact text matches. We will then expose both search approaches through dedicated APIs and integrate them into the Blazor frontend so that users can switch between traditional and semantic search.

14

Building Semantic Product Search with Vector Embeddings

In the previous chapter, we extended the eShop application with generative AI by integrating Ollama and Llama 3.2 into the Catalog microservice through `Microsoft.Extensions.AI`. We used the `IChatClient` abstraction to build an AI-powered customer-support capability and exposed it through the Blazor frontend. With the application now able to generate natural-language responses, we can build on the same AI foundation to improve how users search for products.

In this chapter, we will move beyond traditional keyword matching and implement **semantic product search** using **vector embeddings** and **vector storage**. Instead of searching only for exact words in product names or descriptions, semantic search represents both product information and user queries as **numerical vectors**, allowing the application to retrieve products based on meaning and contextual similarity.

We will use an Ollama-hosted `all-minilm` embedding model to generate vector embeddings and integrate it with the Catalog microservice. We will also use `Microsoft.Extensions.VectorData` to work with vector records through a consistent abstraction and create a `ProductVector` model that stores product metadata together with its embedding.

For the hands-on implementation, we will use the `CommunityToolkit.VectorData.InMemory` provider so that we can focus on the semantic-search workflow without introducing another external service. Because this provider works through the `Microsoft.Extensions.VectorData` abstractions, the same application design can later be adapted to a persistent **vector database** when durable or larger-scale vector storage is required.

Building on this foundation, we will implement both traditional keyword search and semantic search, expose them through separate Catalog endpoints, and integrate them into a Blazor search page that allows users to switch between the two approaches. By the end of the chapter, the eShop application will be able to retrieve products not only by the words users type, but also by the meaning behind their queries.

In this chapter, we'll be covering the following topics:

- Understanding semantic product search, **vector databases**, and embeddings
- Configuring the embedding model with Ollama
- Adding vector-search integrations to the Catalog microservice
- Creating the `ProductVector` model
- Implementing traditional and semantic product search
- Exposing and testing the search endpoints
- Integrating and testing search in the Blazor frontend

Understanding semantic product search, vector databases, and embeddings

After adding AI-powered customer support in the previous chapter, the next step is to improve product discovery with semantic search. We will begin by establishing the concepts behind vector databases and vector embeddings before configuring the implementation with `Microsoft.Extensions.AI` and `Microsoft.Extensions.VectorData`.

At the heart of this approach are vector embeddings, which allow the application to compare products and queries based on meaning rather than exact wording. Semantic search uses these embeddings to determine how closely a user's query relates to the available product information.

In this chapter, Ollama hosts the `all-minilm` embedding model, `Microsoft.Extensions.AI` generates the embeddings, and `Microsoft.Extensions.VectorData` provides the abstractions used to store and search product vectors.

Before looking more closely at how these embeddings are generated, it is useful to understand where they are stored and how they can be searched efficiently. This brings us to the role of vector databases. A vector database is a specialized database designed to store, manage, and query high-dimensional vectors. These vectors are numerical representations that capture the semantic meaning of data, such as text, images, or audio. A vector database indexes and stores vector embeddings for fast retrieval and **similarity search**, enabling you to search for information based on concepts or context rather than exact keywords. Traditional databases

cannot capture these nuances in the same way, while vector databases are designed specifically for similarity-based retrieval.

The following figure shows where vector storage and the AI backing services fit within the eShop microservices solution, supporting customer support, semantic search, and related retrieval scenarios.

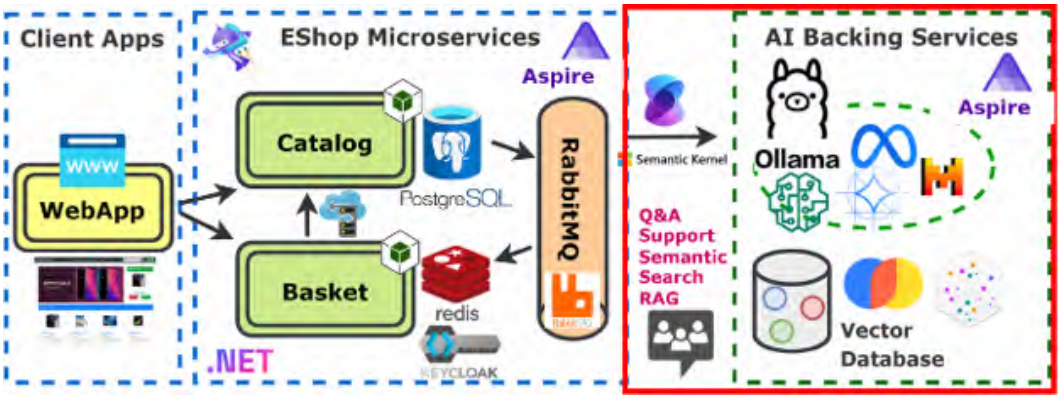


Figure 14.1 – Architecture diagram showing WebApp, Catalog, and Basket microservices connected to AI backing services (Microsoft Agent Framework, Ollama, and vector storage) for Q&A, support, semantic search, and related retrieval scenarios

With the role of vector storage in the wider architecture established, we can now look more closely at the concepts that make semantic search possible. We will first compare traditional and vector databases to understand why similarity-based retrieval requires a different approach to storing and searching data. We will then examine vectors and vector embeddings, including how they represent meaning and enable semantic comparisons between product information and user queries.

Comparing traditional and vector databases

To understand how data storage and search have evolved, compare traditional databases with vector databases. Traditional databases store data in structured formats such as rows and columns. While they are effective for specific tasks, traditional databases fall short when you need to perform **semantic searches** and search based on the meaning of data rather than exact words.

In a traditional database, searching for "`laptop`" might retrieve only records containing the exact word "`laptop`." But a vector database could understand that "`notebook computer`" has the same meaning and include those results too. Instead of storing structured raw data such as strings or integers, vector databases store **unstructured data** such as documents, images, audio, and video by converting it into numerical **embeddings**. These are dense, high-

dimensional vectors created using AI models, allowing the database to understand the relationship between pieces of data.

Understanding vectors and vector embeddings

Vectors are a foundational concept in AI, allowing you to represent complex data such as text, images, or audio as compact numerical summaries that models can process.

A **vector** is a mathematical object that has both magnitude and direction, and it is typically represented as a list of numbers, for example: `[1.2, 3.4,-0.8]`. In two-dimensional space, you can visualize a vector as an arrow pointing in a certain direction, but in AI, vectors often exist in high-dimensional space, which is harder to visualize but useful for computations.

Instead of describing an object like a fruit with words such as "round, orange, and sweet," you encode these features numerically in a vector, for example: `[roundness: 0.9, color: 0.8, sweetness: 0.7]`.

So, how do vectors relate to vector embeddings? A vector is a numerical representation made up of a sequence of values. A vector embedding is a vector generated by an embedding model to represent the characteristics or semantic meaning of data, such as text, images, or audio. In other words, all vector embeddings are vectors, but in AI, an *embedding* specifically refers to a vector whose values encode meaningful relationships in the original data.

Because embeddings capture these relationships numerically, semantically similar data tends to produce vectors that are positioned close together in the embedding space. This proximity allows applications to compare meaning rather than rely only on exact words or values.

To understand how vector embeddings are created, consider the following three-step process:

1. Start with raw data, such as a sentence, an image, or a sound file.
2. Pass the data through an AI embeddings model, such as a Transformer model.
3. The embedding model transforms the input into a vector embedding – a list of numbers representing the data's features and meanings.

For example, you can use the **OpenAI Tokenizer** tool (`https://platform.openai.com/tokenizer`) to see how a phrase such as "*artificial intelligence*" is divided into tokens and to view the total number of tokens it contains. The tool also shows the token IDs assigned to each token by the selected encoding.

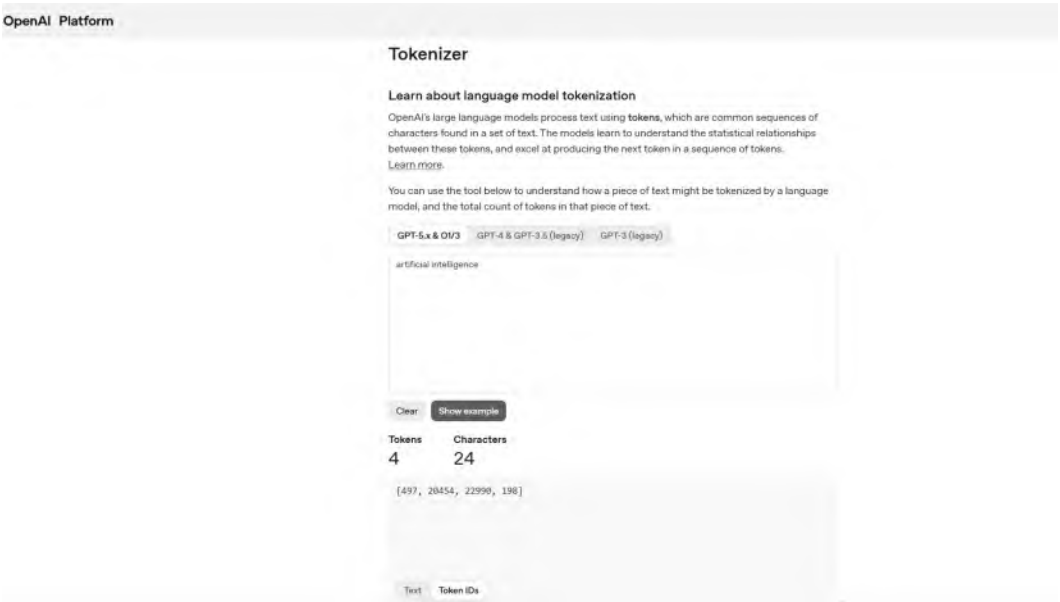


Figure 14.2 – OpenAI Tokenizer tool

If you replace "*intelligence*" with another word, such as "*knowledge*," the tokenization changes for the modified part of the phrase, while tokens corresponding to unchanged text may remain the same. This illustrates how a model breaks text into units for processing. The token IDs themselves do not represent semantic similarity; they are identifiers for tokens.

Therefore, if two sentences have similar meaning, their embeddings will be close together in high-dimensional space.

Embeddings are important because they allow you to compare and analyze data based on semantic meaning, not just surface-level features. By encoding complex relationships between data points in hundreds or thousands of dimensions, embeddings enable AI to move beyond exact matches and understand synonymous relationships and contextual meanings. This means you can ***find similar vectors***: if two sentences have similar meaning, their embeddings will be close together in high-dimensional space.

As *Figure 14.3* illustrates, embeddings for semantically similar words such as "cat" and "kitten" appear closer together, while less closely related terms appear farther apart.

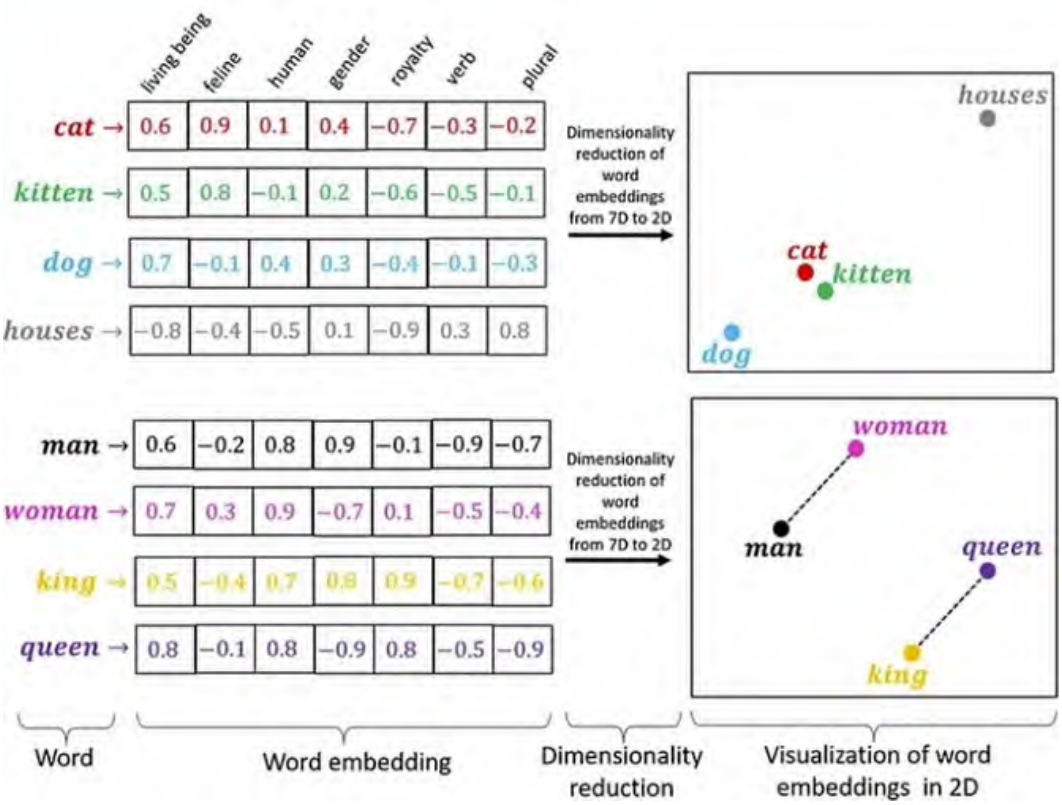


Figure 14.3 – Visualization of word embeddings showing similar words (cat, kitten) close together and others (dog, houses, man, woman, king, queen) at varying distances (Rozado D (2020) Wide range screening of algorithmic bias in word embedding models using large sentiment lexicons reveals underreported bias types. PLoS ONE 15(4): e0231189. `https://doi.org/10.1371/journal.pone.0231189`*)*

We now have the conceptual foundation for semantic search: vector embeddings provide the numerical representations used to capture meaning, while vector storage enables those representations to be searched by similarity. The next step is to put these concepts into practice by configuring the embedding model that will generate vectors for both product information and user queries.

Configuring the embedding model with Ollama

Now that you understand how embeddings support semantic similarity, the next step is to configure the embedding model that will be used by the eShop application. We will use the `all-minilm` model through Ollama to generate vector representations of product data and user queries.

You can browse the available embedding models from the **Models** section on the Ollama website (`https://ollama.com/`). In the **Models** section, select **Embedding** to get the list of embedding models.

Select the `all-minilm` model; you will see its details, including the command to pull the model:

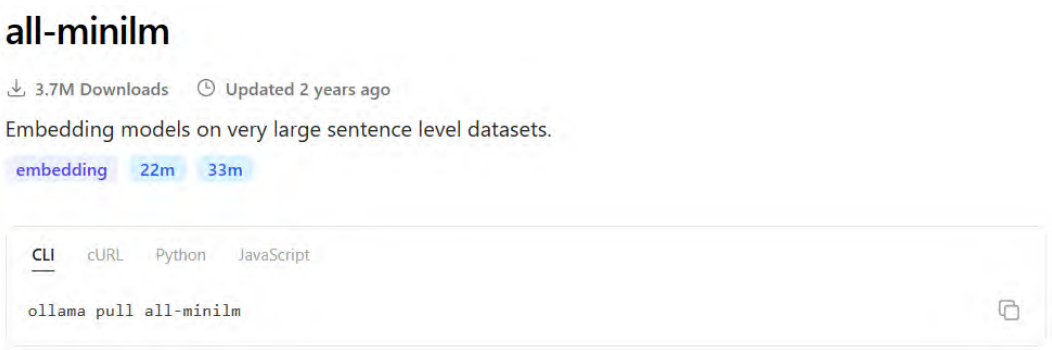


Figure 14.4 – The all-minilm model details page on the Ollama website, showing model information and the ollama pull all-minilm command

Copy the model name `all-minilm` for use in your application configuration.

Next, verify that the Aspire AppHost project already includes the Ollama hosting integration. Open the project file and confirm that the `CommunityToolkit.Aspire.Hosting.Ollama` package is referenced:

```
<PackageReference Include="CommunityToolkit.Aspire.Hosting.Ollama"
Version="13.4.0" />
```

Before configuring the embedding model, make sure the AppHost project is set as the startup project. This ensures that running the solution starts the Aspire application model and all of the resources defined in AppHost.

To add the `all-minilm` model to Ollama, open the AppHost `Program.cs` file and locate the existing Ollama resource, which already hosts the `Llama 3.2` chat model:

```
var ollama = builder
    .AddOllama("ollama", 11434)
    .WithDataVolume()
    .WithLifetime(ContainerLifetime.Persistent)
```

```
    .WithOpenWebUI();
var llama = ollama.AddModel("llama3.2");
```

Add a new line to register the `all-minilm` embedding model:

```
var embedding = ollama.AddModel("all-minilm");
```

This adds the `all-minilm` embedding model to the same Ollama resource so that Aspire can download and manage it alongside the `Llama 3.2` chat model.

Next, update your Catalog microservice configuration to reference the embedding model. In the Catalog project registration, add `WithReference` and `WaitFor` for the embedding model:

```
var catalog = builder
    .AddProject<Projects.Catalog>("catalog")
    .WithReference(catalogDb)
    .WithReference(rabbitmq)
    .WithReference(llama)
    .WithReference(embedding)
    .WaitFor(catalogDb)
    .WaitFor(rabbitmq)
    .WaitFor(llama)
    .WaitFor(embedding);
```

This injects the environment variable for the connection string to the `all-minilm` model, making it available to the Catalog microservice.

The connection string value for the `all-minilm` embedding model is exposed in the following form:

```
ConnectionStrings__ollama-all-minilm="Endpoint=http://localhost:11434;Model=all-
minilm"
```

When you run the Aspire application, the Ollama resource automatically downloads the `all-minilm` model. You can monitor the download progress in the **Aspire dashboard**. Because the model is relatively small, the download is typically quick.

With the embedding model available through Ollama, the next step is to connect it to the Catalog microservice and provide the vector-storage capability required for semantic search.

Adding vector-search integrations to the Catalog microservice

To support semantic search, the Catalog microservice uses `Microsoft.Extensions.VectorData.Abstractions` together with `CommunityToolkit.VectorData.InMemory`.

Earlier versions of this implementation used Semantic Kernel for the in-memory vector-store connector. In the current version, semantic search is implemented directly with `Microsoft.Extensions.AI` for embedding generation and `Microsoft.Extensions.VectorData` for vector storage and similarity search, so an agent framework is not required.

`Microsoft.Extensions.VectorData` provides a common abstraction for working with vector data without coupling the application directly to a specific vector database or storage provider. In this respect, it plays a role similar to Entity Framework Core for relational data: your application can create, read, update, and search vector records through a consistent API while the underlying storage implementation can vary.

Vector-store providers implement these abstractions for particular storage systems. In this chapter, we will use `CommunityToolkit.VectorData.InMemory`, which provides an in-memory vector store and keeps the focus on the semantic-search workflow without introducing another external service. Because the application works through `VectorStore` and `VectorStoreCollection<TKey, TRecord>`, the same overall design can later be adapted to another vector-store provider when persistent storage is required.

Before adding this capability to the Catalog microservice, first verify that the `CommunityToolkit.Aspire.OllamaSharp` client package used for the existing Q&A functionality is already installed. Then, add and `CommunityToolkit.VectorData.InMemory`.

The Catalog project already targets .NET 10 and includes the current Aspire, Ollama, and Entity Framework Core integrations. Before adding the vector-search packages, its relevant package references look as follows:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
<PropertyGroup>
<TargetFramework>net10.0</TargetFramework>
<Nullable>enable</Nullable>
<ImplicitUsings>enable</ImplicitUsings>
</PropertyGroup>
```

```
<ItemGroup>
<PackageReference Include="Aspire.Npgsql.EntityFrameworkCore.PostgreSQL"
Version="13.5.4" />
<PackageReference Include="CommunityToolkit.Aspire.OllamaSharp" Version="13.4.0" /
>
<PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="10.0.12">
<PrivateAssets>all</PrivateAssets>
<IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</
IncludeAssets>
</PackageReference>
</ItemGroup>
<ItemGroup>
<ProjectReference Include="..\ServiceDefaults\ServiceDefaults.csproj" />
</ItemGroup>
</Project>
```

Install the `Microsoft.Extensions.VectorData.Abstractions` and `CommunityToolkit.VectorData.InMemory` packages (by right-clicking **Dependencies**, selecting **Manage NuGet Packages**, and searching for the relevant packages). After installation, your project file should include references to `Microsoft.Extensions.VectorData.Abstractions` and `CommunityToolkit.VectorData.InMemory`:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
<PropertyGroup>
<TargetFramework>net10.0</TargetFramework>
<Nullable>enable</Nullable>
<ImplicitUsings>enable</ImplicitUsings>
</PropertyGroup>
<ItemGroup>
<PackageReference Include="Aspire.Npgsql.EntityFrameworkCore.PostgreSQL"
Version="13.5.4" />
<PackageReference Include="CommunityToolkit.Aspire.OllamaSharp" Version="13.4.0" /
>
<PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="10.0.12">
<PrivateAssets>all</PrivateAssets>
<IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</
IncludeAssets>
</PackageReference>
<PackageReference Include="Microsoft.Extensions.VectorData.Abstractions"
```

```
Version="10.10.0" />
<PackageReference Include="CommunityToolkit.VectorData.InMemory" Version="1.0.1" /
>
</ItemGroup>
<ItemGroup>
<ProjectReference Include="..\ServiceDefaults\ServiceDefaults.csproj" />
</ItemGroup>
</Project>
```

These additions provide the vector-data abstractions and the in-memory vector-store implementation used throughout the chapter. With them in place, the Catalog microservice can generate embeddings through Ollama, store product vectors in memory, and perform similarity searches against user queries.

The next step is to define how each product and its embedding will be represented in the vector store. For this, you will create a `ProductVector` class in the Catalog microservice.

Creating the ProductVector model

With the embedding and vector-store integrations in place, we now need a record type that represents each product in the vector store. For this, we will create a `ProductVector` class in the Catalog microservice to hold the product metadata together with its embedding.

This also marks the first step in extending the existing layered architecture for semantic search. We will proceed through the application in the following order:

1. **Models**: Create the `ProductVector` class to represent product metadata and vector embeddings.
2. **Services**: Implement traditional keyword search and semantic search.
3. **Endpoints**: Expose both search approaches through minimal **API** routes in the Catalog microservice.
4. **Frontend**: Add a Blazor search page with a checkbox that users can select to switch between traditional and semantic search.

We will begin with the **Models** layer. In Visual Studio, navigate to the Catalog microservice, expand the `Models` folder, and add a new class file named `ProductVector.cs`. Keeping the vector representation separate from the existing `Product` entity allows the semantic-search functionality to evolve without changing the core domain model.

Begin by defining the class:

```
namespace Catalog.Models;
public class ProductVector
{
}
```

Next, add the necessary `using` directives at the top of the file for vector data annotations:

```
using Microsoft.Extensions.VectorData;
using System.ComponentModel.DataAnnotations.Schema;
```

Now, implement the attributes for the `ProductVector` class. The `Id` property is marked with `[VectorStoreKey]`, indicating it as the primary key for vector records. The `Name`, `Description`, `Price`, and `ImageUrl` properties are each decorated with `[VectorStoreData]`, indicating that they should be stored as record data rather than used as vector fields.

The `Vector` property is where the actual embedding array is stored, using a `ReadOnlyMemory<float>` type. This property is annotated with `[VectorStoreVector(384, DistanceFunction = DistanceFunction.CosineSimilarity)]`, specifying a 384-dimensional embedding and cosine similarity as the distance function for vector operations. The `[NotMapped]` attribute ensures this property is not mapped to a traditional database column.

The complete `ProductVector` class is as follows:

```
using Microsoft.Extensions.VectorData;
using System.ComponentModel.DataAnnotations.Schema;
namespace Catalog.Models;
public class ProductVector
{
    [VectorStoreKey]
    public int Id { get; set; }
    [VectorStoreData]
    public string Name { get; set; } = default!;
    [VectorStoreData]
    public string Description { get; set; } = default!;
    [VectorStoreData]
    public decimal Price { get; set; }
    [VectorStoreData]
    public string ImageUrl { get; set; } = default!;
    [NotMapped]
```

```
    [VectorStoreVector(
        384,
        DistanceFunction = DistanceFunction.CosineSimilarity)]
    public ReadOnlyMemory<float> Vector { get; set; }
}
```

Keeping vector-specific data in `ProductVector` gives the semantic-search implementation a dedicated record shape while leaving the core Product entity unchanged. The `Microsoft.Extensions.VectorData` attributes define how each property participates in vector storage by identifying the record key, stored metadata, and embedding vector. With the vector model defined, we can now move to the Services layer and implement the two search approaches used by the application.

Implementing traditional and semantic product search

We will begin with a conventional keyword search to establish a baseline, then prepare the dependencies required for semantic search, and finally implement the semantic-search workflow in `ProductAIService`.

The two approaches solve the same user-facing problem in different ways. Traditional search matches the text supplied by the user against product data, while *semantic search* compares vector representations to retrieve products based on meaning. Implementing them side by side makes it easier to see how the behavior of the application changes when vector search is introduced.

Adding traditional keyword search

Before we implement the semantic search, we'll add a traditional keyword search to establish a clear baseline. *Traditional search* matches the user's query against the text stored in the application, typically using exact or partial keyword matches rather than comparing semantic meaning.

In this application, we'll implement the traditional search in the existing `ProductService` class. The search will use **Entity Framework Core** to filter product names based on the supplied query. This approach works well when the user enters an exact product name or a recognizable part of the name. It is less effective when the wording differs from the text stored in the product data.

To implement traditional search in the `ProductService` class, open Visual Studio and navigate to the Catalog microservice. Expand the **Services** folder and locate the `ProductService` class. At the end of this class, create a new method for searching products by keyword.

The following code defines the `SearchProductsAsync` method, which filters products by name using the `Contains` operation:

```
public async Task<IEnumerable<Product>> SearchProductsAsync(string query)
{
    return await dbContext.Products
        .Where(p => p.Name.Contains(query))
        .ToListAsync();
}
```

This method uses `dbContext` to access the `Products` entity and applies the query to the `Name` property. It returns the products whose names contain the supplied search term.

For a production implementation, you could extend the search to include product descriptions or introduce indexes for larger datasets. Here, however, the focus remains on a simple name-based search so that it can serve as a clear baseline for comparison with semantic search.

Implementing the semantic search

With the keyword-search baseline in place, the next step is to implement semantic search in `ProductAIService`. Unlike traditional keyword search, semantic search requires an **embedding generator** to convert product information and user queries into vectors, along with a **vector store** for storing and comparing those representations.

Before implementing `SearchProductsAsync`, we need to make sure the existing AI functionality uses the current `Microsoft.Extensions.AI` APIs and then register the additional dependencies required for semantic search.

If you are updating an earlier version of the project, first verify that the existing `SupportAsync` method uses `GetResponseAsync`:

```
var resultPrompt = await chatClient.GetResponseAsync(chatHistory);
return resultPrompt.Text;
```

This keeps the customer-support functionality compatible with the same `Microsoft.Extensions.AI version` used by the semantic-search implementation. With the existing **AI** integration aligned, we can now register the additional services that semantic search depends on: the embedding generator and the vector store.

Registering the embedding generator and vector store

The `AppHost` already defines and references the `Llama 3.2` chat model and the `all-minilm` embedding model, allowing Aspire to make their connection information available to the Catalog microservice at runtime.

The relevant AppHost references in `Program.cs` are

```
    .WithReference(llama)
    .WithReference(embedding)
```

These references make the corresponding model connections available to `Catalog`:

- `ollama-llama3-2` for chat
- `ollama-all-minilm` for embedding generation

Because Aspire supplies this connection information at runtime, the `Catalog` microservice does not need to hardcode the Ollama host addresses.

With the model resources available, configure `Program.cs` of the `Catalog` microservice to register the Ollama chat client, embedding generator, and in-memory vector store:

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.AddServiceDefaults();
builder.AddNpgsqlDbContext<ProductDbContext>(
    connectionName: "catalogdb");
builder.Services.AddScoped<ProductService>();
builder.Services.AddScoped<ProductAIService>();
builder.Services.AddMassTransitWithAssemblies(
    Assembly.GetExecutingAssembly());
// Register Ollama-based chat and embedding services
builder.AddOllamaSharpChatClient("ollama-llama3-2");
builder.AddOllamaSharpEmbeddingGenerator("ollama-all-minilm");
// Register the in-memory vector store
builder.Services.AddInMemoryVectorStore();
var app = builder.Build();
// Configure the HTTP request pipeline.
app.MapDefaultEndpoints();
app.UseMigration();
app.MapProductEndpoints();
```

```
app.UseHttpsRedirection();
app.Run();
```

This configuration registers the components required by the AI functionality:

- `AddOllamaSharpChatClient` registers the chat client used by the existing **Q&A** functionality.
- `AddOllamaSharpEmbeddingGenerator` registers the `all-minilm` model for embedding generation.
- `AddInMemoryVectorStore` registers the in-memory vector-store provider used for semantic search.

With these services registered, `ProductAIService` can receive the embedding generator and vector store through dependency injection.

Implementing semantic search in ProductAIService

With the required dependencies registered, we can now implement the semantic-search workflow itself. `ProductAIService` uses `ProductDbContext` to retrieve product data, `IEmbeddingGenerator<string, Embedding<float>>` to generate vectors, and `VectorStore` to access the in-memory collection of product vectors.

Define `ProductAIService` with these dependencies and obtain a typed `VectorStoreCollection<int, ProductVector>` named `products` from the registered `VectorStore`:

```
public class ProductAIService(
    ProductDbContext dbContext,
    IChatClient chatClient,
    IEmbeddingGenerator<string,
    Embedding<float>> embeddingGenerator,
    VectorStore vectorStore)
{
    private readonly VectorStoreCollection<int, ProductVector>
productVectorCollection =
        vectorStore.GetCollection<int, ProductVector>("products");
}
```

Next, implement `SearchProductsAsync`. The method first checks whether the product-vector collection exists. If it does not, it calls `InitEmbeddingsAsync` to generate embeddings for the products and populate the vector store.

```
public async Task<IEnumerable<Product>> SearchProductsAsync(string query)
{
    if (!await productVectorCollection.CollectionExistsAsync())
    {
        await InitEmbeddingsAsync();
    }
    var queryEmbedding = await embeddingGenerator.GenerateVectorAsync(query);
    var results = productVectorCollection.SearchAsync(queryEmbedding, top: 1);
    List<Product> products = [];
    await foreach (var resultItem in results)
    {
        products.Add(new Product
        {
            Id = resultItem.Record.Id,
            Name = resultItem.Record.Name,
            Description = resultItem.Record.Description,
            Price = resultItem.Record.Price,
            ImageUrl = resultItem.Record.ImageUrl
        });
    }
    return products;
}
```

The `InitEmbeddingsAsync` helper method initializes the vector collection, retrieves the products from PostgreSQL, generates a descriptive text representation for each product, converts that text into an embedding, and stores the resulting `ProductVector` record:

```
private async Task InitEmbeddingsAsync()
{
    await productVectorCollection.EnsureCollectionExistsAsync();
    var products = await dbContext.Products.ToListAsync();
    foreach (var product in products)
    {
        var productInfo = $"{product.Name}] is a product that costs
[{product.Price}] and description is [{product.Description}]";
        var productVector = new ProductVector
```

```
        {
            Id = product.Id,
            Name = product.Name,
            Description = product.Description,
            Price = product.Price,
            ImageUrl = product.ImageUrl,
            Vector = await embeddingGenerator.GenerateVectorAsync(productInfo)
        };
        await productVectorCollection.UpsertAsync(productVector);
    }
}
```

This approach ensures that on the first semantic-search request, the vector store is populated if the product embeddings have not yet been generated. Subsequent searches only need to generate an embedding for the user's query, compare it with the stored product vectors, and return the closest match.

Generating all product embeddings during the first search keeps the example simple, but it is not ideal for a production environment. In a larger application, embeddings would typically be generated ahead of time or updated in the background when product data changes, avoiding the latency of initializing the vector store during a user request.

With the semantic-search workflow complete, both search approaches are now available in the Services layer: `ProductService` handles traditional keyword matching, while `ProductAIService` handles vector-based semantic retrieval. The next step is to expose both through the `Catalog` API.

Exposing and testing the search endpoints

In this section, we will add separate endpoints for traditional search and semantic search, keeping the two approaches clearly distinguishable at the API level. We will then test both endpoints from `Catalog.http` and use the Aspire dashboard to observe how the semantic-search request flows through `Catalog` to the Ollama embedding model.

Adding the Catalog search endpoints

To expose the two search strategies, open the `ProductEndpoints` class in the `Endpoints` folder. Add one **HTTP** `GET` endpoint for traditional keyword search and another for semantic search.

The following code shows the relevant endpoint mappings within `MapProductEndpoints`:

```
public static class ProductEndpoints
{
    public static void MapProductEndpoints(this IEndpointRouteBuilder app)
    {
        var group = app.MapGroup("/products");
        // GET all
        group.MapGet("/", async (ProductService service) =>
        {
            var products = await service.GetProductsAsync();
            return Results.Ok(products);
        })
            .WithName("GetAllProducts")
            .Produces<List<Product>>(StatusCodes.Status200OK);
        // GET by ID
        group.MapGet("/{id}", async (int id, ProductService service) =>
        {
            var product = await service.GetProductByIdAsync(id);
            if (product is null) return Results.NotFound();
            return Results.Ok(product);
        })
            .WithName("GetProductById")
            .Produces<Product>(StatusCodes.Status200OK)
            .Produces(StatusCodes.Status404NotFound)
            .Produces(StatusCodes.Status204NoContent);
        // Support AI
        group.MapGet("/support/{query}", async (string query, ProductAIService
service) =>
        {
            var response = await service.SupportAsync(query);
            return Results.Ok(response);
        })
            .WithName("Support")
            .Produces(StatusCodes.Status200OK);
        // Traditional Search
        group.MapGet("search/{query}", async (string query, ProductService
service) =>
        {
            var products = await service.SearchProductsAsync(query);
            return Results.Ok(products);
```

```
        })
            .WithName("SearchProducts")
            .Produces<List<Product>>(StatusCodes.Status200OK);
        // AI Semantic Search
        group.MapGet("aisearch/{query}", async (string query, ProductAIService
service) =>
        {
            var products = await service.SearchProductsAsync(query);
            return Results.Ok(products);
        })
            .WithName("AISearchProducts")
            .Produces<List<Product>>(StatusCodes.Status200OK);
    }
}
```

These mappings expose two distinct search endpoints:

- `GET /products/search/{query}` uses `ProductService` to perform traditional keyword matching.
- `GET /products/aisearch/{query}` uses `ProductAIService` to perform semantic search based on vector similarity.

Although both endpoints call a method named `SearchProductsAsync`, the underlying implementations are different. The traditional endpoint searches product names in the relational database, while the semantic endpoint generates an embedding for the query and compares it with the stored product vectors.

With both search strategies now exposed through the Catalog API, we can test how they respond to different types of queries.

Testing the Catalog search endpoints

To test the endpoints, we use the `Catalog.http` file in the Catalog project. Make sure Docker Desktop is running, set **AppHost** as the startup project, and start the Aspire application so that Catalog, Ollama, and the other required resources are available.

First, you scroll to the end of the `Catalog.http` file and add two new HTTP `GET` methods for invoking your search endpoints. One is for traditional search and the other for AI search. The

traditional search endpoint uses `/search/{query}` and the AI search endpoint uses `/aisearch/{query}`.

```
### Traditional Search
GET {{Catalog_HostAddress}}/search/Hiking
Accept: application/json
### AI Search
GET {{Catalog_HostAddress}}/aisearch/Something_for_rainy_days
Accept: application/json
```

You then right-click **AppHost** and run the application.

Start with the traditional-search request. Send the following request:

```
GET {{Catalog_HostAddress}}/products/search/Hiking
```

The request is routed to `ProductService`, which searches product names for the supplied term. If the catalog contains a product named `Hiking Poles`, the response resembles the following:

```
[
  {
    "id": 2,
    "name": "Hiking Poles",
    "description": "Ideal for camping and hiking trips",
    "price": 24.99,
    "imageUrl": "product2.png"
  }
]
```

This demonstrates the behavior of *conventional keyword search*: the query succeeds because the product name contains the word `Hiking`.

Next, send the semantic-search request:

```
GET {{Catalog_HostAddress}}/products/aisearch/Something_for_rainy_days
```

This request is routed to `ProductAIService`. On the first semantic search, the service initializes the product-vector collection if it does not already exist, generates embeddings for the products, and stores them in the in-memory vector store. It then generates an embedding for the user's query and compares it with the stored product vectors.

Because semantic search compares meaning rather than exact wording, a query such as `"something for rainy days"` can return a relevant product such as an `Outdoor Rain Jacket`, even though the query does not contain the product name itself. The first request may take longer because the product embeddings are initialized during that call.

You can use the **Traces** view in the Aspire dashboard to follow the semantic-search request through the Catalog microservice and observe the calls made to Ollama for embedding generation. Subsequent semantic searches reuse the initialized product vectors for as long as the in-memory store remains available.

With both endpoints verified, the backend now supports traditional and semantic product search through separate API routes. The next step is to integrate these endpoints into the Blazor frontend so that users can choose between the two search modes.

Integrating search into the Blazor frontend

With both Catalog search endpoints working, the next step is to expose them through the Blazor frontend. The client will use one API method to route each query to either the keyword or semantic endpoint based on the search mode selected by the user.

To build the Blazor frontend application for your semantic product search, you start by opening Visual Studio and navigating to the `frontend` folder to access the web application project. The first step is to arrange your API clients to integrate with the Catalog microservice.

Open the `CatalogApiClient` class, which already contains the HTTP integration with the Catalog microservice. Add a `SearchProducts` method that accepts the user's query and a **Boolean** `aiSearch` value. This value determines which Catalog endpoint the client calls:

```
public async Task<List<Product>?> SearchProducts(string query, bool aiSearch)
{
    if (aiSearch)
    {
        return await httpClient.GetFromJsonAsync<List<Product>>($"/products/
aisearch/{query}");
    }
    else
    {
        return await httpClient.GetFromJsonAsync<List<Product>>($"/products/
search/{query}");
    }
}
```

When `aiSearch` is `true`, the method sends the query to the semantic-search endpoint. Otherwise, it uses the traditional keyword-search endpoint. With this client-side routing in place, we can now build the page that allows users to enter a query and choose between the two search modes.

In the `Components/Pages` folder, add a new **Razor component** named `Search.razor`. This page will provide the user interface for the search functionality and connect the controls on the page to the `CatalogApiClient` method you just added. The page should do the following:

- Expose the `/search` route through the @page directive
- Inject `CatalogApiClient` to access the Catalog search endpoints
- Provide a text field for the user's search query
- Include a checkbox for switching between traditional and semantic search
- Provide a **Search** button that submits the query using the selected search mode
- Display the returned products in a table

The following code defines the search page:

```
@page "/search"
@attribute [StreamRendering(true)]
@rendermode InteractiveServer
@inject CatalogApiClient CatalogApiClient
<PageTitle>Search Products</PageTitle>
<h1>Search Products</h1>
<p>Search our amazing outdoor products that you can purchase.</p>
<div class="form-group">
<label for="search" class="form-label">Type your question:</label>
<div class="input-group mb-3">
<input type="text" id="search" class="form-control" @bind="searchTerm"
placeholder="Enter your search term" />
<button id="btnSearch" class="btn btn-primary" @onClick="DoSearch"
type="submit">Search</button>
</div>
<div class="form-check form-switch mb-3">
<InputCheckbox id="aiSearchCheckBox" @bind-Value="aiSearch" />
<label class="form-check-label" for="aiSearch">Use Semantic Search</label>
</div>
<hr />
</div>
@if (products == null)
```

```
{
    <p><em>Loading...</em></p>
}
else if (products.Count == 0)
{
    <p><em>No product found.</em></p>
}
else
{
    <table>
    <thead>
    <tr>
    <th>Image</th>
    <th>Name</th>
    <th>Description</th>
    <th>Price</th>
    </tr>
    </thead>
    <tbody>
    @foreach (var product in products)
    {
        <tr>
        <!-- Simulating images being hosted on a CDN -->
        <td><img height="80" width="80" src="https://raw.githubusercontent.com/
MicrosoftLearning/mslearn-dotnetmicroservices/main/images/products/
@product.Id.png" /></td>
        <td>@product.Name</td>
        <td>@product.Description</td>
        <td>@product.Price.ToString("C2")</td>
        </tr>
    }
    </tbody>
    </table>
}
@code {
    private string searchTerm = default!;
    private bool aiSearch = false;
    private List<Product>? products = [];
    private async Task DoSearch(MouseEventArgs e)
    {
        await Task.Delay(500);
```

```
            products = await CatalogApiClient.SearchProducts(searchTerm, aiSearch);
        }
    }
```

The `aiSearch` value bound to the checkbox determines which endpoint `CatalogApiClient` calls. When the user selects **Use Semantic Search**, the query is sent to `/products/aisearch/{query}`; otherwise, it is sent to `/products/search/{query}`. After the request completes, the returned `products` are assigned to products, causing the Blazor UI to render the results table.

With the search page implemented, add it to the WebApp navigation so that users can access it from the main menu. Open `NavMenu.razor` and add a new navigation item for the `/search` route:

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="search">
        <span class="bi bi-list-nested-nav-menu"
              aria-hidden="true"></span>
        Search
    </NavLink>
</div>
```

With the frontend now connected to both search endpoints, the complete search workflow is in place. The final step is to run the application and compare how traditional and semantic search respond to the same user queries.

Testing the complete search experience

The backend and frontend pieces are now connected, so the final step is to test the complete user flow and compare traditional keyword search with semantic search.

Start the `AppHost` project and use the **Aspire dashboard** to confirm that the required application resources are running. Wait for the **Catalog** microservice and **Blazor WebApp** to become ready, along with their supporting resources. Catalog references both the Ollama-hosted `Llama 3.2` and `all-minilm` models; the semantic-search workflow specifically uses `all-minilm` to generate embeddings.

The Aspire dashboard provides a convenient overview of these resources as they transition from **Starting** or **Waiting** to **Running**. Confirm that resources such as **ollama-all-minilm**, **ollama-llama3-2**, **catalogdb**, **Catalog**, and the **WebApp** are ready before continuing.

Once the application is ready, open the WebApp endpoint from the Aspire dashboard. The main navigation includes the existing pages together with the new **Search** option.

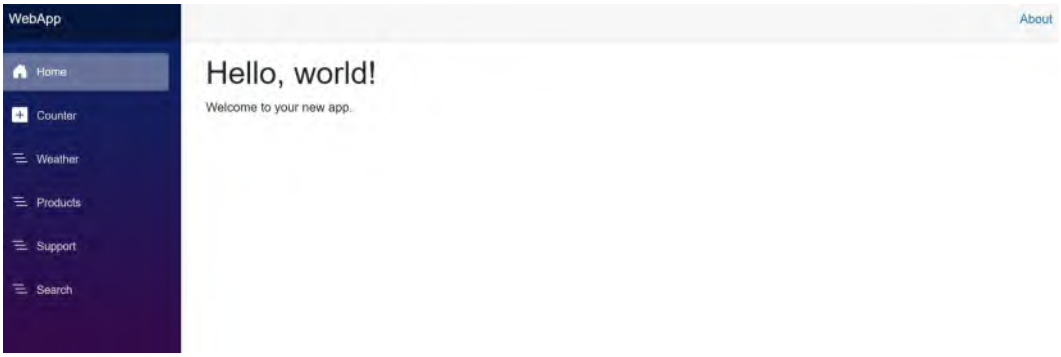


Figure 14.5 – Blazor WebApp navigation showing the Search page alongside the existing application pages.

Select **Search** to open the interface. The page provides a search field, a **Search** button, and a toggle for switching between traditional and semantic search.

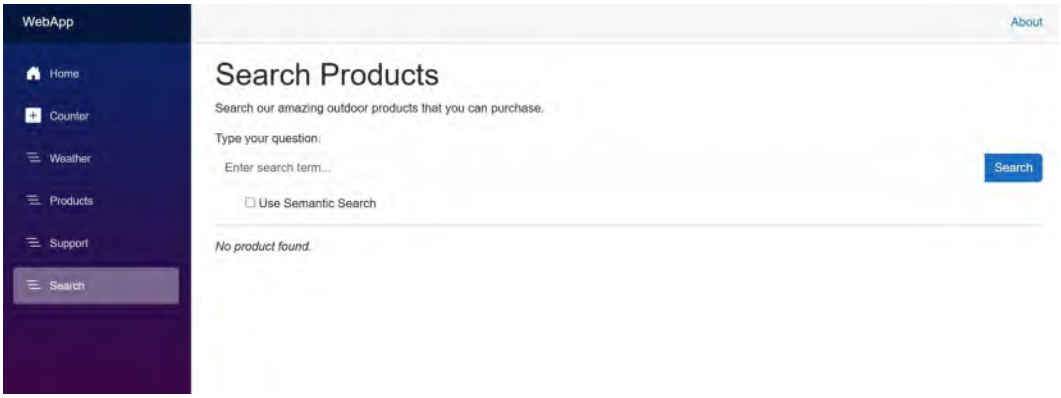


Figure 14.6 – Blazor WebApp Search page

Begin by testing traditional keyword search:

1. Enter a query such as `hiking`.
2. Leave **Use Semantic Search** disabled.
3. Select **Search**.

The application returns a matching product such as **Hiking Poles**, demonstrating that traditional search works when the supplied term appears directly in the product name.

Next, try a more descriptive query such as `something for rainy days` while leaving semantic search disabled. In this case, the traditional search returns no matching product because the product name does not contain the words in the query.

Now, test the same query using semantic search:

1. Enter `something for rainy days`.
2. Enable **Use Semantic Search**.
3. Select **Search**.

This time, the application can return a relevant product such as **Outdoor Rain Jacket**. Rather than looking for the exact words in the query, semantic search generates an embedding for the query and compares it with the stored product vectors to find the closest semantic match.

The first semantic search may take longer because the application initializes the product embeddings if the in-memory vector collection has not yet been created. Subsequent searches reuse those vectors for the lifetime of the application process.

You can try additional descriptive queries to observe how the returned results change based on semantic similarity, even when the query does not contain the product name itself.

At this point, you have verified the complete search workflow from the Blazor frontend through the Catalog API and `Services` layer to either PostgreSQL-backed keyword search or Ollama-powered vector search. The eShop application can now support both exact text matching and meaning-based product discovery.

Summary

In this chapter, we built an end-to-end semantic product search workflow using `Microsoft.Extensions.AI`, `Microsoft.Extensions.VectorData`, the `CommunityToolkit.VectorData.InMemory` provider, the `ProductVector` model, dedicated Catalog endpoints, and a Blazor search interface.

We began by comparing keyword and semantic search and then examined how vector embeddings represent meaning. We configured the Ollama-hosted `all-minilm` model, connected it to Catalog through Aspire, and used the current `Microsoft.Extensions.VectorData` abstractions to store and search product vectors. We then implemented a keyword-search baseline, added semantic search in `ProductAIService`, exposed both strategies through separate endpoints, and integrated them into a single Blazor page.

The final comparison showed the key distinction: keyword search relies on text matches, while semantic search retrieves products according to contextual similarity. The implementation uses an in-memory vector store for simplicity, but the same abstraction can be replaced with a persistent vector database when the application requires durable or larger-scale vector storage.

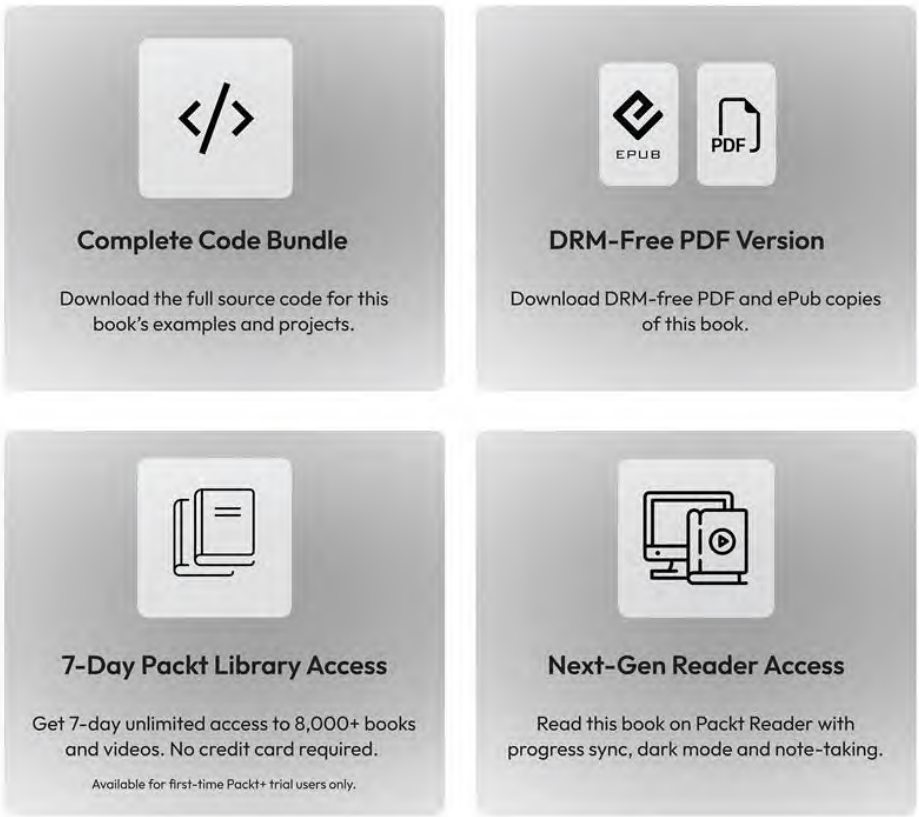
With this final capability in place, the eShop application now brings together the full set of concepts developed throughout the book: independently focused microservices, PostgreSQL and Redis backing services, synchronous and asynchronous communication, RabbitMQ and MassTransit messaging, Keycloak-based security, Blazor integration, Azure deployment, observability through Aspire, and generative AI capabilities for both customer support and semantic product search. You have taken the application from a basic distributed system to a cloud-native, AI-enabled solution in which infrastructure, communication, security, deployment, and intelligent features work together as one coherent architecture.

This completes the journey we began in *Chapter 1*. You now have a practical foundation for designing, building, securing, observing, deploying, and extending distributed applications with Aspire—and for adding AI capabilities where they provide meaningful value rather than treating them as separate from the rest of the system.

15

Unlock Your Exclusive Benefits

Your copy of this book includes the following exclusive benefits:



Follow the guide below to unlock them. The process takes only a few minutes and needs to be completed once.

Unlock this Book's Free Benefits in 3 Easy Steps

Step 1

Keep your purchase invoice ready for *Step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/en-us/unlock?step=1`.

> **NOTE**
>
> If you bought this book directly from Packt, no invoice is required. After *Step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to `packtpub.com/unlock`.

On the page that opens (similar to *Figure 15.1* on desktop), search for this book by name and select the correct edition.

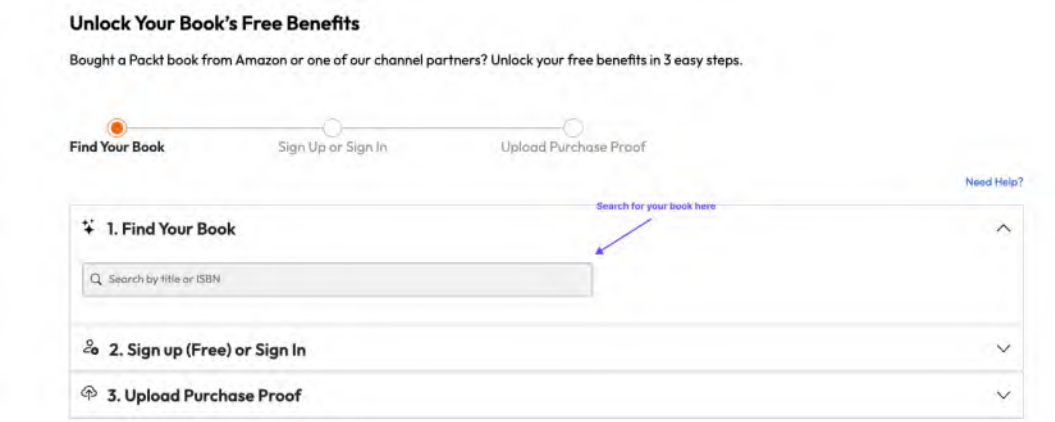


Figure 15.1 – Packt unlock landing page on desktop

Step 3

After selecting your book, sign in to your Packt account or create one for free. Then upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need Help

If you get stuck and need help, visit `https://www.packtpub.com/unlock-benefits/help` for a detailed FAQ on how to find your invoices and more. This QR code will take you to the help page.

NOTE

If you are still facing issues, reach out to `customercare@packt.com`.



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

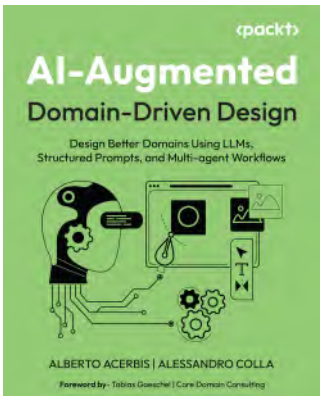
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



AI-Augmented Domain-Driven Design

Alberto Acerbis, Alessandro Colla

ISBN: 978-1-80742-105-2

- Understand how LLMs reason, where they help, and where they fail
- Apply prompt engineering to Domain-Driven Design workflows
- Use AI-assisted reasoning to refine domain events and bounded contexts
- Build rulebooks that guide and constrain LLM reasoning
- Apply AI to EventStorming and domain modeling activities
- Design AI agents for domain discovery and DDD workflows
- Orchestrate multi-agent systems for domain modeling
- Strengthen software architecture decisions while preserving domain clarity



Pragmatic Clean Code

Krystyna Ślusarczyk

ISBN: 978-1-80849-379-9

- Choose clear, expressive, and consistent names
- Design focused functions with effective signatures
- Apply SOLID principles to everyday design principles
- Reduce coupling between classes and improve cohesion within them
- Organize classes and projects so the codebase is easy to navigate
- Write readable unit tests and improve testability
- Refactor messy C# code to make it readable, testable, and easy to extend

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Subscribe to Deep Engineering

Join thousands of developers and architects who want to understand how software is changing, deepen their expertise, and build systems that last.

Deep Engineering is a weekly expert-led newsletter for experienced practitioners, featuring original analysis, technical interviews, and curated insights on architecture, system design, and modern programming practice.

Scan the QR or visit the link to subscribe for free.



`https://packt.link/deep-engineering-newsletter`

Index

A

access token **223**

advanced message queuing protocol (AMQP) **178**

AI-powered product support service **266**

AI support endpoint **266**

AppHost project **4, 28–159, 179–229, 269**
- backing services 97
- dashboard 9, 41
- dependencies 25
- orchestration 41, 228
- resources 23
- services 179

ASP.NET Core Minimal APIs **134**

ASP.NET Core patterns **137**

Aspire **1, 21–133, 177–268**
- AppHost project 9
- Aspire application 251
- backing services 96
- client integrations 31
- dashboard 5
- hosting integration 33
- integration 228
- service discovery 227

Aspire application **251**

Aspire integrations **93**

Aspire orchestration **82**
- Visual Studio 92

Aspire project responsibilities **47**

Aspire project structure **42**
- orchestration 41

Aspire Service Discovery **161, 227**

AspireSample.ApiService **49**

AspireSample.AppHost **43**

AspireSample.ServiceDefaults **48**

AspireSample.Web **49**

asynchronous communication **176**
- architecture 185
- event flow 175

azd down command **263**

Azure account **255**

Azure Container Apps **74, 249**
- Aspire application 251
- Azure account 255
- prerequisites 255
- services 249

Azure Developer CLI **251**
- Aspire application 251
- Azure account 255
- prerequisites 256

Azure portal **250**

Azure resources **262**

B

backing services **14, 99, 249**
- AI backing services 17
- LLM inference services 16
- vector databases 16

Basket API endpoints **150**

Basket CRUD operations **136**

Basket endpoints **134, 221**

Basket.http file **153**

Basket microservice **3, 73–227**
- authentication 205
- authorization 207
- Basket endpoints 219
- data storage 134

BasketService **193**

BasketService class **148**
- logic 134

Blazor **266**

Blazor AI support feature **266**

Blazor frontend **289**

Blazor web application **228**
- output caching 245
- Products page 238

Blazor WebApp **313**

business logic **84**

C

Catalog API client **228**
Catalog data layer **82**
business logic 84
dependency injection 102
Catalog.http file **127**
Catalog microservice **3, 72–188, 227, 266, 290**
orchestration 87
services 93
Visual Studio 89
Catalog microservice endpoints **126**
Catalog search endpoints **310**
Catalog service layer **115**
CatalogApiClient **169**
CatalogApiClient class **233**
CI/CD **13**
client **214**
client integrations **31, 93, 147**
cloud-native distributed architectures **12**
containers 12
microservices 12
observability 14
cloud-native principles **2**
CommunityToolkit.VectorData.InMemory **297**
container orchestrators **13**
container resources **38**
CreateProductAsync **121**

D

data access layer **84**
data persistence **98**
data seeding **132**
data storage **136**
data Transfer objects (DTOs) **86**
data volumes **249**
DataSeeder class **108**
dependency injection **118–167**
deployment **249**
development plan **78**
development roadmap **72**
DevOps **13**
distributed application architecture **5**
distributed applications **21, 41, 316**
orchestration 47
resources 26
distributed cache **134**
distributed tracing **14**
domain layer **84**
domain model **82**
dual-write problem **192**

E

embedding generator **304**
embedding model **289**
end-to-end data flow **162**
Entity Framework Core **103**
Entity Framework Core migration **82**
data 84
data seeding 113
Visual Studio 111
environment awareness **25**
eShop application **2, 205–249, 316**
Aspire application 255
Basket microservice 3
client 213
prerequisites 255
service communication 1
eShop distributed architecture **72**
frontend 75
eShop distributed microservices architecture **71**
frontend 71
eShop distributed microservices development **72**
exclusive benefits **317**

F

fundamental ASP.NET Core and Entity Framework Core patterns **85**
orchestration 25, 85

G

generative AI capabilities **288**
GlobalUsings.cs **103–168**

H

help **319**
Host Builder **33**
hosting integration **33, 94**
backing services 97
HTTP communication **162**
HTTP PUT operation **122**

I

IChatClient **278**
identity and access management **206**
authorization 205
IDistributedCache **193**
IntegrationEvent **183**
integrations **24**
client integrations 31
hosting integration 34

K

Keycloak **205**
Basket endpoints 205
client 206
Keycloak Hosting **205**
Keycloak realm **219**
KISS principle **137**

L

large language models **265**
parameters 267
LLM inference services **17**
LLMs **268**
models 267
parameters 267

M

MapProductEndpoints **120**
MassTransit **175**
Microsoft Agent Framework **265**
Microsoft.Extensions.AI **265, 302**
Microsoft.Extensions.VectorData **297**
migration class **105**
migration files **105**

N

n-layer architecture pattern **84**
domain layer 92

O

OAuth 2.0 **206**
Ollama **265, 294**
integration in Aspire 266
OllamaSharp chat client 277
OllamaSharp chat client **276**
OpenID Connect **206**
OpenID Connect client **212**
outbox pattern **192**
output caching **227**

P

pgAdmin **112**
PostgreSQL **81**
backing services 100
PostgreSQL client integration **82**
presentation layer **84**
Product class **232**
Product endpoints **82**
ProductService 119
Product entity **121, 162**
Product model **161**
product-price-change event flow **176**
product-price-change events **176**
product retrieval **130**
Product service layer **82**
business logic 84
dependency injection 118
ProductAIService **277, 304**
ProductDbContext **108**
ProductEndpoints **119**
ProductPriceChangedIntegrationEvent **184**
Products page **228**
Products Razor page 241
Products Razor page **228**
ProductService **115, 301**
ProductVector model **299**

Program.cs **118**
Program.cs file **149**
project folder structure **115, 137**
project references **36**
provider-independent AI integration **266**
purchase invoice **318**

R

RabbitMQ **175**
Redis **136, 243**
Redis cache **42**
Redis Insight **156**
RequireAuthorization extension method **221**
authorization 220
resource consumption **31**
resource provisioning **31**
REST API endpoints **82, 136**

S

search endpoints **290**
semantic product search **289**
embedding generator 302
model 289
traditional databases 291
vectors 289
semantic search **274, 289**
embedding generator 302
service communication **6, 58**
service discovery **13–42, 161**
environment awareness 25
resources 21
ServiceDefaults project **49–140, 185, 229**
setup verification **255**
shared configuration **42**
shared integration events **176**
ShoppingCart **141**
ShoppingCartItem **142**
small language models (SLMs) **267**
solution organization **2, 72**
synchronous communication **162**

T

test user **217**
traditional databases **291**
traditional keyword search **290**

U

UseMigration extension method **107**

V

vector databases **18, 268, 290**
services 268
vector embeddings **290**
vector-search integrations **290**
vector store **302**
embedding generator 303
generator 304
Visual Studio **42, 88, 148, 228**
dashboard 45