HALLO

.NET 9.0

Practical ASP.NET Core Minimal API

AGUS KURNIAWAN

Hallo .NET 9.0: Practical ASP.NET Core Minimal API

Agus Kurniawan

Ilmu Data

21 August 2025

Table of Contents

Preface
Acknowledgments
1 Introduction
- 1.1 Overview of .NET 9.0
  - 1.1.1 Unified Platform
  - 1.1.2 Performance Enhancements
  - 1.1.3 Improved Cloud and Container Support
  - 1.1.4 Enhanced C# Language Features
  - 1.1.5 Blazor and WebAssembly Innovations
  - 1.1.6 Expanded AI and Machine Learning Capabilities
  - 1.1.7 Better Security and Compliance
  - 1.1.8 Enhanced Tooling and Development Experience
- 1.2 Understanding ASP.NET Core Minimal API
- 1.3 Benefits of Using Minimal APIs
- 1.4 Best Practices and Use Cases
- 1.5 Setting Up the Development Environment
  - 1.5.1 Installing .NET 9.0 SDK
  - 1.5.2 Installing SSL Certificates Development Tool
  - 1.5.3 Setting Up an Integrated Development Environment (IDE)
  - 1.5.4 Verifying the Setup
  - 1.5.5 Additional Tools and Extensions

2 ASP.NET Core Minimal API Development
- 2.1 Introduction
- 2.2 Exercise 1: Hello World - ASP.NET Core Minimal API
  - 2.2.1 Objective
  - 2.2.2 Requirements
  - 2.2.3 Lab Steps
  - 2.2.4 Conclusion
- 2.3 Exercise 2: RESTful Service Request and Response
  - 2.3.1 Objective
  - 2.3.2 Requirements
  - 2.3.3 Lab Steps

2.3.4 Conclusion
2.4 Exercise 3: OpenAPI Documentation
2.4.1 Conclusion
2.5 Exercise 4: Buidling a Calculator Service
2.5.1 Objective
2.5.2 Requirements
2.5.3 Lab Steps
2.5.4 Conclusion
2.6 Exercise 5: Upload and Download File Web
2.6.1 Objective
2.6.2 Requirements
2.6.3 Lab Steps
2.6.4 Conclusion
2.7 Exercise 6: Exception Handling and Logging
2.7.1 Objective
2.7.2 Requirements
2.7.3 Lab Steps
2.7.4 Conclusion
2.8 Exercise 7: Middleware and Filters
2.8.1 Objective
2.8.2 Requirements
2.8.3 Lab Steps
2.8.4 Conclusion
3 Accessing SQL and NoSQL Databases
3.1 Introduction
3.2 .NET Entity Framework Core
3.3 Entity Framework Core tools
3.4 Exercise 8: EF Core 9.0 Code First and ASP.NET Core Minimal API
3.4.1 Objective
3.4.2 Requirements
3.4.3 Lab Steps
3.4.4 Conclusion
3.5 Exercise 9: EF Core 9.0 Database First and ASP.NET Core Minimal API
3.5.1 Objective
3.5.2 Requirements

3.5.3 Lab Steps
3.5.4 Conclusion
3.6 Introduction to Database Transactions
3.7 Exercise 10: Database Transaction
3.7.1 Objective
3.7.2 Requirements
3.7.3 Lab Steps
3.7.4 Conclusion
3.8 Introduction to NoSQL Databases
3.9 Exercise 11: NoSQL Database and ASP.NET Core Minimal API
3.9.1 Objective
3.9.2 Requirements
3.9.3 Lab Steps
3.9.4 Conclusion
4 Deep Dive into Web Security
4.1 Introduction
4.2 Exercise 12: Authentication and Authorization
4.2.1 Objective
4.2.2 Requirements
4.2.3 Lab Steps
4.2.4 Conclusion
4.3 Exercise 13: Role-Based Access Control (RBAC)
4.3.1 Objective
4.3.2 Requirements
4.3.3 Lab Steps
4.3.4 Conclusion
4.4 Exercise 14: Data Privacy and Protection
4.4.1 Objective
4.4.2 Requirements
4.4.3 Lab Steps
4.4.4 Conclusion
4.5 Exercise 15: Rate Limiting and Throttling
4.5.1 Objective
4.5.2 Requirements
4.5.3 Lab Steps
4.5.4 Conclusion

4.6 Exercise 16: Configuring CORS in ASP.NET Core 9.0 Minimal API
4.6.1 Objective
4.6.2 Requirements
4.6.3 Lab Steps
4.6.4 Conclusion
5 Monitoring and Deployment
5.1 Introduction
5.1.1 Monitoring in ASP.NET Core 9.0 Minimal API
5.1.2 Deployment of ASP.NET Core 9.0 Minimal API
5.2 Exercise 17: Health Check and Monitoring
5.2.1 Objective
5.2.2 Requirements
5.2.3 Lab Steps
5.2.4 Implement Custom Health Checks (Advanced)
5.2.5 Conclusion
5.3 Exercise 18: Deploying to Web Server IIS
5.3.1 Objective
5.3.2 Requirements
5.3.3 Lab Steps
5.3.4 Conclusion
5.4 Exercise 19: Deploying to Linux Server with Nginx
5.4.1 Objective
5.4.2 Requirements
5.4.3 Lab Steps
5.4.4 Conclusion
5.5 Exercise 20: Deploying to Container Platforms
5.5.1 Objective
5.5.2 Requirements
5.5.3 Lab Steps
5.5.4 Conclusion
Appendix A: C# Cheat Sheet
Appendix B: Resources
SQL Server 2025 High Availability & Disaster Recovery: Always On Solutions Course
Enhance Your Learning with Our Udemy Course
Dive Deeper into Containerization with Our Udemy Course

Build Secure PHP APIs Like a Pro with Laravel 12, OAuth2, and JWT
Master Real-World Logging & Visualization with the Full ELK Stack
Appendix C: Source Code
About

Preface

In the ever-evolving landscape of software development, staying abreast of the latest technologies and frameworks is not just a requirement but a necessity. With the release of .NET 9.0, Microsoft continues its tradition of offering robust, efficient, and forward-thinking solutions for developers worldwide. Among its many features, ASP.NET Core Minimal API stands out as a revolutionary approach to building web APIs with less boilerplate code, enabling developers to focus more on their application's core functionality.

"Hallo .NET 9.0: Practical ASP.NET Core Minimal API" is crafted to guide you through the journey of understanding and implementing Minimal APIs in .NET 9.0. This book aims to equip you with the knowledge and practical skills to build efficient and scalable web APIs using the new features and paradigms introduced in .NET 9.0. Whether you are a seasoned developer or just starting, this book will serve as a comprehensive resource for mastering Minimal APIs.

Throughout this book, we'll delve into various aspects of ASP.NET Core Minimal API, starting from the basics and gradually moving to more advanced topics. You'll learn about:

What will you learn:

- The fundamentals of .NET 9.0 and its improvements over previous versions.
- Step-by-step guidance on setting up and configuring a Minimal API project.
- Best practices for structuring your projects for maintainability and scalability.
- Advanced features like dependency injection, middleware integration, and data access.
- Practical examples demonstrating the use of Minimal APIs in real-world scenarios.

- Deployment strategies and performance optimization tips.

This book is designed for a wide range of readers - from beginners who have a basic understanding of .NET and C# to experienced developers looking to enhance their skills with the latest advancements in .NET 9.0. It’s also a valuable resource for IT professionals, students, and anyone interested in developing modern web applications efficiently.

Agus Kurniawan

Depok, August 2025

Acknowledgments

A heartfelt thank you to the community of developers, contributors, and technology enthusiasts whose feedback and insights have been invaluable in shaping this book.

As you turn these pages, I hope you find “Hallo .NET 9.0: Practical ASP.NET Core Minimal API” not only informative but also inspiring, opening new avenues for your creativity and innovation in the world of software development.

1 Introduction

1.1 Overview of .NET 9.0

.NET 9.0 marks a significant milestone in the evolution of Microsoft's .NET platform. Building upon the foundation laid by its predecessors, .NET 9.0 brings a host of new features, improvements, and optimizations that cater to the needs of modern software development. This section delves into the core aspects of .NET 9.0, highlighting its most impactful changes and how they benefit developers and businesses alike.

1.1.1 Unified Platform

.NET 9.0 continues the journey towards a unified framework, further integrating capabilities across different .NET components. This unification simplifies the development process, making it easier for developers to build applications that run seamlessly across various platforms, including Windows, Linux, macOS, and mobile devices.

1.1.2 Performance Enhancements

One of the hallmark features of .NET 9.0 is its enhanced performance. The runtime and core libraries have undergone significant optimizations, resulting in faster application startup times, reduced memory footprint, and improved overall efficiency. These enhancements ensure that applications built on .NET 9.0 are not only faster but also more resource-efficient.

1.1.3 Improved Cloud and Container Support

.NET 9.0 offers enhanced support for cloud and container-based environments. With native cloud integration and optimized container performance, it becomes a go-to choice for building cloud-native applications. This version also includes features that simplify the development and deployment of microservices and serverless applications.

1.1.4 Enhanced C# Language Features

C# continues to evolve alongside .NET, and version 9.0 of the framework fully supports the latest version of the C# language. This update introduces new language features that promote cleaner, more maintainable code, enhancing developer productivity and application maintainability.

1.1.5 Blazor and WebAssembly Innovations

Blazor, Microsoft's framework for building interactive web UIs with C#, receives significant updates in .NET 9.0. These include performance improvements and new features for Blazor WebAssembly, enabling developers to build highly performant client-side web applications.

1.1.6 Expanded AI and Machine Learning Capabilities

.NET 9.0 expands its capabilities in AI and machine learning with enhanced ML.NET libraries. This makes it easier for developers to integrate machine learning into their applications, leveraging the power of AI for more intelligent, data-driven solutions.

1.1.7 Better Security and Compliance

Security is a top priority in .NET 9.0, with strengthened security measures and compliance features. This ensures that applications built using .NET 9.0 are not only robust and performant but also secure and compliant with the latest industry standards.

1.1.8 Enhanced Tooling and Development Experience

Finally, .NET 9.0 brings improvements to its tooling and development experience. With a more refined Visual Studio integration, developers can enjoy a smoother, more efficient development workflow, complete with powerful debugging and diagnostic tools.

1.2 Understanding ASP.NET Core Minimal API

Minimal APIs in ASP.NET Core 9 represent a streamlined, low-ceremony method for building HTTP APIs. They are designed to reduce the complexity and boilerplate code traditionally associated with setting up a new API in ASP.NET Core.

Key Features of Minimal APIs

- **Simplicity and Conciseness:** Emphasize how Minimal APIs reduce the amount of code needed to start a basic API project, making them ideal for small services or microservices.
- **Routing and Middleware Integration:** Explain how routing is handled in Minimal APIs and how they seamlessly integrate with existing middleware in the ASP.NET Core ecosystem.
- **Dependency Injection:** Describe how Minimal APIs support dependency injection, allowing services to be injected directly into route handlers.

Building a Minimal API

- **Setting Up:** Step-by-step guide on setting up a new project with a Minimal API in ASP.NET Core 9, including the required NuGet packages and project configuration.
- **Defining Endpoints:** Demonstrate how to define HTTP endpoints (GET, POST, PUT, DELETE) using the simplified syntax of Minimal APIs.
- **Request and Response Handling:** Overview of handling requests and sending responses, including parsing query parameters and returning different types of responses (JSON, plain text, etc.).

1.3 Benefits of Using Minimal APIs

- **Reduced Complexity:** Discuss how Minimal APIs make it easier to build and maintain small to medium-sized APIs by reducing the layers of abstraction and the amount of boilerplate code.
- **Improved Performance:** Highlight any performance improvements associated with using Minimal APIs, particularly in terms of memory footprint and startup time.

- **Flexibility and Testability:** Explain how the simplicity of Minimal APIs offers greater flexibility in development and makes unit testing more straightforward.

1.4 Best Practices and Use Cases

- **When to Use Minimal APIs:** Provide guidance on scenarios where Minimal APIs are the most beneficial, such as microservices, small web services, or when building APIs for simple applications.
- **Best Practices:** Share best practices for structuring and organizing Minimal API projects, error handling, and security considerations.

1.5 Setting Up the Development Environment

To effectively work with .NET 9.0 and utilize Native AOT, it's crucial to set up a robust development environment. This section walks you through the steps to prepare your system for .NET 9.0 development, focusing on tools, software, and configurations necessary for taking full advantage of Native AOT.

1.5.1 Installing .NET 9.0 SDK

- **Download the SDK:** Begin by downloading the .NET 9.0 Software Development Kit (SDK) from the official Microsoft .NET website. Ensure you select the correct version for your operating system.

- **Installation Process:** Follow the installation instructions specific to your OS. This typically involves running an installer or executing a set of commands in the terminal.

1.5.2 Installing SSL Certificates Development Tool

To enable HTTPS in your ASP.NET Core applications, you need to install the SSL certificates development tool. This tool is available as a .NET global tool, which you can install using the following command:

```
dotnet tool install --global dotnet-dev-certs
```

This command installs the `dotnet-dev-certs` tool globally, allowing you to manage development certificates for HTTPS in your applications.

If you use Ubuntu, you may need to install the `libnss3-tools` package before installing the tool. Now you can use the tool to generate and install the required certificates:

```
dotnet dev-certs https
sudo -E dotnet dev-certs https -ep /usr/local/share/ca-certificates/aspnet/https.crt
--format PEM
```

1.5.3 Setting Up an Integrated Development Environment (IDE)

- **Choosing an IDE:** Visual Studio, Visual Studio Code, or JetBrains Rider are recommended IDEs for .NET development. Choose one that best fits your development style and needs.

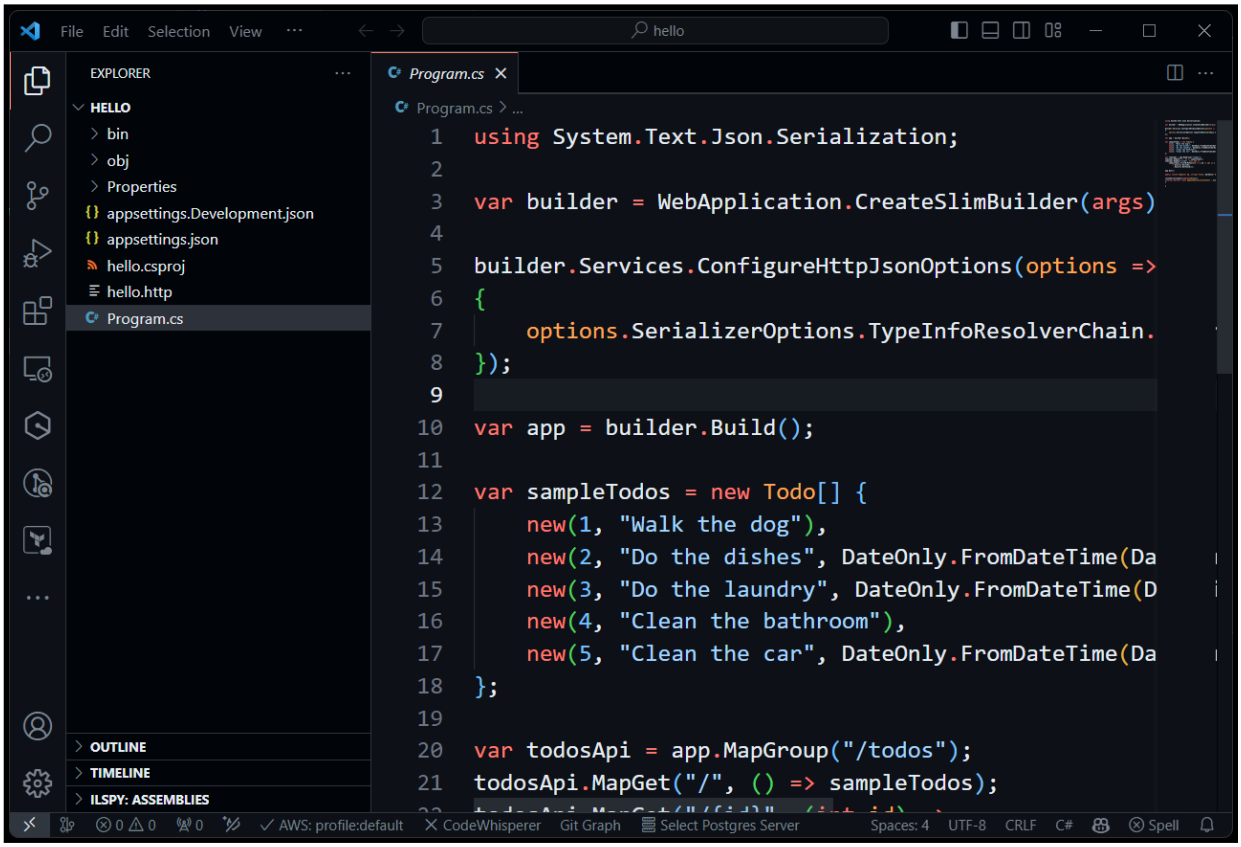


Figure 1.1 Visual Studio Code.

- **IDE Configuration:** Install the necessary extensions or plugins for .NET development. For Visual Studio, the .NET desktop development

workload is essential. For Visual Studio Code, the C# extension by OmniSharp is required.

- In this book, we will use Visual Studio Code for all code examples and demonstrations.

- We will also use the C# extension by **REST Client**(Huachao Mao) for Visual Studio Code, https://marketplace.visualstudio.com/items?itemName=humao.rest-client. This extension allows us to test our APIs using a simple text file. The extension is shown in Figure 1.2.

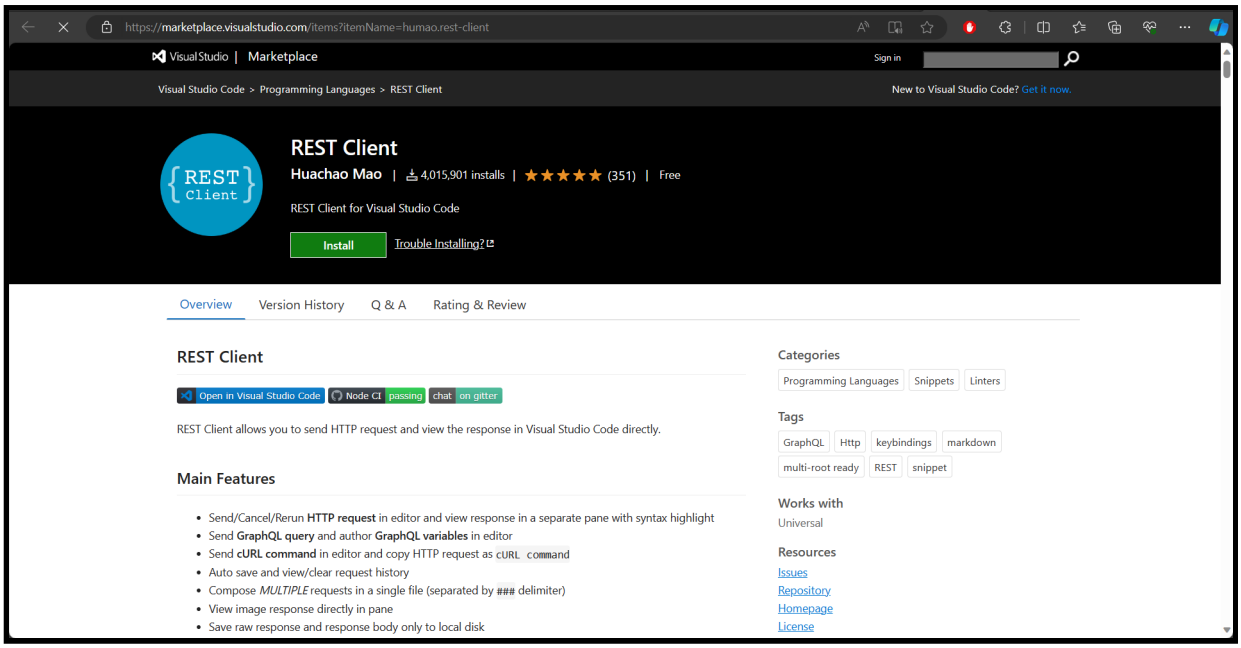


Figure 1.2 Visual Studio Code extension for REST Client.

1.5.4 Verifying the Setup

- **Testing the .NET Installation:** After installing the SDK, open a command prompt or terminal and run `dotnet --version` to verify the installation.

- **Creating a Test Project:** Create a simple .NET project using the command line or your IDE to ensure everything is working correctly. This can be a basic “Hello World” application.

- **Experimenting with ASP.NET Core Minimal API:** Compile the test project with ASP.NET Core Minimal API to confirm that your setup supports this feature. Monitor the compilation process and output for any errors or issues.

1.5.5 Additional Tools and Extensions

- **Source Control Integration:** Consider installing Git and integrating it with your IDE for version control.

- **Debugging and Diagnostic Tools:** Familiarize yourself with debugging tools available in your IDE, as they are essential for development and troubleshooting.

2 ASP.NET Core Minimal API Development

2.1 Introduction

Welcome to the exciting world of ASP.NET Core Minimal API development in .NET 9.0. This chapter serves as your comprehensive guide, combining the robust features of ASP.NET Core Minimal API with the performance improvements offered by .NET 9.0. It's a journey through innovative and efficient web development techniques.

Chapter Overview

In this chapter, we embark on a series of exercises designed to build your proficiency in ASP.NET Core 9.0 Minimal API:

1. **Exercise 1: Hello World - ASP.NET Core Minimal API**
    - Kickstart your journey with a simple "Hello World" application, introducing the basics of ASP.NET Core Minimal API.
2. **Exercise 2: RESTful Service Request and Response**
    - Dive deeper into the creation of a RESTful service, focusing on handling requests and sending responses.
3. **Exercise 3: Swagger API Documentation**
    - Learn how to implement Swagger for your API documentation, making your web services more accessible and easier to use.
4. **Exercise 4: Building a Calculator Service**
    - Develop a calculator service to understand handling different types of HTTP requests and performing basic operations.
5. **Exercise 5: Upload and Download File Web API**
    - Explore the process of setting up APIs for file upload and download, a common requirement in modern web applications.
6. **Exercise 6: Exception Handling and Logging**
    - Delve into best practices for exception handling and logging to build robust and reliable web applications.
7. **Exercise 7: Middleware and Filters**
    - Conclude with an exploration of middleware and filters, crucial for managing HTTP requests and responses in your API.

Preparing for Hands-On Labs

These exercises are designed to be hands-on and engaging, providing practical experience in building, optimizing, and deploying ASP.NET Core Minimal API using.NET 9.0. Each exercise builds on the previous one, ensuring a comprehensive understanding of the subject.

2.2 Exercise 1: Hello World - ASP.NET Core Minimal API

In this exercise, you'll create a basic "Hello World" application using ASP.NET Core 9.0 Minimal API. You'll learn how to set up, modify, and run a compiled ASP.NET Core Minimal API project, gaining a foundation in this innovative approach to .NET development.

2.2.1 Objective

Create a basic "Hello World" Web API application using ASP.NET Core 9.0 with the `webapi` template, showcasing ASP.NET Core Minimal API capabilities.

2.2.2 Requirements

- .NET 9.0 SDK installed
- A preferred code editor (e.g., Visual Studio, Visual Studio Code)
- Basic understanding of ASP.NET Core and C#

2.2.3 Lab Steps

Here's a step-by-step guide to creating your first ASP.NET Core Minimal API project:

1. Create a New ASP.NET Core Minimal API Project

- Open a command prompt or terminal.
- Navigate to the directory where you want to create the project.
- Or create a new directory for the project:

```
mkdir hello
cd hello
```

- Run the following command to create a new ASP.NET Core Minimal API project template:

```
dotnet new webapi
```

- Open the project in your code editor:

```
code .
```

- You should see the following project structure in your code editor:

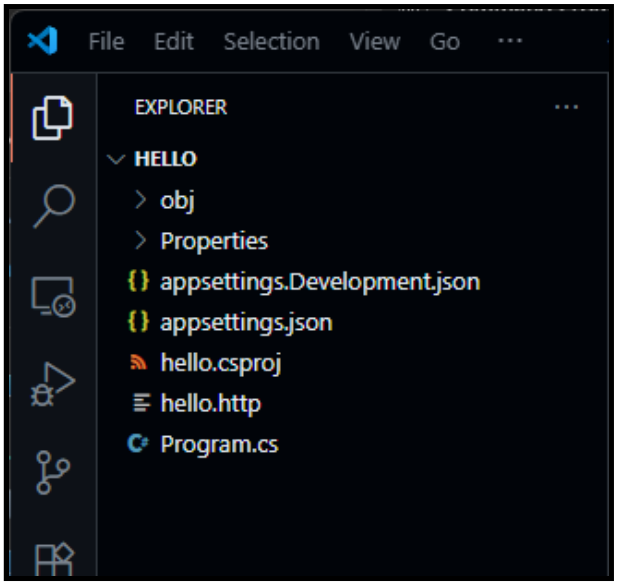


Figure 2.1 Project structure of the ASP.NET Core Minimal API project.

2. Build and Run the Application

- Return to the command prompt or terminal.
- Run the following command to build the project:

```
dotnet build
```

- To run the application, you can use the following command:

```
dotnet run
```

- To run the application with https profile, you can use the following command:

```
dotnet run --launch-profile https
```

3. Test the API

- Open a web browser or use a tool like Postman.
- Navigate to `http://localhost:<server-port>/weatherforecast`.
- Change the port number `<server-port>` to the port number displayed in the command prompt or terminal.
- You should see a response displaying weather forecast.

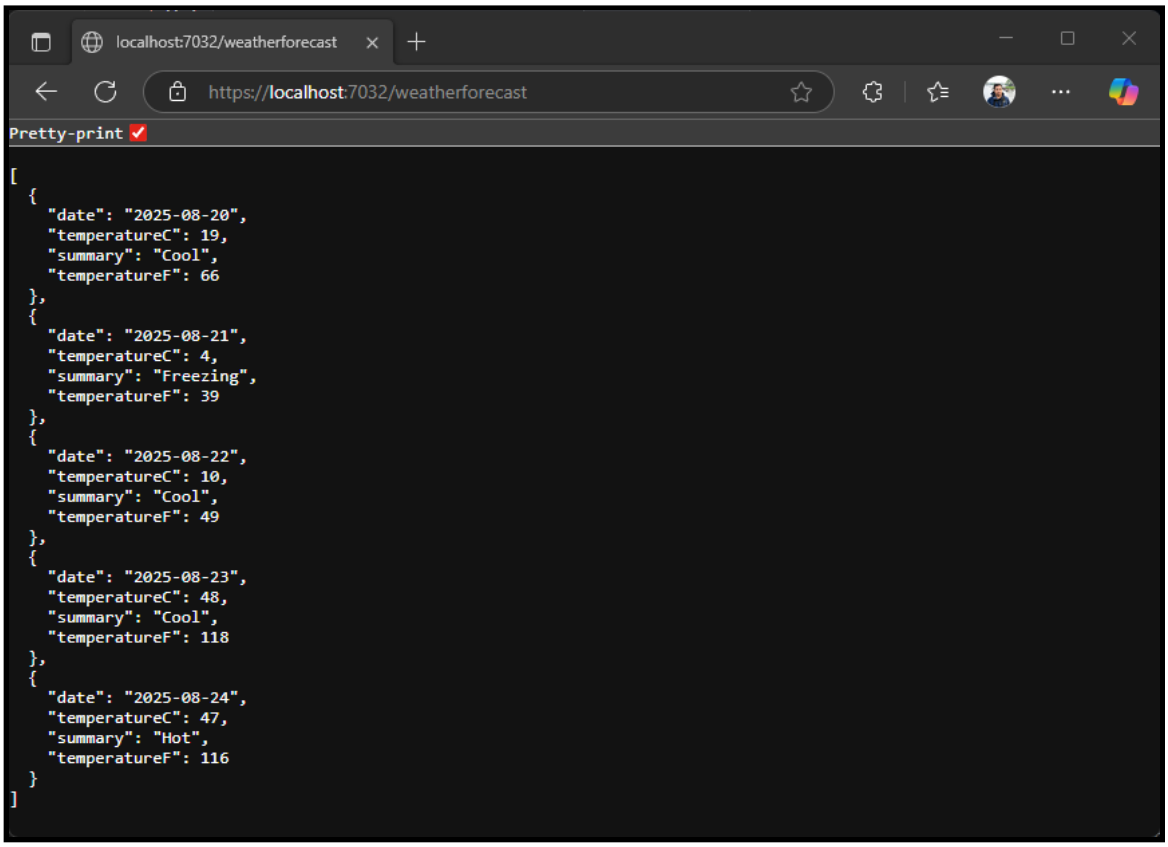


Figure 2.2 Output weather forecast application.

- We can also use the REST Client extension to test the API. The following is the content of the `hello.http` file:

```
@hello_HostAddress = http://localhost:5259

GET {{hello_HostAddress}}/weatherforecast
Accept: application/json

###
```

- Modify port `5259` to the port number displayed in the command prompt or terminal.
- Click the **Send Request** button to test the API.
- You should see a response displaying weather forecast.

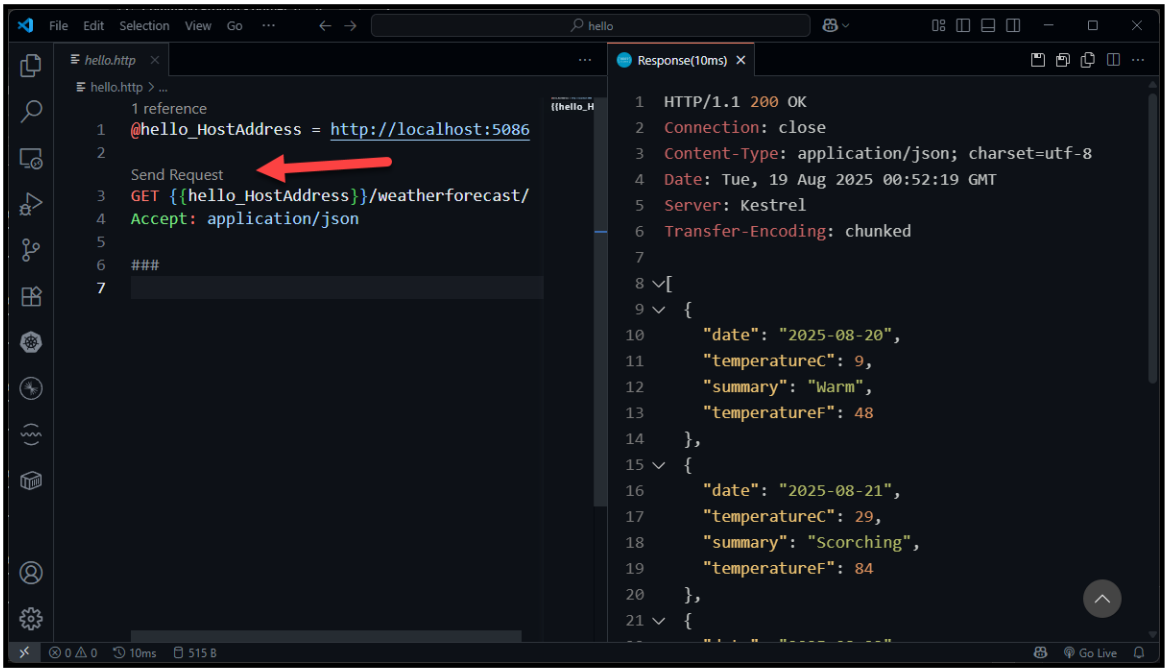


Figure 2.3 Test using http file with REST Client tool.

2.2.4 Conclusion

This lab has guided you through creating a basic “Hello World” application using the ASP.NET Core 9.0 Minimal API. You’ve seen firsthand how to set up, modify, and run a compiled ASP.NET Core Minimal API project, gaining a foundation in this innovative approach to .NET development.

2.3 Exercise 2: RESTful Service Request and Response

This lab provides a practical introduction to creating RESTful services with ASP.NET Core 9 using the minimal API approach. It’s designed to offer hands-on experience with the core concepts of RESTful service development in a modern and efficient way.

2.3.1 Objective

Develop a RESTful service using ASP.NET Core 9 Minimal API that handles various HTTP methods (GET, POST, PUT, DELETE) and provides appropriate responses.

2.3.2 Requirements

- .NET 9.0 SDK installed
- A preferred code editor (e.g., Visual Studio, Visual Studio Code)

2.3.3 Lab Steps

1. Create a New ASP.NET Core Minimal API Project

- Open a command prompt or terminal.
- Navigate to your desired working directory.
- Run the following command to create a new ASP.NET Core Minimal API project:

```
mkdir restfulapi
cd restfulapi
dotnet new webapi
```

- Open the project in your code editor:

```
code .
```

2. Modify the Project for Minimal API

- Open the project in your code editor.
- Remove all endpoints from the `Program.cs` file.
- Add the content of GET Restfull API with the following code:

```
...

// In-memory data store
var items = new List<string>();
for (int i = 1; i <= 5; i++)
{
   items.Add($"Item {i}");
}

// GET endpoint
app.MapGet("/items", () => items);
```

```
...
```

- Add the content of POST Restfull API with the following code:

```
// POST endpoint
app.MapPost("/items", (string item) =>
{
    items.Add(item);
    return Results.Created($"/items/{items.Count - 1}", item);
});
```

- Add the content of PUT Restfull API with the following code:

```
// PUT endpoint
app.MapPut("/items/{id}", (int id, string item) =>
{
    if (id < 0 || id >= items.Count)
    {
        return Results.NotFound();
    }
    items[id] = item;
    return Results.NoContent();
});
```

- Add the content of DELETE Restfull API with the following code:

```
// DELETE endpoint
app.MapDelete("/items/{id}", (int id) =>
{
    if (id < 0 || id >= items.Count)
    {
        return Results.NotFound();
    }
    items.RemoveAt(id);
    return Results.Ok();
});
```

3. Build and Run the Application

- Use the command prompt or terminal to build and run the project:

```
dotnet run
```

- You can also run the application with https profile:

```
dotnet run --launch-profile https
```

4. Testing the API

- You can use tools like Postman or the built-in REST client in Visual Studio Code to test your API.

- Test the following endpoints:
    - **GET** `/items`: Retrieve a list of items.
    - **POST** `/items`: Add a new item (include the item as a plain text in the request body).
    - **PUT** `/items/{id}`: Update an item at a specific index (provide the index in the URL and the new item as plain text in the request body).
    - **DELETE** `/items/{id}`: Delete an item at a specific index.
- We also test using the REST Client extension. The following is the content of the `restfulapi.http` file:

```
@restfulapi_HostAddress = http://localhost:5013

GET {{restfulapi_HostAddress}}/items
Accept: application/json

###
POST  {{restfulapi_HostAddress}}/items?item=item 20
Accept: application/json

###

PUT {{restfulapi_HostAddress}}/items/2?item=item 2-edited
Accept: application/json

###
DELETE  {{restfulapi_HostAddress}}/items/1
Accept: application/json

###
```

- Modify port `5013` to the port number displayed in the command prompt or terminal or based your configuration.
- Click the **Send Request** button to test the API.
- You should see a response displaying the list of items.

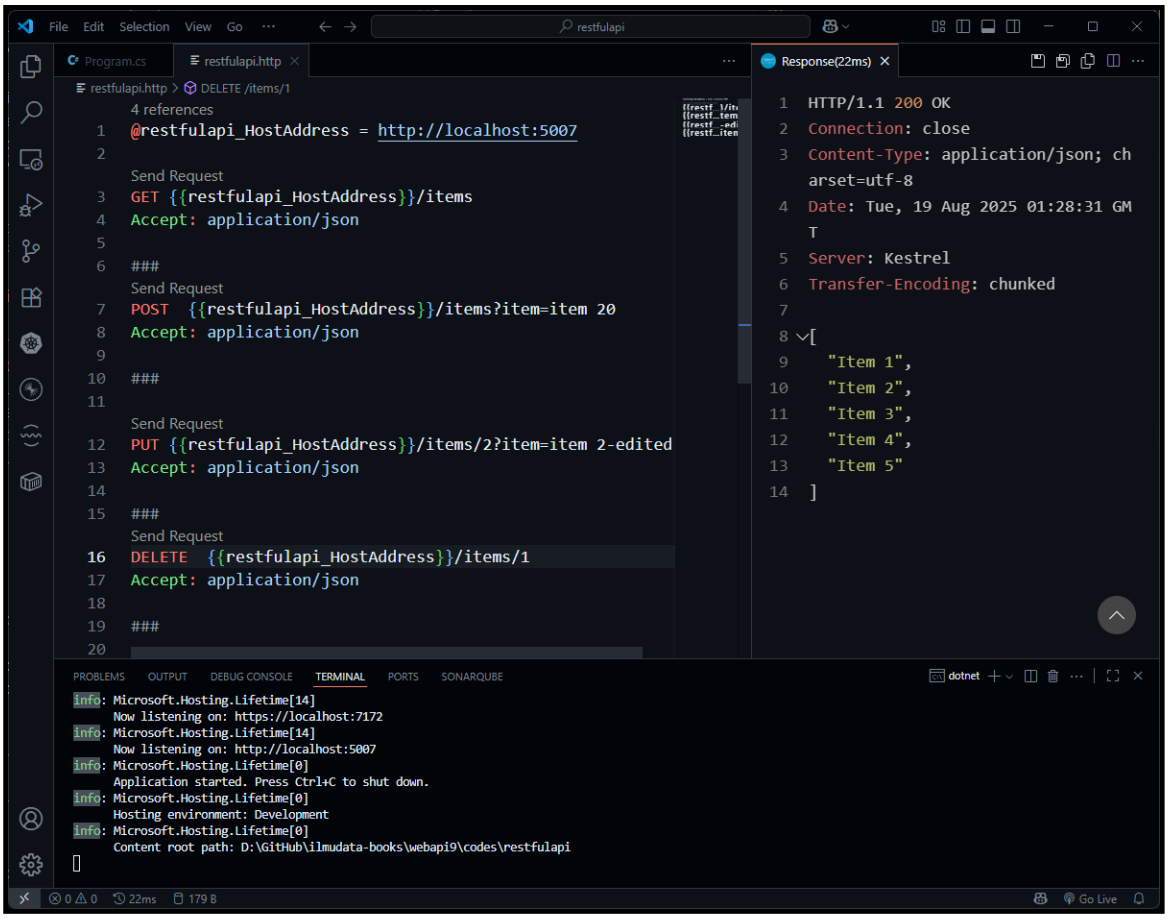


Figure 2.4 Test using http file for Restful API application.

2.3.4 Conclusion

In this lab, you have created a simple RESTful service using ASP.NET Core 9 Minimal API. This service demonstrates handling different types of HTTP requests and sending appropriate responses. You have also learned how to manage a simple in-memory data store and perform basic CRUD operations.

2.4 Exercise 3: OpenAPI Documentation

With the release of ASP.NET Core 9.0, creating robust and well-documented web APIs has become more streamlined than ever. One of the key changes in the `webapi` template is the transition from Swagger to built-in OpenAPI support. This integration provides developers with a modern, standardized approach to API documentation and testing.

Built-in OpenAPI Integration

- **Out-of-the-Box OpenAPI Support:** When you create a new web API project using the `webapi` template in ASP.NET Core 9.0, OpenAPI support is included by default. This provides an interactive documentation interface without any additional setup.

- **Microsoft.AspNetCore.OpenApi**: ASP.NET Core 9.0 uses the built-in OpenAPI package to generate API documentation. This package provides a set of tools for generating OpenAPI specifications based on your API code.

```
builder.Services.AddOpenApi();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
}
```

Explaining the Code

We can see the following code in the `Program.cs` file:

```
// Add services to the container.
builder.Services.AddOpenApi();
```

- **`AddOpenApi()`:** This method registers the OpenAPI services in the application's dependency injection container. The OpenAPI generator produces the OpenAPI specification for your API - a standardized, machine-readable representation of your API's structure and capabilities. It includes information about available endpoints, their parameters, response types, and other details necessary for documenting and interacting with the API.

```
if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
}
```

- **`if (app.Environment.IsDevelopment())`:** This conditional statement checks if the application is running in the development environment. It's a best practice to enable OpenAPI documentation only in development, as exposing your API's structure in production environments can lead to security risks.

- `app.MapOpenApi();`: This method maps the OpenAPI specification endpoint to your application. By default, it serves the generated OpenAPI specification as a JSON document at `/openapi/v1.json`. This JSON file contains the complete API specification that can be consumed by various tools and documentation generators.

Accessing OpenAPI Documentation

- **OpenAPI Specification:** Once your ASP.NET Core web API is running, you can access the OpenAPI specification by navigating to the `/openapi/v1.json` endpoint in your web browser. For example, if your API is hosted at `http://localhost:5000`, you can view the OpenAPI specification at `http://localhost:5000/openapi/v1.json`.

- **Scalar UI (Optional):** While OpenAPI specification provides the raw JSON format, you can enhance the developer experience by adding Scalar UI for a more interactive documentation interface. To add Scalar UI, install the package:

  ```
  dotnet add package Scalar.AspNetCore --version 2.6.9
  ```

  Then modify your `Program.cs`:

  ```
  builder.Services.AddOpenApi();

  var app = builder.Build();

  if (app.Environment.IsDevelopment())
  {
      app.MapOpenApi();
      app.MapScalarApiReference();
  }
  ```

Benefits of Built-in OpenAPI Support

- **Standard Compliance:** OpenAPI is an industry-standard specification for describing REST APIs, ensuring better interoperability with various tools and platforms.
- **Lightweight:** The built-in OpenAPI support is more lightweight compared to the previous Swagger implementation.
- **Modern Tooling:** OpenAPI specifications can be consumed by various modern tools for code generation, testing, and documentation.
- **Better Performance:** The native implementation provides better performance and reduced dependencies.

Customizing OpenAPI Documentation

- **Modifying OpenAPI Settings:** You can customize the OpenAPI documentation by configuring options such as API title, version, description, and adding custom metadata.
- **Adding XML Comments:** You can include XML documentation comments in your code to provide more detailed descriptions in the generated OpenAPI specification.

Demo

We can use the previous `restfulapi` project to demonstrate the built-in OpenAPI support in ASP.NET Core 9.0. First, let's update the project to use OpenAPI:

1. **Update the `Program.cs` file:**

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

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

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
}

// In-memory data store
var items = new List<string>();
for (int i = 1; i <= 5; i++)
{
   items.Add($"Item {i}");
}

// GET endpoint
app.MapGet("/items", () => items);

// POST endpoint
app.MapPost("/items", (string item) =>
{
    items.Add(item);
    return Results.Created($"/items/{items.Count - 1}", item);
});

// PUT endpoint
app.MapPut("/items/{id}", (int id, string item) =>
{
    if (id < 0 || id >= items.Count)
    {
        return Results.NotFound();
    }
```

```
    items[id] = item;
    return Results.NoContent();
});

// DELETE endpoint
app.MapDelete("/items/{id}", (int id) =>
{
    if (id < 0 || id >= items.Count)
    {
        return Results.NotFound();
    }
    items.RemoveAt(id);
    return Results.Ok();
});

app.Run();
```

> *You don't need to add the `builder.Services.AddOpenApi()` line if you are using the `webapi` template, as it is already included by default.*

2. **Run the project:**

```
dotnet run --launch-profile https
```

3. **Access the OpenAPI specification:**

   - Open a web browser and navigate to `https://localhost:<port>/openapi/v1.json`.
   - You should see the OpenAPI specification in JSON format, which provides a complete description of your API's endpoints and models.

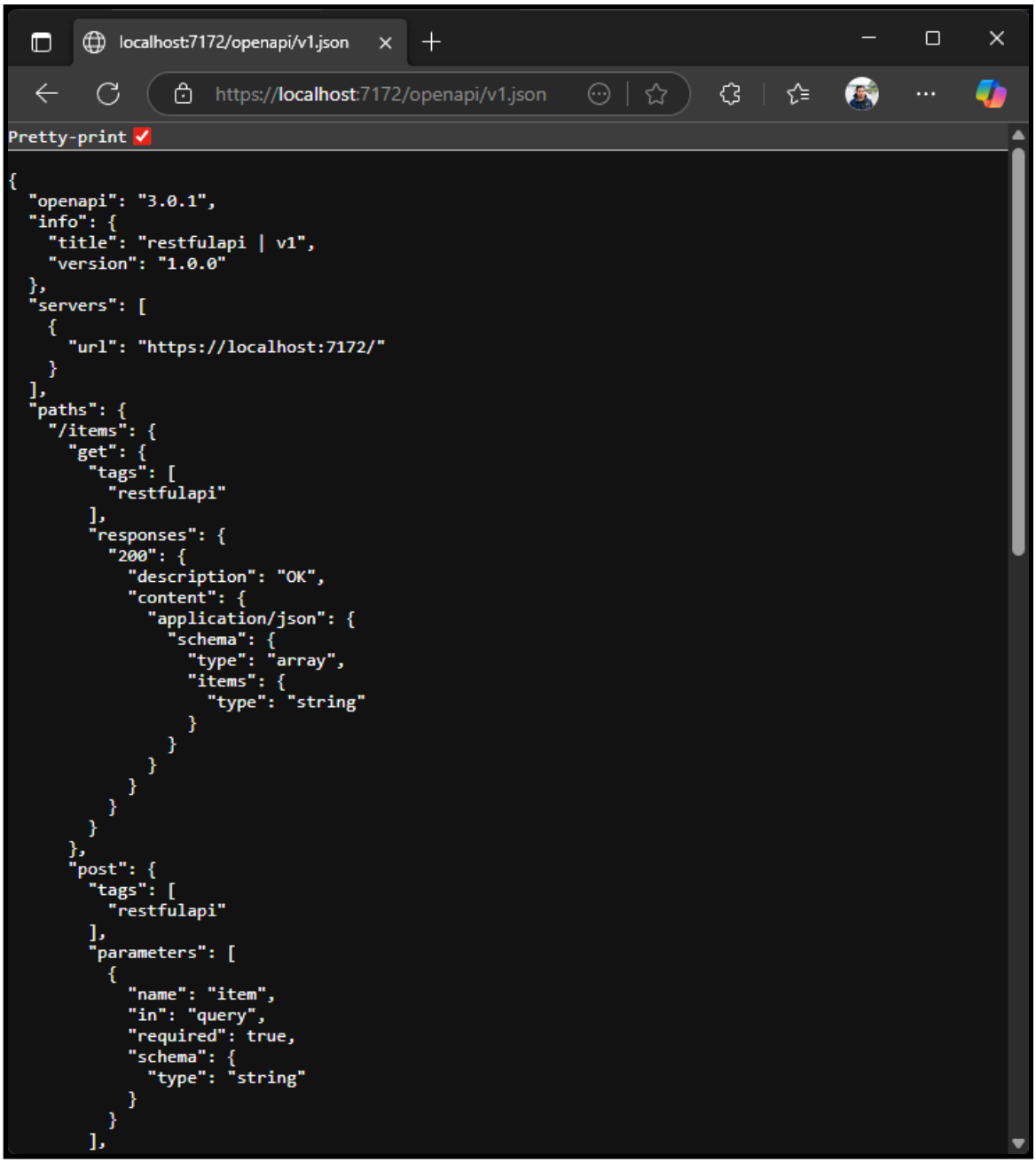


Figure 2.5 OpenAPI specification JSON for `restfulapi` project.

4. **Optional: Add Scalar UI for better visualization:**

   - Install Scalar UI package:

     ```
     dotnet add package Scalar.AspNetCore --version 2.6.9
     ```

   - Update your `Program.cs` to include Scalar UI:

```
if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
    app.MapScalarApiReference();
}
```

- Navigate to `https://localhost:<port>/scalar/` to view the interactive API documentation.

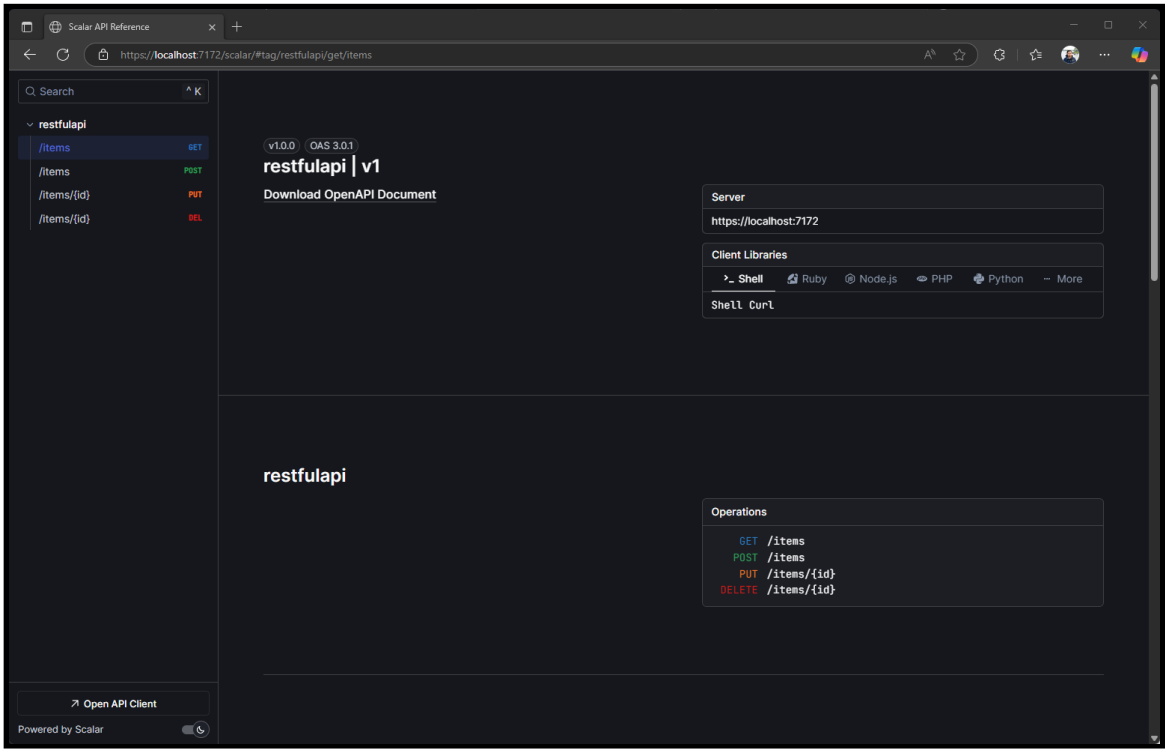


Figure 2.6 Scalar UI interface showing API documentation.

5. **Testing the API:**

- You can use the OpenAPI specification with tools like Postman, Insomnia, or any OpenAPI-compatible client.
- Import the OpenAPI specification URL (`https://localhost:<port>/openapi/v1.json`) into your preferred API testing tool.

Since we have used Scalar UI, you can also test the API directly from the Scalar UI interface. Click on the endpoints to see their details, and you can even execute requests directly from the documentation interface.

Click `/itens` endpoint to see the details of the GET request. You can also click on the **Test Request** button to execute the request directly from the documentation interface.

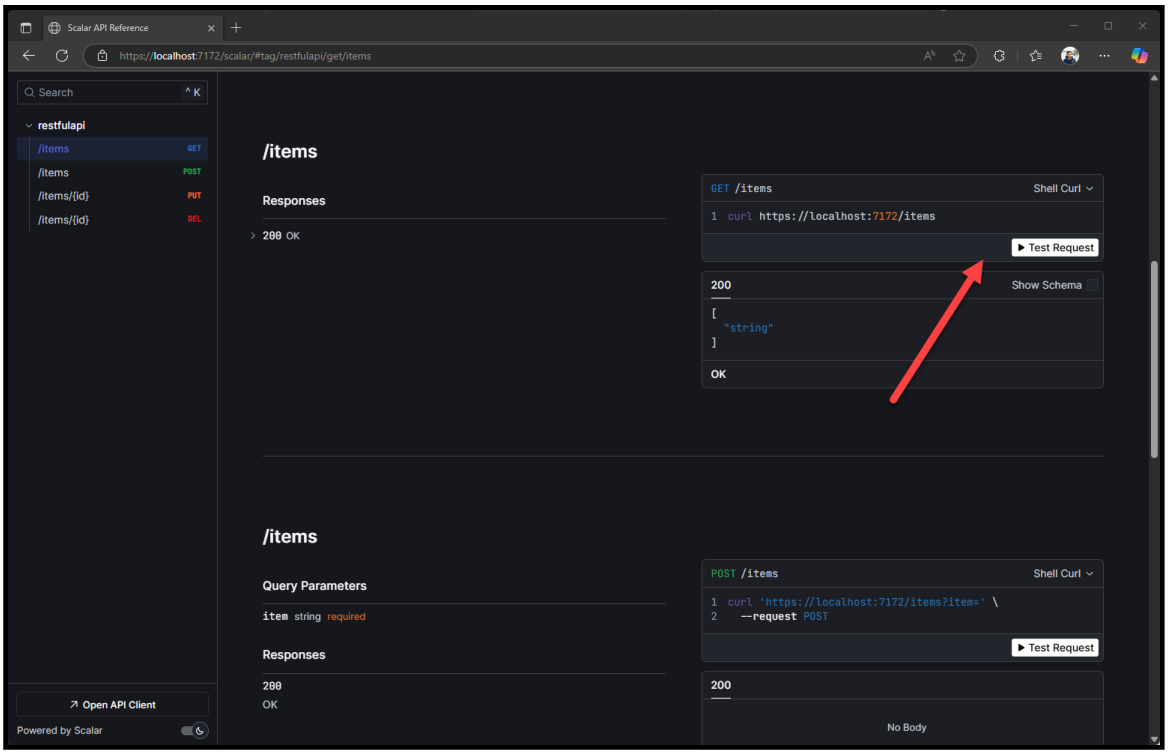


Figure 2.7 Testing API using Scalar UI.

You will have a form like Figure 2.8. Click the **Send** button to execute the request. You should see a response displaying the list of items.

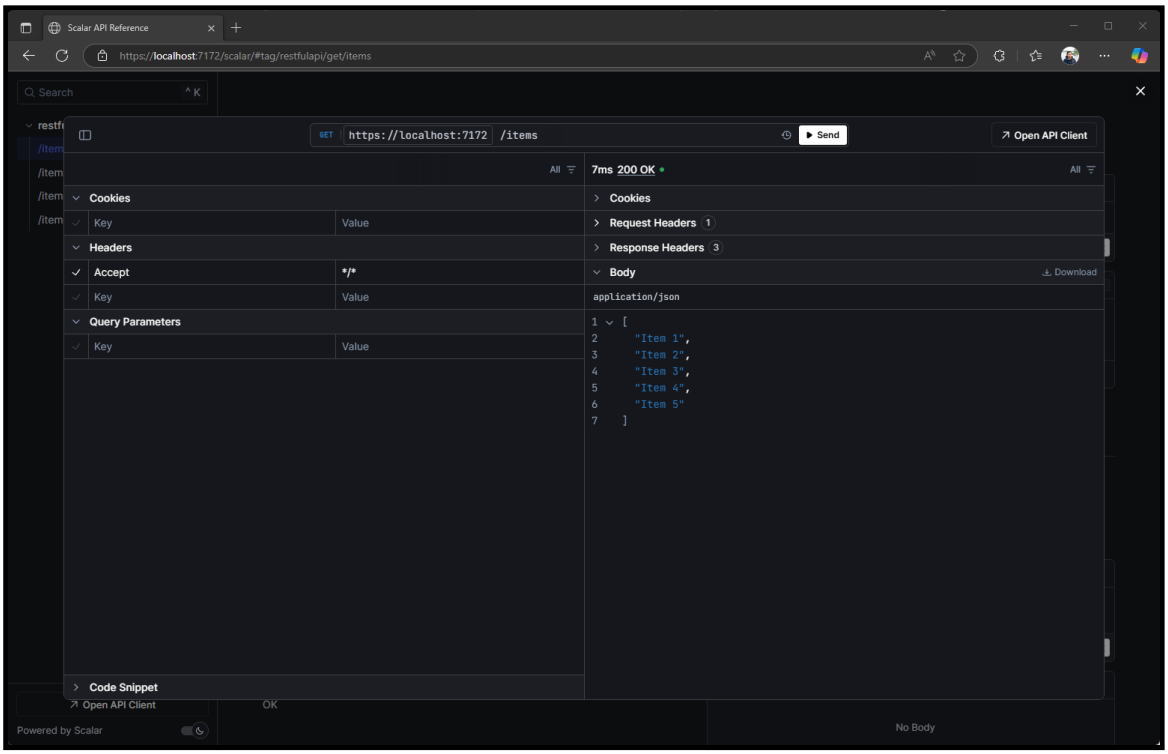


Figure 2.8 Make /items request from Scalar UI.

6. **OpenAPI and Postman**

If you prefer to use Postman, you can import the OpenAPI specification directly into Postman:

- Open Postman and click on the **Import** button.
- Select the **Link** tab and paste the OpenAPI specification URL (`https://localhost:<port>/openapi/v1.json`).
- Click **Continue** and then **Import** to load the API endpoints into Postman.
- You can now test the API endpoints directly from Postman, using the imported OpenAPI specification to guide you through the available operations.
- If you got failed to import the OpenAPI specification, you can also export the OpenAPI specification as a JSON file and import it into Postman.

2.4.1 Conclusion

The transition to built-in OpenAPI support in ASP.NET Core 9.0's `webapi` template represents a significant improvement in API documentation and tooling. This change provides developers with a more standardized, lightweight, and performant approach to API documentation. The OpenAPI specification ensures better compatibility with modern development tools and provides a solid foundation for API-first development practices. Whether you are building a new API or migrating existing services to ASP.NET Core 9.0, the integrated OpenAPI support offers enhanced developer experience and better industry standard compliance.

2.5 Exercise 4: Buidling a Calculator Service

In this lab, you'll create a basic calculator web service using ASP.NET Core 9 Minimal API. You'll learn how to handle different types of HTTP requests and perform basic arithmetic operations, gaining a foundation in this innovative approach to .NET development.

2.5.1 Objective

Develop a basic calculator web service using ASP.NET Core 9 Minimal API that supports basic arithmetic operations.

2.5.2 Requirements

- .NET 9.0 SDK installed
- A preferred code editor (e.g., Visual Studio, Visual Studio Code)

2.5.3 Lab Steps

1. Create a New ASP.NET Core Minimal API Project

- Open a command prompt or terminal.
- Run the following command to create a new ASP.NET Core Minimal API project:

```
mkdir calculatorapi
cd calculatorapi
dotnet new webapi
code .
```

2. Define the Numeric Record

- Open the `Program.cs` file in your code editor.
- Define the `Numeric` record at the last of the file:

```
record Numeric(double Number1, double Number2, double Result = 0);
```

3. Implement the Calculator Endpoints Using Numeric

- Modify the existing calculator endpoints to use the `Numeric` record. For addition, subtraction, multiplication, and division, the input and output are represented by the `Numeric` record.
- Open the `Program.cs` file in your code editor.
- Add endpoints for the four addition arithmetic operation:

```
...

var calculatorApi = app.MapGroup("/api/calculator");
calculatorApi.MapPost("/add", (Numeric numbers) =>
{
    var result = numbers.Number1 + numbers.Number2;
    return Results.Ok(new Numeric(numbers.Number1, numbers.Number2, result));
});
...
```

- Add the following code snippet for subtraction:

```
calculatorApi.MapPost("/subtract", (Numeric numbers) =>
{
    var result = numbers.Number1 - numbers.Number2;
    return Results.Ok(new Numeric(numbers.Number1, numbers.Number2, result));
});
```

- Add the following code snippet for multiplication:

```
calculatorApi.MapPost("/multiply", (Numeric numbers) =>
{
    var result = numbers.Number1 * numbers.Number2;
    return Results.Ok(new Numeric(numbers.Number1, numbers.Number2, result));
});
```

- Add the following code snippet for division:

```
calculatorApi.MapPost("/divide", (Numeric numbers) =>
{
    if (numbers.Number2 == 0)
    {
        return Results.BadRequest("Cannot divide by zero");
    }
    var result = numbers.Number1 / numbers.Number2;
    return Results.Ok( new Numeric(numbers.Number1, numbers.Number2, result));
});
```

  In this setup, each arithmetic operation takes a `Numeric` record as input, performs the calculation, and returns a new `Numeric` record with the result.

3. Build and Run the Application

- Use the command prompt or terminal to build and run the project:

```
dotnet run
```

- You can also run the application with https profile:

```
dotnet run --launch-profile https
```

4. Modify .http File for Testing

- Modify a content of calculatorapi.http file.

```
@calculatorapi_HostAddress = http://localhost:5003

###
POST {{calculatorapi_HostAddress}}/api/calculator/add
```

```
Accept: application/json
Content-Type: application/json

{
  "Number1": 10,
  "Number2": 5
}

###
POST {{calculatorapi_HostAddress}}/api/calculator/subtract
Accept: application/json
Content-Type: application/json

{
  "Number1": 10,
  "Number2": 5
}

###
POST {{calculatorapi_HostAddress}}/api/calculator/multiply
Accept: application/json
Content-Type: application/json

{
  "Number1": 10,
  "Number2": 5
}

###
POST {{calculatorapi_HostAddress}}/api/calculator/divide
Accept: application/json
Content-Type: application/json

{
  "Number1": 10,
  "Number2": 5
}
```

- Save the file.

4. Testing the Calculator Service

- Use tools like Postman to test your calculator service by sending POST requests.
- For each operation, send a JSON object in the request body, e.g., `{ "Number1": 5, "Number2": 3 }`, and specify the content type as `application/json`.
- Test the following endpoints:
  - **Addition:** `http://localhost:5003/api/calculator/add`
  - **Subtraction:** `http://localhost:5003/api/calculator/subtract`
  - **Multiplication:** `http://localhost:5003/api/calculator/multiply`
  - **Division:** `http://localhost:5003/api/calculator/divide`
- You can also test using the REST Client extension.

- Click the **Send Request** button to test the API.
- You should see a response displaying the result of the operation.

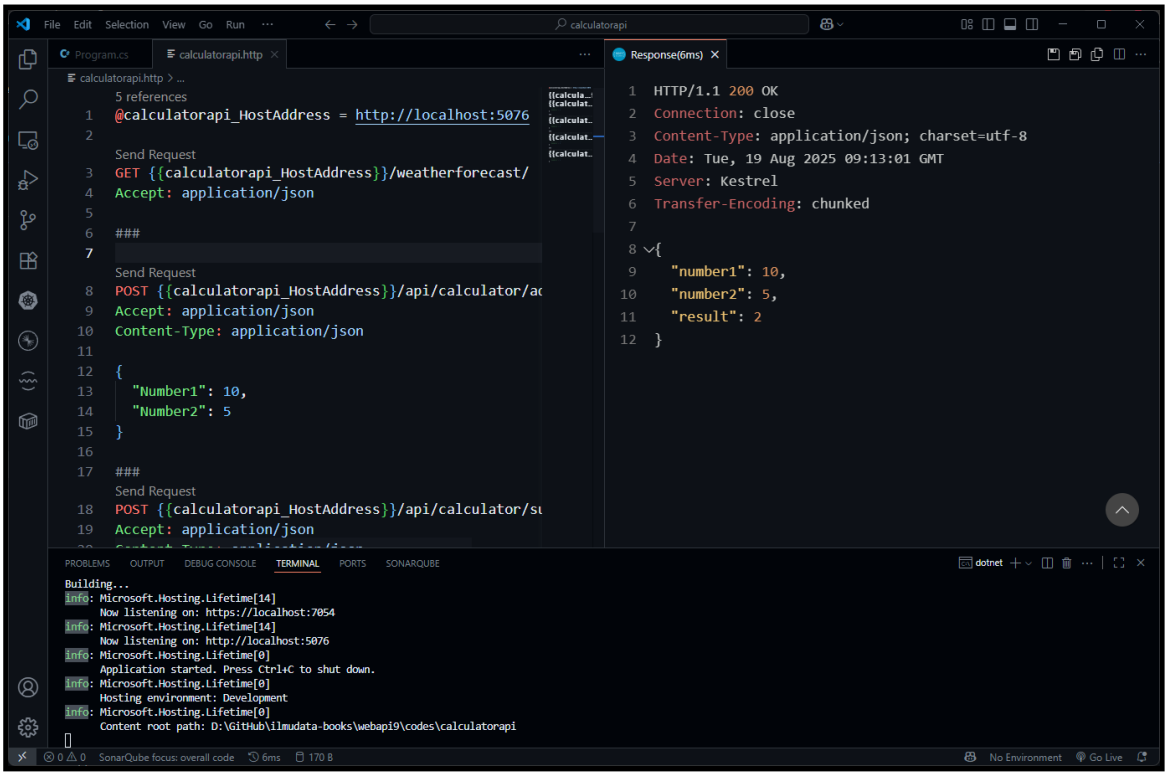


Figure 2.9 Response from server using REST Client extension.

2.5.4 Conclusion

In this lab, you have created a calculator web service using ASP.NET Core 9 Minimal API with a focus on using a record type (`Numeric`) for clean and efficient data handling. This approach demonstrates the power of records in simplifying parameter passing and result representation in web APIs.

2.6 Exercise 5: Upload and Download File Web

In this exercise, you'll learn how to implement file upload and download functionality using ASP.NET Core 9.0 Minimal API. Handling files is a common requirement in modern web applications, and this lab will guide you through setting up endpoints for uploading files to the server and downloading them from a designated directory. You'll gain practical experience with multipart form data, file streams, and serving static content, all within the context of a minimal API project.

2.6.1 Objective

Develop an ASP.NET Core 9 Web API that allows users to upload and download files. This lab demonstrates handling file streams and setting appropriate API endpoints.

2.6.2 Requirements

- .NET 9.0 SDK installed
- A preferred code editor (e.g., Visual Studio, Visual Studio Code)
- Basic knowledge of ASP.NET Core

2.6.3 Lab Steps

1. Create a New ASP.NET Core Minimal API Project

- Open a command prompt or terminal.
- Create a new project using the `webapi` template:

```
mkdir fileapi
cd fileapi
dotnet new webapi
```

- Open the project in your code editor:

```
code .
```

2. Create `uploads` folder

- Create a new folder named `uploads` in the root of web root directory `wwwroot`.
- If you don’t have `wwwroot` folder, create it inside the root of the project.
- The `wwwroot` folder is used to serve static files, such as HTML, CSS, JavaScript, and images, in ASP.NET Core Minimal API applications.
- You can see the project structure in your code editor:

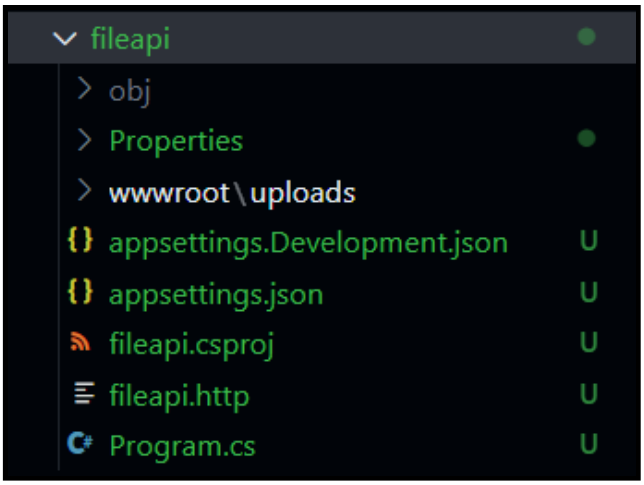


Figure 2.10 Project structure of the ASP.NET Core Minimal API project.

3. Disable Atiforgery

- Since we use `multipart/form-data` for file upload, we need to disable the antiforgery token validation.
- Open the `Program.cs` file in your code editor.

```
...
builder.Services.AddAntiforgery();

var app = builder.Build();
app.UseAntiforgery();

...
```

4. Create the API for File Upload to Use `uploads`

- Open the `Program.cs` file in your code editor.
- Import the necessary namespaces at the top of the file:

```
using Microsoft.AspNetCore.Mvc;
```

- Add a service for file upload, pointing to the `wwwroot/uploads` directory:

```
var fileApi = app.MapGroup("/api/file");
fileApi.MapPost("/upload", async (IFormFile file,[FromForm]string description) =>
{
    var uploadsFolderPath = Path.Combine( app.Environment.WebRootPath, "uploads");
    Directory.CreateDirectory(uploadsFolderPath);
    var filePath = Path.Combine( uploadsFolderPath, file.FileName);

    using (var stream = new FileStream(filePath, FileMode.Create))
    {
```

```
        await file.CopyToAsync(stream);
    }

    return Results.Ok(new { FilePath = $"/Uploads/{file.FileName}", Description = des
}).DisableAntiforgery();
```

- We use `.DisableAntiforgery()` to disable the antiforgery token validation for this endpoint.

5. Create the API for File Download to Use `wwwroot/uploads`

- Add the following code snippet for the file download endpoint, pointing to the `wwwroot/Uploads` directory:

```
fileApi.MapGet("/download/{fileName}", async (string fileName) =>
{
    var filePath = Path.Combine( app.Environment.WebRootPath, "Uploads", fileName);
    if (!File.Exists(filePath))
    {
        return Results.NotFound("File not found.");
    }

    var memoryStream = new MemoryStream();
    using (var stream = new FileStream(filePath, FileMode.Open, FileAccess.Read))
    {
        await stream.CopyToAsync(memoryStream);
    }
    memoryStream.Position = 0;
    return Results.File(memoryStream, "application/octet-stream", fileName);
});
```

6. Build and Run the Application

- In the terminal, build and run the project using `dotnet run`.

```
dotnet run
```

- We can also run the application with https profile:

```
dotnet run --launch-profile https
```

7. Modify .http File for Testing

- Open the `fileapi.http` file in your code editor.
- Write the following scripts

```
@fileapi_HostAddress = https://localhost:7010


### Upload File
```

```
POST {{fileapi_HostAddress}}/api/file/upload
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW
Accept: application/json

------WebKitFormBoundary7MA4YWxkTrZu0gW
Content-Disposition: form-data; name="description"

This is a test file
------WebKitFormBoundary7MA4YWxkTrZu0gW
Content-Disposition: form-data; name="file"; filename="example.txt"
Content-Type: text/plain

< ./example.txt

------WebKitFormBoundary7MA4YWxkTrZu0gW--

### Download File
GET {{fileapi_HostAddress}}/api/file/download/example.txt
```

- Save the file.
- We also create a file named `example.txt` in the root of the project.
- Just write some text in the file.

8. Testing the API

- Use a tool like Postman or a similar HTTP client to test the file upload and download functionality.
- If you use Postman, you follow these steps:
    - Import `https://localhost:7201/openapi/v1.json` to Postman to get the API endpoints.
    - Create a new request in Postman or open a new tab.
    - Set the method to **POST**.
    - Enter the URL of your file upload endpoint, which will be something like `https://localhost:7201/api/file/upload`. Change this URL to match your API's URL.
    - In the 'Headers' section, you don't need to explicitly set the `Content-Type` to `multipart/form-data` as Postman will automatically add it when you select a file to upload.
    - Go to the **Body** tab in Postman.
    - Select **form-data**.
    - In the key field, select **File** from the dropdown.
    - Click on the **Select Files** button or simply drag and drop the file into the key-value area to attach the file you want to upload.
    - Add key `description` and value `This is a test file`.

- Click the **Send** button to execute the request.
- You should receive a response indicating that the file upload was successful.

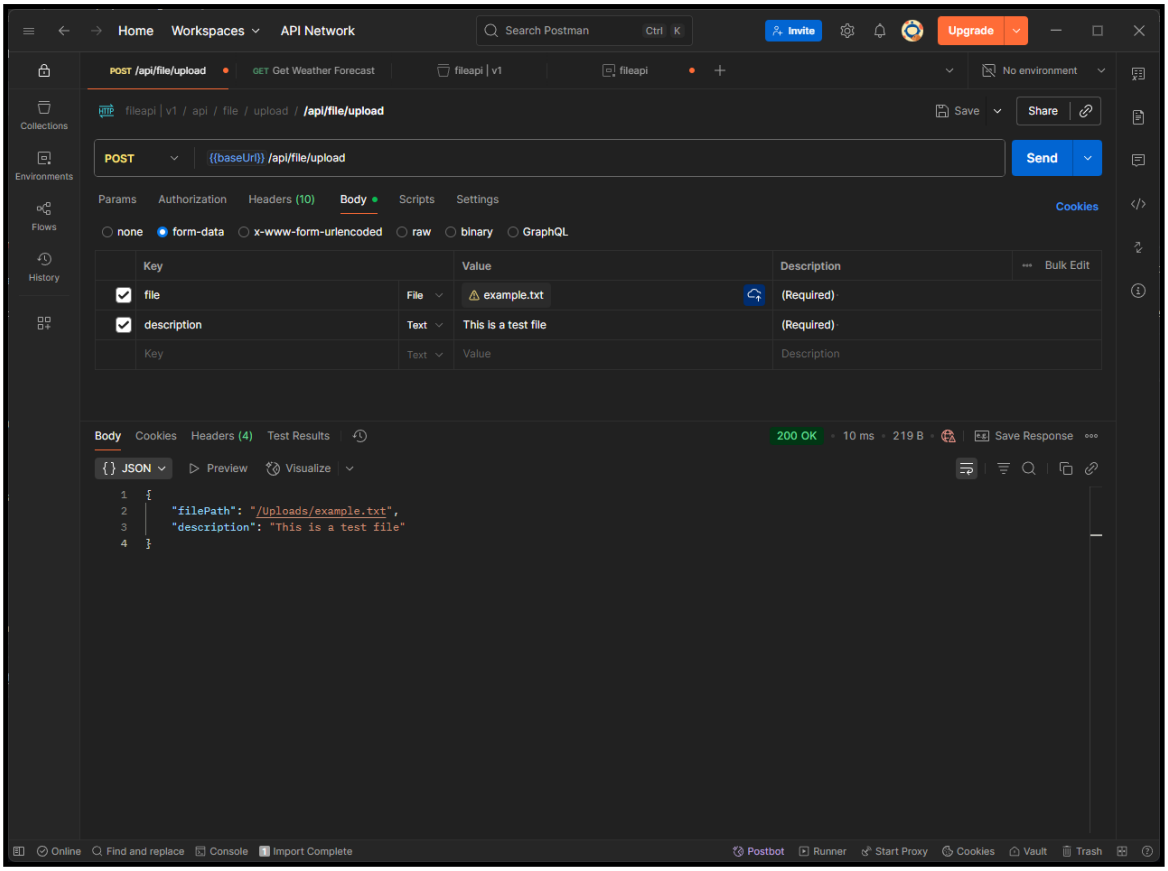


Figure 2.11 Response from server.

- You can also test using the REST Client extension.
- Click the **Send Request** button to test the API.

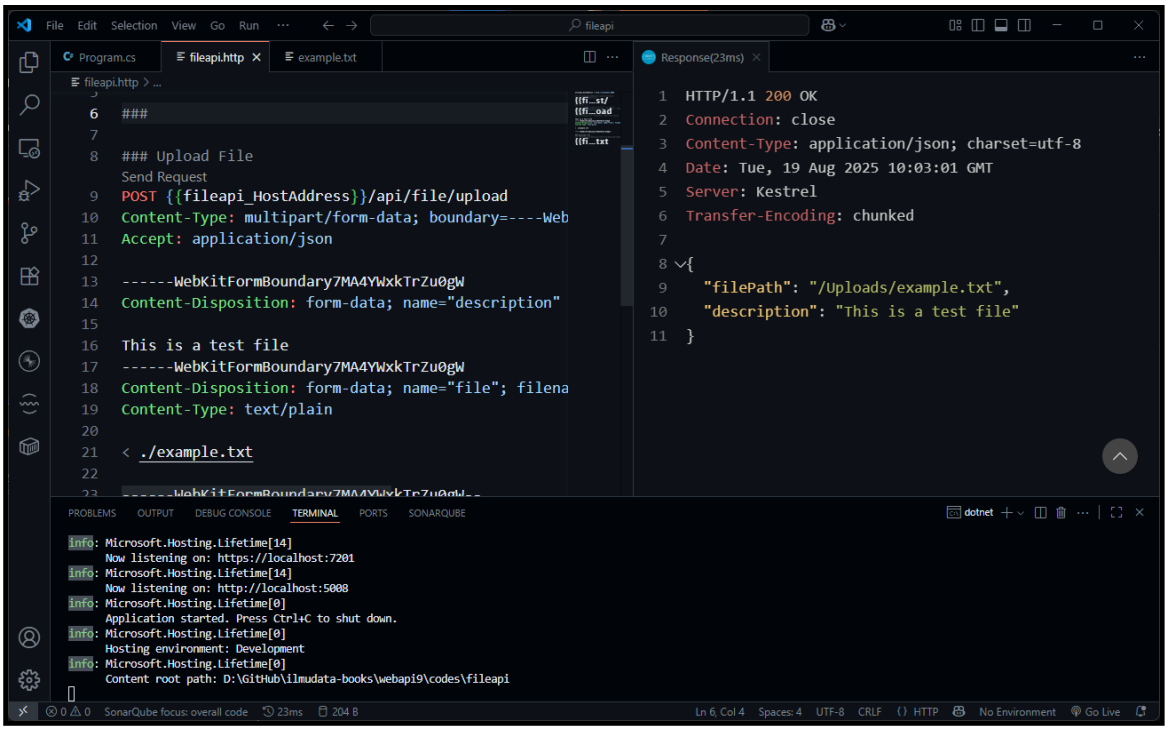


Figure 2.12 Response from server using REST Client extension.

2.6.4 Conclusion

In this lab, you've set up a minimal API for file uploading and downloading in ASP.NET Core 9, utilizing the `wwwroot` directory for storing files. This approach is typical in web applications for managing static content, including files uploaded by users.

2.7 Exercise 6: Exception Handling and Logging

In this lab, you'll enhance your ASP.NET Core 9.0 Minimal API project to handle various HTTP errors and implement logging to a file. Exception handling and logging are crucial for building robust APIs that can effectively communicate errors to clients and maintain a log for troubleshooting and monitoring.

2.7.1 Objective

Enhance the ASP.NET Core 9.0 Minimal API project to handle various HTTP errors (500, 502, 404, 400) and implement logging to a file.

2.7.2 Requirements

- .NET 9.0 SDK installed
- Visual Studio Code or another code editor
- REST Client extension installed in Visual Studio Code for testing
- A package for logging to a file, such as `Serilog`

2.7.3 Lab Steps

1. Set Up the Project and Serilog

- Create a new project (if not already created) and navigate to the project directory as previously described.

```
mkdir exceptionhandlingapi
cd exceptionhandlingapi
dotnet new webapi
```

- Open the project in your code editor.

```
code .
```

- Install Serilog via NuGet:

```
dotnet add package Serilog
dotnet add package Serilog.AspNetCore
dotnet add package Serilog.Extensions.Logging
dotnet add package Serilog.Sinks.File
dotnet add package Serilog.Sinks.Console
```

- Configure Serilog in `Program.cs`:

```
using Microsoft.AspNetCore.Diagnostics;
using Microsoft.AspNetCore.Mvc;
using Serilog;



var builder = WebApplication.CreateBuilder(args);
builder.Logging.ClearProviders();

...

// Configure Serilog
var logger = new LoggerConfiguration()
  .WriteTo.Console()
  .WriteTo.File("log-.txt", rollingInterval: RollingInterval.Day)
  .CreateLogger();
// Register Serilog
builder.Logging.AddSerilog(logger);

// Other configurations...
```

2. Implement Custom Error Handling and Responses

- Modify the `Program.cs` to handle different types of errors:

```
var app = builder.Build();

app.UseExceptionHandler("/error");

app.MapGet("/error", (HttpContext httpContext) =>
{
    var exceptionFeature = httpContext.Features.Get< IExceptionHandlerFeature>();
    var exception = exceptionFeature?.Error;

    var problemDetails = new ProblemDetails
    {
        Status = 500,
        Title = "An error occurred while processing your request."
    };

    if (exception is FileNotFoundException)
    {
        problemDetails.Status = 404;
        problemDetails.Title = "File not found.";
    }
    else if (exception is InvalidOperationException)
    {
        problemDetails.Status = 400;
        problemDetails.Title = "Invalid operation.";
    }

    // Logging the exception
    app.Logger.LogError( exception, "An error occurred: {ErrorMessage}", exception.Me

    return Results.Problem( problemDetails.Title, statusCode: problemDetails.Status);
});

// Define other routes...
```

3. Create Endpoints to Simulate Errors

- Add endpoints to simulate different errors:

```
app.MapGet("/causeinternalerror", () =>
{
    throw new Exception( "Internal server error." );
});

app.MapGet("/causefileerror", () =>
{
    throw new FileNotFoundException( "Example file not found." );
});

app.MapGet("/causeinvalidoperation", () =>
{
    throw new InvalidOperationException( "Invalid operation example." );
```

```
});

// Other routes...
```

4. Build and Run the Application

- Build and run your project using `dotnet run`.
- You can also run the application with https profile:

```
dotnet run --launch-profile https
```

5. Create the `.http` File for Testing Different Errors

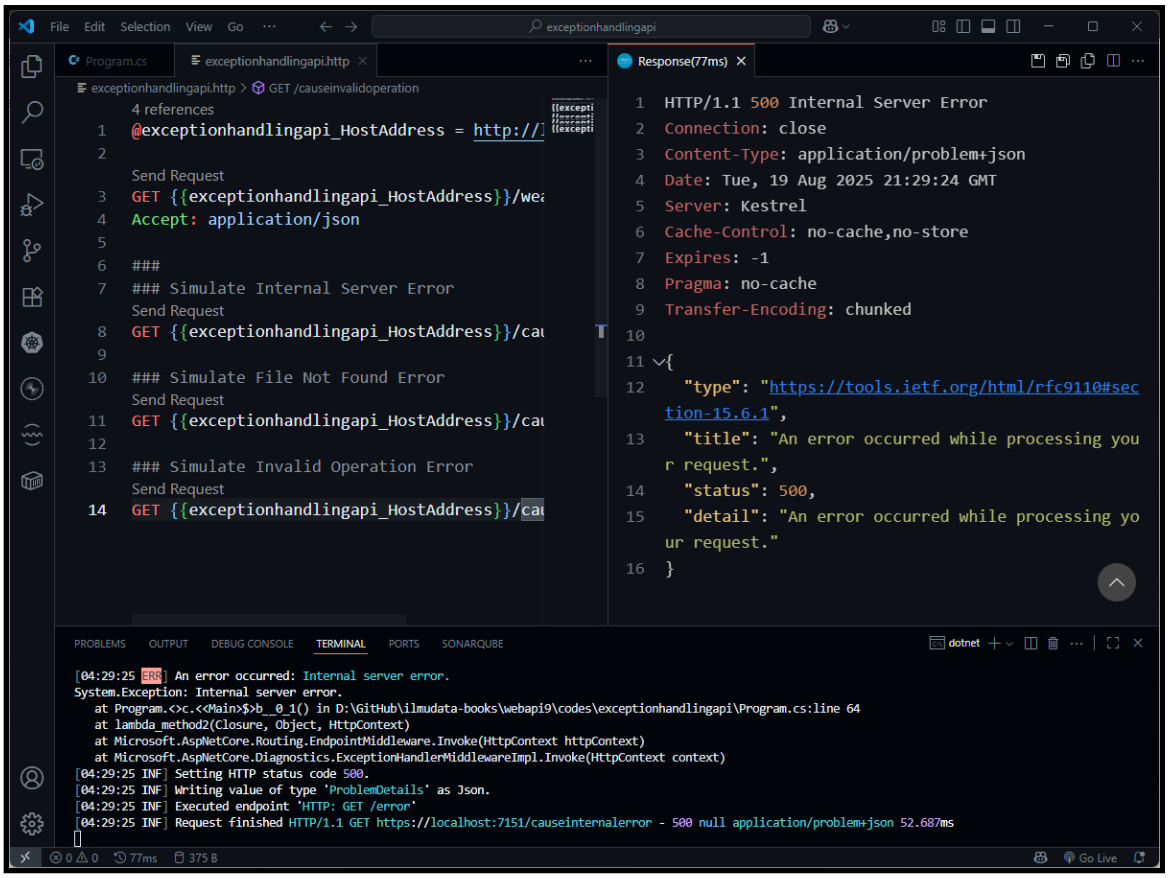
- Create `ExceptionHandlingApi.http` with the following content:

```
@exceptionhandlingapi_HostAddress = http://localhost:5100

GET {{exceptionhandlingapi_HostAddress}}/weatherforecast
Accept: application/json

### Simulate Internal Server Error
GET {{exceptionhandlingapi_HostAddress}}/causeinternalerror

### Simulate File Not Found Error
GET {{exceptionhandlingapi_HostAddress}}/causefileerror

### Simulate Invalid Operation Error
GET {{exceptionhandlingapi_HostAddress}}/causeinvalidoperation
```

- Change port `5100` to the port number displayed in the command prompt or terminal.

6. Testing Error Handling and Logging

- Use the REST Client extension in Visual Studio Code to test each endpoint.
- Click the **Send Request** button to test the API.
- You should see a response displaying the error message.



- Check the `log-xxxxxx` file in your project directory for logged error details.

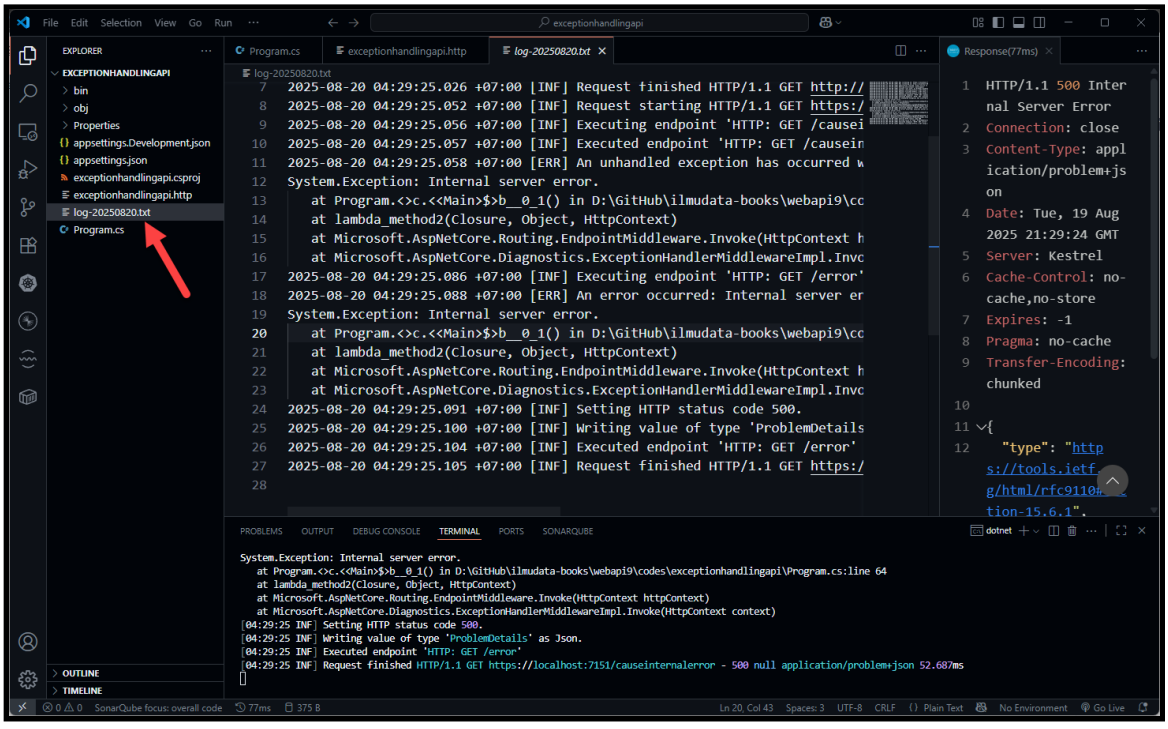


Figure 2.13 Log file.

2.7.4 Conclusion

In this modified lab, you have enhanced an ASP.NET Core 9.0 Minimal API application to handle different HTTP errors and log exceptions to a file. This approach is vital for building robust APIs that can effectively communicate errors to clients and maintain a log for troubleshooting and monitoring.

2.8 Exercise 7: Middleware and Filters

In this lab, you'll learn how to implement custom middleware and filters in ASP.NET Core 9.0 Minimal API. You'll explore the use of middleware for request processing and handling, as well as the use of filters for request filtering and response formatting.

A middleware is a component that is executed on every request in the ASP.NET Core pipeline. It can be used to perform actions such as logging, authentication, and error handling. Filters are used to apply cross-cutting concerns to specific endpoints or controllers, such as authorization, caching, and response formatting.

2.8.1 Objective

Implement custom middleware and demonstrate the use of filters in an ASP.NET Core 9.0 Minimal API project.

2.8.2 Requirements

- .NET 9.0 SDK installed
- Visual Studio Code or another code editor
- REST Client extension installed in Visual Studio Code for testing

2.8.3 Lab Steps

1. Create a New ASP.NET Core Minimal API Project

- Open a command prompt or terminal.
- Create a new ASP.NET Core Minimal API project:

```
mkdir middlewareapi
cd middlewareapi
dotnet new webapi
```

- Open the project in your code editor:

```
code .
```

2. Implement Custom Middleware

- Open the project in your code editor.
- Modify the `Program.cs` to include a simple logging middleware:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

// Custom Middleware for Logging
app.Use(async (context, next) =>
{
    Console.WriteLine("Request Incoming");
    await next();
    Console.WriteLine("Response Outgoing");
});

// Define routes
app.MapGet("/", () => "Hello World");

app.Run();
```

- We can also create a middleware class.
- Create a new file named `LoggingMiddleware.cs` in the root of the project.
- Add the following code snippet:

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

public class LoggingMiddleware
{
    private readonly RequestDelegate _next;

    public LoggingMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        Console.WriteLine( "LoggingMiddleware>> Request Incoming" );
        await _next(context);
        Console.WriteLine( "LoggingMiddleware>> Response Outgoing" );
```

```
    }
}
```

- In the `Program.cs` file, modify the `Main` method to use the middleware. Write these after `builder.Build()`:

```
...

// Custom Middleware for Logging
app.UseMiddleware<LoggingMiddleware>();

...
```

3. Implement a Simple Filter (Optional in Minimal API)

- In Minimal APIs, filters as known in MVC aren't directly available. However, you can achieve similar functionality using middleware or service injection in endpoint delegates.
- For simplicity, we'll focus on middleware in this lab. (For advanced scenarios, consider converting to a Controller-based approach to use filters.)

4. Build and Run the Application

- Build and run the project using:

```
dotnet run
```

- You can also run the application with https profile:

```
dotnet run --launch-profile https
```

5. Create the `.http` File for Testing

- Create `middlewareapi.http` with the following content to test the middleware:

```
@middlewareapi_HostAddress = http://localhost:5104

GET {{middlewareapi_HostAddress}}/weatherforecast
Accept: application/json

###
```

- Change port `5104` to the port number displayed in the command prompt or terminal.

6. Testing the Middleware

- Open `middlewareapi.http` in Visual Studio Code.

- Use the REST Client extension to send the request.
- Observe the console output to see the middleware in action.
- You should see the following output in the console.

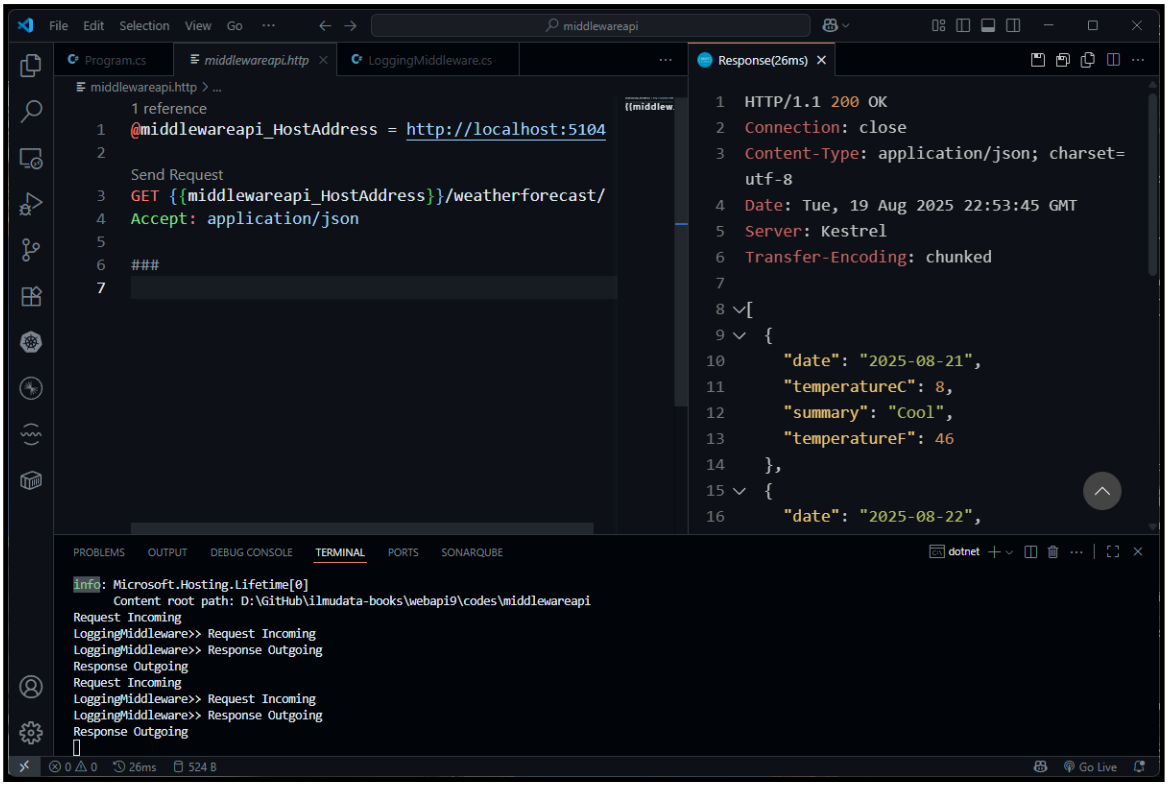


Figure 2.14 Response from server using REST Client extension.

- Another way to test the middleware is to use the Swagger UI.
- Open a web browser and navigate to `https://localhost:5062/swagger`.
- Change port `5062` to the port number displayed in the command prompt or terminal.

2.8.4 Conclusion

In this lab, you've implemented a custom middleware in an ASP.NET Core 9.0 Minimal API application. This middleware logs messages to the console before and after processing HTTP requests. While Minimal APIs don't support filters in the same way as MVC, middleware offers a powerful alternative for request processing and handling.

3 Accessing SQL and NoSQL Databases

3.1 Introduction

Welcome to the comprehensive guide on accessing SQL and NoSQL databases in .NET 9.0. In today's data-driven world, the ability to efficiently interact with various types of databases is essential for any application. This chapter is crafted to equip you with the necessary skills and knowledge to seamlessly integrate and manipulate data stored in both SQL and NoSQL databases using the latest .NET technologies.

Understanding the Database Landscape

- **SQL Databases:** We start by exploring SQL databases, the traditional and widely used systems for managing structured data. Here, you'll learn about establishing connections, executing SQL queries, and handling transactions using .NET's rich set of libraries and tools.
- **NoSQL Databases:** Transitioning from the structured world of SQL, we delve into the realm of NoSQL databases. These systems offer flexibility and scalability for handling unstructured data, and we'll cover how to interact with popular NoSQL databases using .NET.

Key Concepts and Technologies

- **Entity Framework Core:** A significant part of this chapter will focus on Entity Framework Core, .NET's flagship ORM, which simplifies data access in both SQL and NoSQL databases.
- **ADO.NET:** For SQL databases, ADO.NET remains a cornerstone, and we'll revisit its powerful features and how they integrate into modern .NET applications.
- **NoSQL Client Libraries:** We'll also explore various client libraries available for NoSQL databases, discussing how to choose and work with them effectively in .NET.

Practical Examples and Best Practices

- **Hands-On Examples:** The chapter will be rich in practical examples, demonstrating CRUD operations, complex queries, and performance optimization techniques in real-world scenarios.
- **Best Practices:** As we navigate through different database technologies, we'll emphasize best practices in database design, query optimization, and maintaining data integrity and security.

Preparing for the Hands-On Labs

The following sections and hands-on labs are designed to be interactive and incremental, ensuring a deep and practical understanding of each topic. Whether you are developing a new application or maintaining an existing one, this chapter will provide you with the tools and insights needed to work effectively with SQL and NoSQL databases in .NET 9.0.

By the end of this chapter, you will be well-equipped to make informed decisions about database technologies and implement robust data access solutions in your .NET applications. Let's embark on this journey to master the art of database interaction in the modern .NET ecosystem.

3.2 .NET Entity Framework Core

Entity Framework Core 9.0 is the latest version of Microsoft's Object-Relational Mapping (ORM) framework, providing a powerful and flexible way to interact with databases in .NET applications. EF Core supports two primary development approaches: Code First and Database First. Each has its unique advantages and use cases.

1. Code First Approach

- **Overview:** In the Code First approach, developers define database models and relationships using C# classes. EF Core then generates the database schema based on these models. This approach is popular in scenarios where the database schema is initially unknown or subject to frequent changes.

- **Getting Started:**

    - **Define Models:** Create C# classes to represent entities. Each class typically corresponds to a table in the database.
    - **Define DbContext:** Create a class that derives from `DbContext`. This class acts as a session with the database, allowing querying and saving data.

    - **Configuration:** Use Data Annotations or Fluent API within the `DbContext` to configure models, relationships, keys, and other constraints.

- **Migrations:** EF Core migrations track changes to the model and update the database schema accordingly. Use commands like `dotnet ef migrations add` and `dotnet ef database update` to manage database changes.

- **Benefits:**

    - Full control over the database schema through code.
    - Seamless integration with version control systems.
    - Ideal for Agile development and rapid prototyping.

2. Database First Approach

- **Overview:** The Database First approach starts with an existing database. Developers use EF Core tools to generate C# classes that map to the database tables, views, and stored procedures. This approach is suitable when working with a legacy database or a database designed by a separate team.

- **Getting Started:**

    - **Scaffold DbContext:** Use the EF Core command-line tools to scaffold a `DbContext` and entity classes from the existing database using the `dotnet ef dbcontext scaffold` command.
    - **Customize Models (if needed):** Modify the generated classes to better fit your application's needs, though be cautious as customizations might be overwritten if scaffolding is repeated.

- **Maintaining Synchronization:** Keep the entity classes and the database schema in sync. Changes to the database require re-scaffolding and potential manual adjustments to the entity classes.

- **Benefits:**

    - Quick setup for applications built around existing databases.
    - Reduces the need for manual coding of the data access layer.
    - Ideal for projects where database schema is managed by a separate database administration team or tool.

Best Practices and Considerations

- **Version Control:** Migrations in the Code First approach should be committed to version control to track changes over time.
- **Database Updates:** In the Database First approach, be cautious about database schema changes as they might require re-scaffolding and code adjustments.
- **Performance:** Regardless of the approach, pay attention to performance implications of your design choices, like lazy loading, eager loading, and the efficiency of generated SQL queries.

Entity Framework Core 9.0 provides robust support for both Code First and Database First approaches, catering to different project requirements and stages. Understanding the strengths and limitations of each approach is key to effectively managing your application's data layer. Whether you are starting from scratch with a new database design or integrating with an existing database, EF Core offers the tools and flexibility needed for modern .NET data access.

3.3 Entity Framework Core tools

Entity Framework Core (EF Core) tools are a set of command-line tools that simplify the development and maintenance of EF Core applications. These tools are available as a .NET global tool, which can be installed using the following command:

```
dotnet tool install --global dotnet-ef
```

Once installed, you can use the `dotnet ef` command to run EF Core commands. For example, to list all available commands, use:

```
dotnet ef -h
```

To update the EF Core tools to the latest version, use:

```
dotnet tool update --global dotnet-ef
```

We'll use EF Core tools throughout this chapter to scaffold database models, generate migrations, and perform other tasks.

3.4 Exercise 8: EF Core 9.0 Code First and ASP.NET Core Minimal API

In this exercise, you'll learn how to use Entity Framework Core 9.0 (EF Core) in an ASP.NET Core Minimal API project. You'll create a simple RESTful service that performs CRUD operations on a SQL Server database. You'll also learn how to use EF Core migrations to create and update the database schema.

In this lab, we use Code First approach to create the database schema.

3.4.1 Objective

Integrate Entity Framework Core 9.0 with ASP.NET Core Minimal API to create a simple RESTful service that performs CRUD operations on a database.

3.4.2 Requirements

- .NET 9.0 SDK installed
- Visual Studio Code or another code editor
- REST Client extension installed in Visual Studio Code for testing
- SQL Server or another EF Core compatible database

3.4.3 Lab Steps

1. Set Up the ASP.NET Core Minimal API Project

- Open a command prompt or terminal.
- Navigate to your desired working directory or create a new one.

```
mkdir efcoredb
cd efcoredb
```

- Create a new ASP.NET Core Minimal API project:

```
dotnet new webapi
```

- Open the project in Visual Studio Code:

```
code .
```

2. Install Entity Framework Core

- Install the necessary EF Core packages via NuGet. For SQL Server and In-MemoryDB, use:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.InMemory
dotnet add package Microsoft.EntityFrameworkCore.Design
```

3. Create a Model and DbContext

- In your project, create a new folder named `Models`.
- Inside `Models`, create a file `Product.cs` with the following content:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
namespace EFCoreDb.Models
{
    public class Product
    {
        [Key]
        [DatabaseGenerated( DatabaseGeneratedOption.Identity )]
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
}
```

- Inside `Models`, create a file `AppDbContext.cs`:

```
using Microsoft.EntityFrameworkCore;

namespace EFCoreDb.Models
{
    public class AppDbContext : DbContext
    {
        public AppDbContext(DbContextOptions<AppDbContext> options)
            : base(options)
        {
        }

        public DbSet<Product> Products { get; set; }
    }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        // Define precision for decimal property
        modelBuilder.Entity<Product>()
            .Property(product => product.Price)
            .HasPrecision(18, 2);
    }
}
```

4. Configure DbContext in Program.cs

- In `Program.cs`, configure EF Core:

```
using EFCoreDb.Models;
```

```
using EFCoreDb.Models;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddDbContext<AppDbContext>(options => {
  var useInMemoryDb = builder.Configuration.GetValue<bool>("UseInMemoryDatabase");
  if (useInMemoryDb)
  {
      options.UseInMemoryDatabase("TrainingDB");
  }
  else
  {
      options.UseSqlServer(builder.Configuration.GetConnectionString("MyDB"));
  }

});

var app = builder.Build();

// Configure the HTTP request pipeline.
// ...
```

- In `appsettings.json` and `appsettings.Development.json` files, add database configuration with the following content:

```
{
    "UseInMemoryDatabase": true,
    "ConnectionStrings": {
        "MyDB": "server=localhost; database=TrainingDB; uid=tester; pwd=pass123"
    }
}
```

- Change the connection string to match your database configuration.

5. Implement CRUD Operations

- In `Program.cs`, implement GET endpoints for `Product`:

```
app.MapGet("/products", async (AppDbContext dbContext) =>
  await dbContext.Products.ToListAsync());

app.MapGet("/products/{id}", async (AppDbContext dbContext, int id) =>
{
    var product = await dbContext.Products.Where(p => p.Id == id).FirstOrDefaultAsync
    return Results.Ok(product);
});

app.MapPost("/products", async (AppDbContext dbContext, Product product) =>
{
    dbContext.Products.Add(product);
    await dbContext.SaveChangesAsync();
    return Results.Created($"/products/{product.Id}", product);
});
```

- The following code is for the update and delete endpoints:

```
app.MapPut("/products/{id}", async (AppDbContext dbContext, Product product, int id)
{
    var p = await dbContext.Products.Where( p => p.Id == id ).FirstOrDefaultAsync();
    if(p != null)
    {
        p.Price = product.Price;
        if(!string.IsNullOrEmpty(product.Name))
            p.Name = product.Name;

        dbContext.Products.Update(p);
        await dbContext.SaveChangesAsync();
    }
    return Results.Ok(p);
});

app.MapDelete("/products/{id}", async ( AppDbContext dbContext, int id) =>
{
    var product = await dbContext.Products.Where(p => p.Id == id).FirstOrDefaultAsync
    if (product != null)
    {
        dbContext.Products.Remove(product);
        await dbContext.SaveChangesAsync();
    }
    return Results.Ok(product);
});
```

- Save the file.
- Next, we migrate the database.

6. Set Up the Database

- Ensure your connection string in `appsettings.json` and `appsettings.Development.json` are correct.
- If you are using SQL Server:
    - Create a database named database based on the connection string.
    - Create a username and password for the database.
    - Change value `"UseInMemoryDatabase": true`, to `"UseInMemoryDatabase": false`, in `appsettings.json` and `appsettings.Development.json`.
    - Use EF Core migrations to create the database:

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

    - After running the above commands, you should see a new database based your database connection string in your SQL Server instance.
    - If you have errors related to Invariant Globalization, you may disable `InvariantGlobalization` as `false` on project file, `efcoredb.csproj`.

- If you are using In-Memory Database:
    - No additional setup is required.

7. Add Scalar UI for OpenAPI (Optional)

- To enhance the API documentation, you can add Scalar UI for OpenAPI. This provides a user-friendly interface to interact with your API.
- Install the Scalar UI package:

```
dotnet add package Scalar.AspNetCore
```

- In `Program.cs`, add the following lines to configure Scalar UI:

```
using Scalar.AspNetCore;

...

if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
    app.MapScalarApiReference();
}
```

8. Build and Run the Application

- Use `dotnet run` to start the application.

```
dotnet run
```

- You can also run the application with https profile:

```
dotnet run --launch-profile https
```

9. Create the `.http` File for Testing

- Create a file `EfCoreMinimalApi.http` with test requests:

```
@efcoredb_HostAddress = http://localhost:5069

### Get all
GET {{efcoredb_HostAddress}}/products
```

```
Accept: application/json

### Get by id

GET {{efcoredb_HostAddress}}/products/1
Accept: application/json

### Create

POST {{efcoredb_HostAddress}}/products
Content-Type: application/json

{
  "name": "Product 1",
  "price": 100
}

###
PUT {{efcoredb_HostAddress}}/products/1
Content-Type: application/json

{
  "name": "Product 41edit",
  "price": 100
}

### Delete by id

DELETE {{efcoredb_HostAddress}}/products/2
Accept: application/json
```

- Change the port number to match your application's port number.

10. Testing the API

- Use the REST Client extension in Visual Studio Code to send requests.
- Open the `efcoredb.http` file and click on the "Send Request" link above each request to test the API endpoints.
- For adding a new product, you can modify the JSON body in the POST request as needed.
- To update and delete products, modify the PUT and DELETE requests accordingly.
- You can see my program output from REST client below:

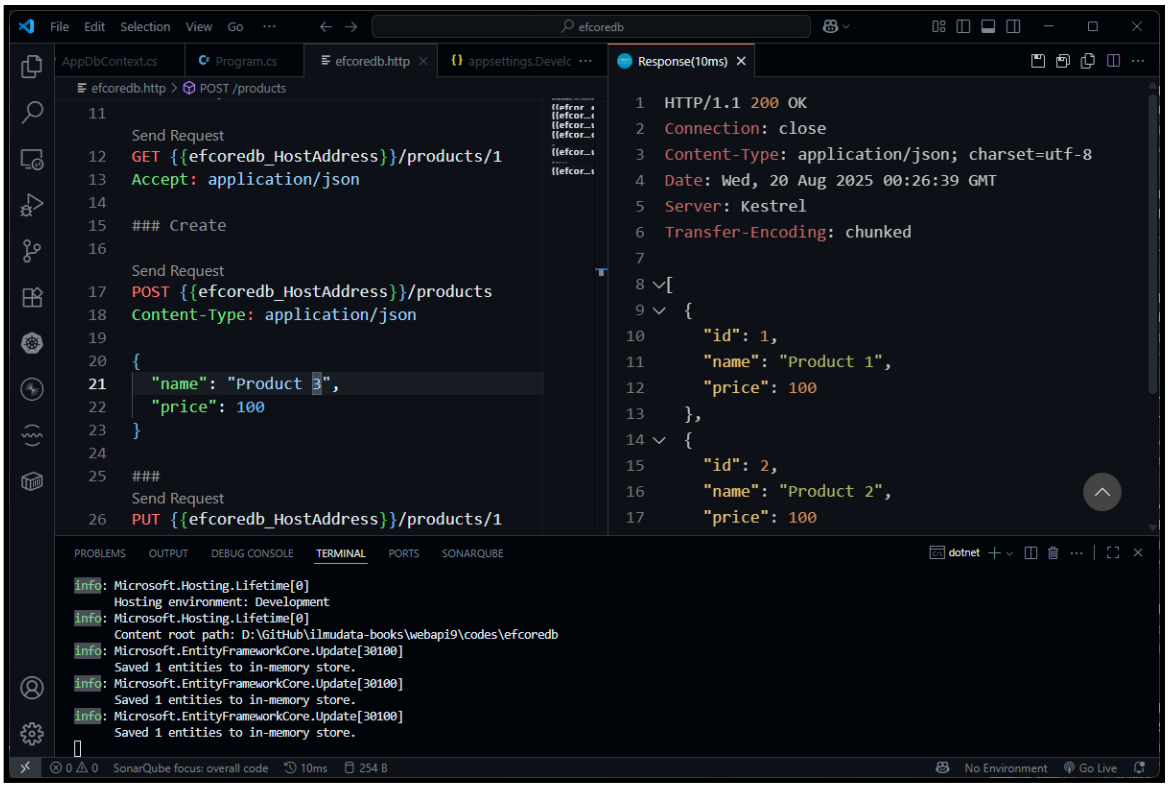


Figure 3.1 REST Client for efcoredb program.

- You also can use Scalar UI to test the API. Open browser and navigate to `https://<server>/scalar`.

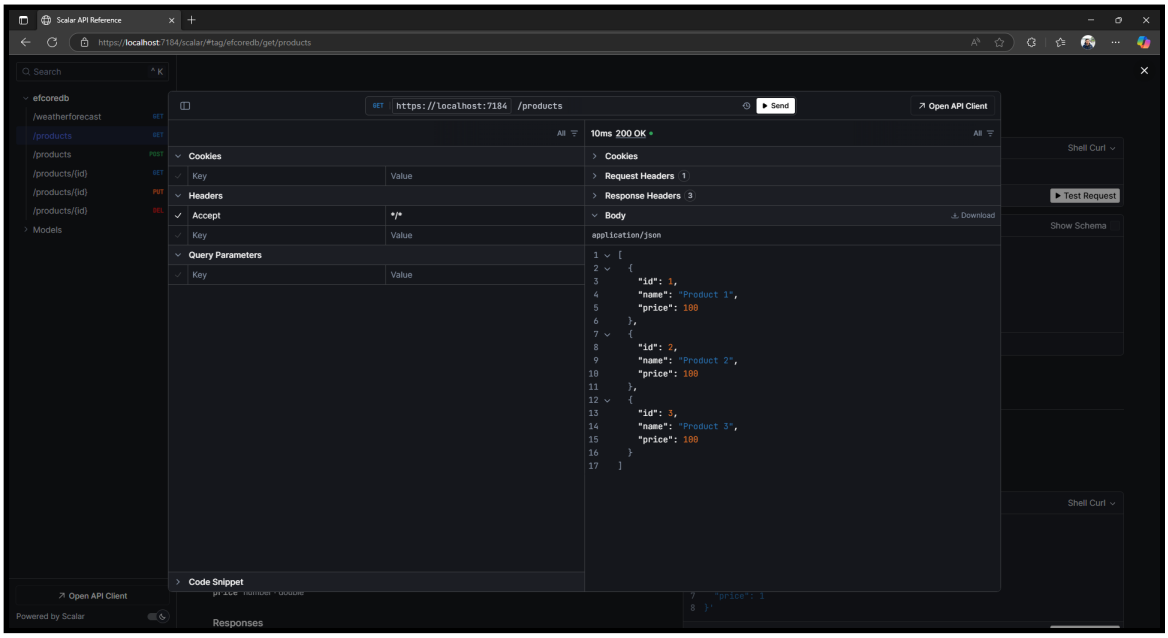


Figure 3.2 Scalar UI for efcoredb program.

3.4.4 Conclusion

In this lab, you've integrated Entity Framework Core 9.0 with Code First approach on ASP.NET Core Minimal API to build a simple RESTful service. This service demonstrates basic CRUD operations on a SQL Server database, showcasing how to use EF Core in modern ASP.NET Core applications.

3.5 Exercise 9: EF Core 9.0 Database First and ASP.NET Core Minimal API

In this exercise, you'll learn how to use Entity Framework Core 9.0 (EF Core) in an ASP.NET Core Minimal API project. You'll create a simple RESTful service that performs CRUD operations on a SQL Server database. You'll also learn how to use EF Core migrations to create and update the database schema.

3.5.1 Objective

Utilize the Database First approach with Entity Framework Core 9.0 in an ASP.NET Core Minimal API project. This lab will guide you through creating a RESTful service that interfaces with an existing database.

3.5.2 Requirements

- .NET 9.0 SDK installed
- Visual Studio Code or another code editor
- REST Client extension installed in Visual Studio Code for testing
- Access to a pre-existing SQL Server database
- SQL Server Management Studio or another database management tool (optional for database inspection)

3.5.3 Lab Steps

1. Create the `Product` Table in SQL Server

- Open SQL Server Management Studio (SSMS) and connect to your SQL Server instance.

- Execute the following SQL script to create a `Product` table:

```
CREATE DATABASE EfCoreLab
GO
USE EfCoreLab
GO

CREATE TABLE Product (
    Id INT IDENTITY PRIMARY KEY,
    Name NVARCHAR(100) NOT NULL,
    Price DECIMAL(18, 2) NOT NULL
);
```

- This script creates a new database named `EfCoreLab` and a `Product` table with `Id`, `Name`, and `Price` columns.
- You may create user and password for the database if you don’t want to use admin user.

2. Set Up the ASP.NET Core Project

- Open a command prompt or terminal.
- Navigate to your desired working directory or create a new one.

```
mkdir efcoredbfirst
cd efcoredbfirst
```

- Create a new ASP.NET Core Minimal API project:

```
dotnet new webapi
```

- Open the project in Visual Studio Code:

```
code .
```

3. Install Entity Framework Core Tools

- Install EF Core Design and SQL Server packages via NuGet:

```
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Tools
```

- Install the EF Core CLI tools globally (if not already installed):

```
dotnet tool install --global dotnet-ef
```

4. Scaffold DbContext and Models from Existing Database

- Assuming you have access to an existing SQL Server database, run the following command to scaffold the DbContext and entity classes. Replace the connection string and other options as needed:

```
dotnet ef dbcontext scaffold "db-connection-string" Microsoft.EntityFrameworkCore.Sql
```

- Change the connection string `db-connection-string` to match your database configuration. Here is an example:

```
"Server=localhost;Database=EfCoreLab;User Id=tester;Password=pass123;TrustServerCerti
```

  If you use database access with Windows Authentication, you can use:

```
"Server=localhost;Database=EfCoreLab;Trusted_Connection=True;TrustServerCertificate=T
```

- If you have errors related to Invariant Globalization, you may disable `InvariantGlobalization` as `false` on project file, `efcoredbfirst.csproj`.

- This command creates C# models and a DbContext based on the schema of the existing database in the `Models` directory.

- You can see my output from the command below:

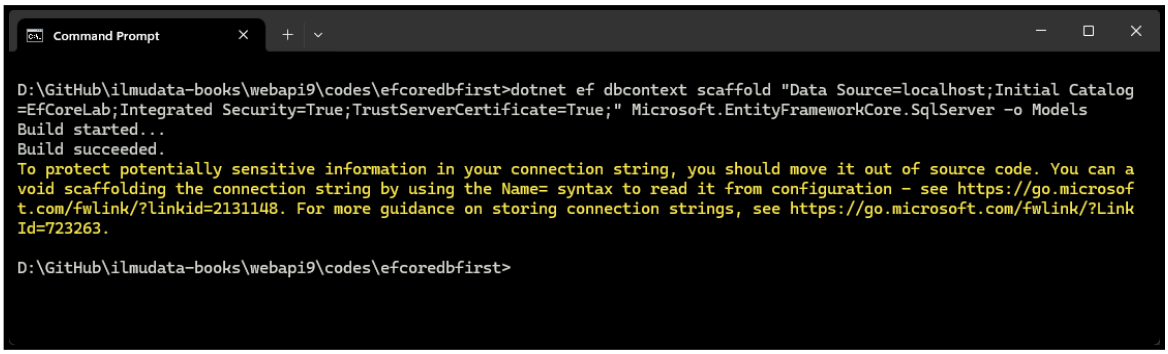


Figure 3.3 Output from scaffold command.

- We have warnings about security on connection string. Open `EfCoreLabContext.cs`, remove or comment out on `protected override void OnConfiguring.`.

- We move our database connection string to `appsettings.json` and `appsettings.Development.json` files.

- In `appsettings.json` and `appsettings.Development.json` files, add database configuration with the following content:

```
{
    "ConnectionStrings": {
        "MyDB": "db-connection-string"
    }
}
```

- Change the connection string `db-connection-string` to match your database configuration.

5. Configure the DbContext in Program.cs

- In `Program.cs`, configure the DbContext:

```
using efcoredbfirst.Models;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddDbContext<EfCoreLabContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("MyDB")));

var app = builder.Build();

// Define routes and logic using the scaffolded DbContext and models
// ...

app.Run();
```

6. Implement Basic CRUD Operations

- Implement minimal API endpoints to perform CRUD operations. Use the scaffolded DbContext and models to interact with the database.

- We can copy the code from previous lab, Exercise 8, step 5.

- Change the DbContext from `AppDbContext` to `EfCoreLabContext`.

- The following is the code for GET endpoints:

```
app.MapGet("/products", async (EfCoreLabContext dbContext) =>
    await dbContext.Products.ToListAsync());

app.MapGet("/products/{id}", async (EfCoreLabContext dbContext, int id) =>
{
    var product = await dbContext.Products.Where(p => p.Id == id).FirstOrDefaultAsync
    return Results.Ok(product);
});
```

- The following is the code for POST and PUT endpoints:

```
app.MapPost("/products", async (EfCoreLabContext dbContext, Product product) =>
{
    dbContext.Products.Add(product);
    await dbContext.SaveChangesAsync();
    return Results.Created($"/products/{product.Id}", product);
});

app.MapPut("/products/{id}", async (EfCoreLabContext dbContext, Product product, int
{
    var p = await dbContext.Products.Where(p => p.Id == id).FirstOrDefaultAsync();
    if(p != null)
    {
        p.Price = product.Price;
        if(!string.IsNullOrEmpty(product.Name))
            p.Name = product.Name;

        dbContext.Products.Update(p);
        await dbContext.SaveChangesAsync();
    }
    return Results.Ok(p);
});
```

- The following is the code for DELETE endpoint:

```
app.MapDelete("/products/{id}", async (EfCoreLabContext dbContext, int id) =>
{
    var product = await dbContext.Products.Where(p => p.Id == id).FirstOrDefaultAsync
    if (product != null)
    {
        dbContext.Products.Remove(product);
        await dbContext.SaveChangesAsync();
    }
    return Results.Ok(product);
});
```

- Save all files.

7. Add Scalar UI for OpenAPI (Optional)

- To enhance the API documentation, you can add Scalar UI for OpenAPI. This provides a user-friendly interface to interact with your API.

- Install the Scalar UI package:

```
dotnet add package Scalar.AspNetCore
```

- In `Program.cs`, add the following lines to configure Scalar UI:

```
using Scalar.AspNetCore;

...

if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
    app.MapScalarApiReference();
}
```

8. Build and Run the Application

- Use `dotnet run` to start the application.
- You can also run the application with https profile:

```
dotnet run --launch-profile https
```

9. Create the `.http` File for Testing

- Create a file `efcoredbfirst.http` in your project with test requests for the CRUD operations:

```
@efcoredbfirst_HostAddress = http://localhost:5219

### Get all
GET {{efcoredbfirst_HostAddress}}/products
Accept: application/json

### Get by id

GET {{efcoredbfirst_HostAddress}}/products/1
Accept: application/json

### Create

POST {{efcoredbfirst_HostAddress}}/products
Content-Type: application/json

{
  "name": "Product 1",
  "price": 100
}

###
PUT {{efcoredbfirst_HostAddress}}/products/1
Content-Type: application/json

{
  "name": "Product 41edit",
  "price": 100
}

### Delete by id

DELETE {{efcoredbfirst_HostAddress}}/products/2
Accept: application/json
```

```
###
```

10. Testing the API

- Use the REST Client extension in Visual Studio Code to send requests and interact with your database.
- For adding a new product, you can modify the JSON body in the POST request as needed.
- To update and delete products, modify the PUT and DELETE requests accordingly.

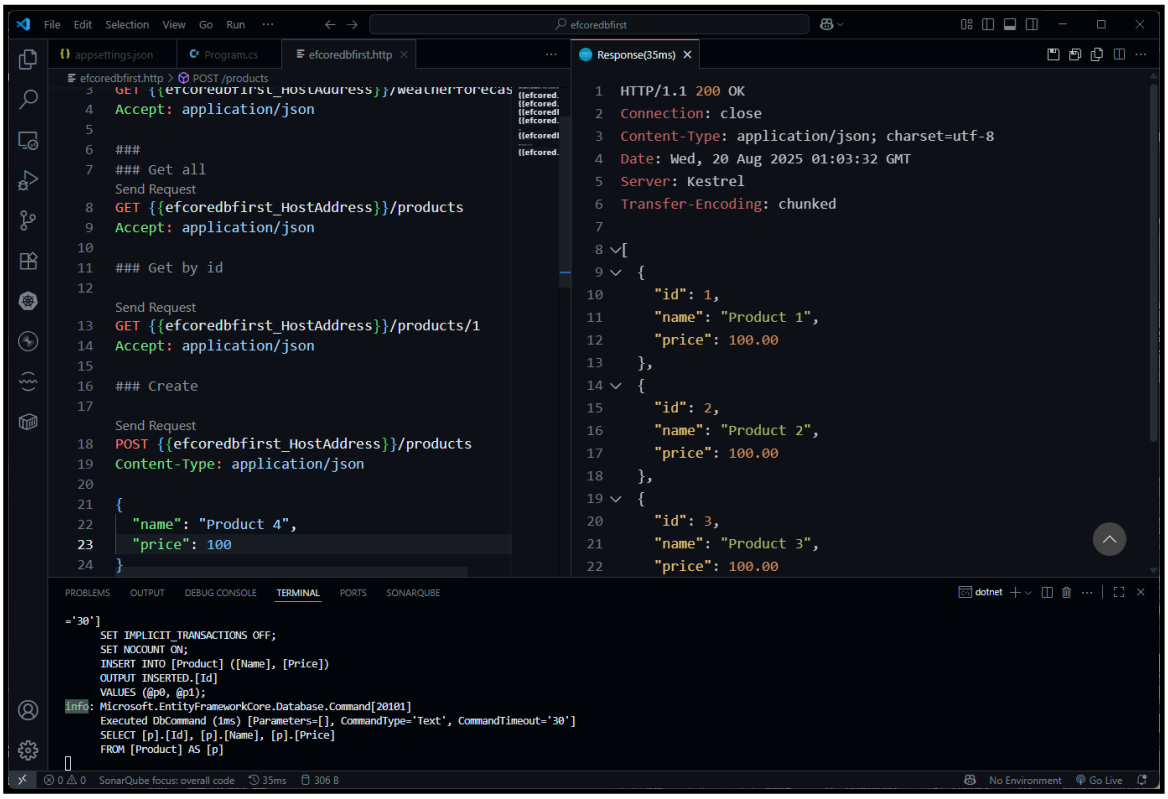


Figure 3.4 REST Client for efcoredbfirst program.

- You also can use Scalar UI to test the API. Open browser and navigate to `https://<server>/scalar`.
- Click on the “Products” endpoint to see the available operations.

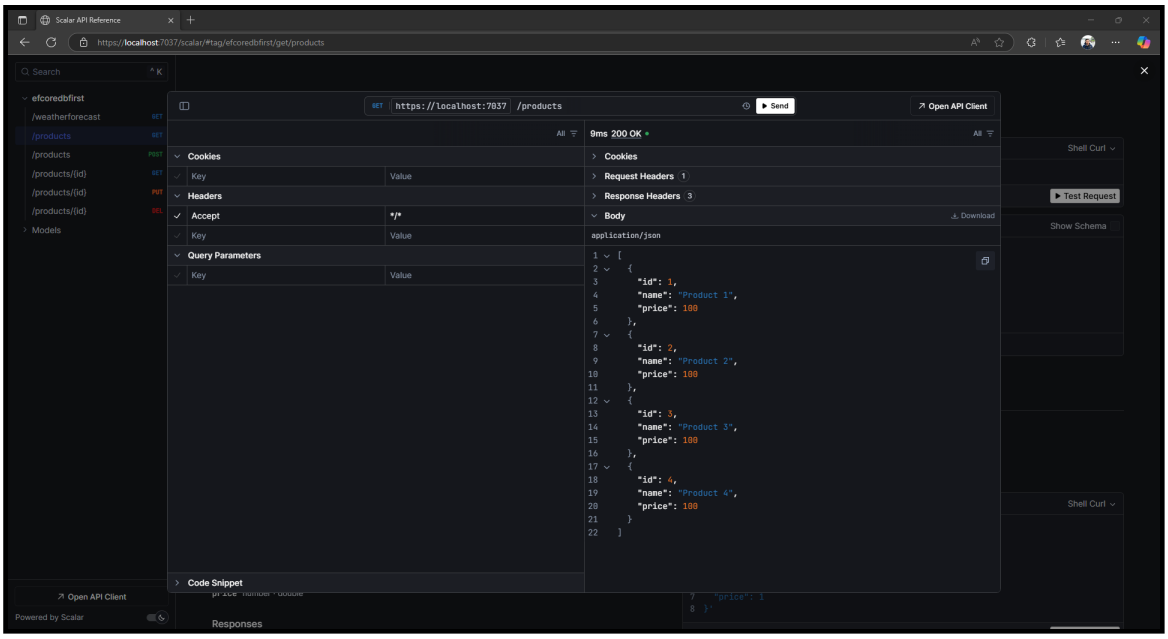


Figure 3.5 Scalar UI for efcoredbfirst program.

3.5.4 Conclusion

In this lab, you have used the Database First approach with EF Core 9.0 in an ASP.NET Core Minimal API project. You've learned how to scaffold DbContext and entity classes from an existing database, configure the DbContext, and create endpoints to perform CRUD operations.

3.6 Introduction to Database Transactions

What is a Database Transaction?

A database transaction is a fundamental concept in database management systems, representing a single unit of work that either fully succeeds or fully fails. It's a sequence of operations performed as a single logical unit, which means that if any part of the transaction fails, the entire transaction fails, and the database state is left unchanged.

In technical terms, a transaction is a series of read/write operations that begins with the command `BEGIN TRANSACTION` and ends with either `COMMIT` (which saves the changes) or `ROLLBACK` (which undoes all changes made during the transaction).

Why Are Database Transactions Important?

1. **Atomicity:** Transactions ensure atomicity, which means that all operations within a transaction block are treated as a single unit. Either all operations are executed successfully, or none are.

2. **Consistency:** Transactions help in maintaining the consistency of the database. They ensure that the database transitions from one valid state to another valid state, without exposing intermediate states to end users.

3. **Isolation:** Transactions provide isolation, meaning they can operate independently without interference from other concurrent transactions, thus preventing data corruption.

4. **Durability:** Once a transaction is committed, the changes it has made to the data are permanent, even in the case of a system failure. This property ensures data integrity and reliability.

How Are Transactions Implemented in Databases?

- **SQL Databases:** In SQL databases like MySQL, SQL Server, and PostgreSQL, transactions are managed using SQL commands. You start a transaction with `BEGIN TRANSACTION`, make your database read and write operations, and then either commit with `COMMIT` or rollback with `ROLLBACK` based on the success or failure of the operations.

- **NoSQL Databases:** Transaction support in NoSQL databases varies depending on the database system. For example, MongoDB offers multi-document transactions, which work similarly to transactions in SQL databases, ensuring atomicity, consistency, isolation, and durability.

- **Entity Framework in .NET:** In .NET applications using Entity Framework, transactions are handled through the DbContext. You can start a transaction using `dbContext.Database.BeginTransaction()`, perform data operations using the DbContext, and then commit or rollback the transaction.

- **Handling in Application Code:** Transactions can also be managed programmatically in application code, where you can include business logic to determine the success or failure of a transaction.

Understanding database transactions is crucial for building robust, reliable, and consistent database-driven applications. They are key in ensuring data integrity and consistency, especially in systems where multiple concurrent operations occur.

By properly implementing transactions, developers can prevent data anomalies, maintain data accuracy, and enhance the overall stability of applications.

3.7 Exercise 10: Database Transaction

In this exercise, you'll learn how to use Entity Framework Core 9.0 (EF Core) in an ASP.NET Core Minimal API project. You'll create a simple RESTful service that performs database transaction on a SQL Server database. You'll also learn how to use EF Core migrations to create and update the database schema.

3.7.1 Objective

Demonstrate how to manage database transactions in an ASP.NET Core 9 Minimal API application using EF Core 9.0. This lab will use a code-first approach with multiple tables to illustrate transaction handling.

3.7.2 Requirements

- .NET 9.0 SDK installed
- Visual Studio Code or another code editor
- SQL Server or another relational database
- REST Client extension in Visual Studio Code for testing

3.7.3 Lab Steps

1. Set Up the ASP.NET Core Project

- Open a command prompt or terminal.
- Navigate to your desired working directory or create a new one.

```
mkdir efcoretrans
cd efcoretrans
```

- Create a new ASP.NET Core Minimal API project:

```
dotnet new webapi
```

- Open the project in Visual Studio Code:

```
code .
```

2. Install Entity Framework Core

- Install the necessary EF Core packages for your chosen database. For SQL Server, use:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Design
```

3. Define Models and DbContext

- Create a folder named `Models`.
- Inside `Models`, create classes for your entities. For illustration, let's use `Order.cs`:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace efcoretrans.Models;
public class Order
{
    [Key]
    [DatabaseGenerated( DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    public string CustomerName { get; set; } = "";
    public List<Product> Products { get; set; }
}
```

- Write codes in `Product.cs` file with following codes:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace efcoretrans.Models;

public class Product
{
    [Key]
    [DatabaseGenerated( DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public decimal Price { get; set; }
}
```

- Inside `Models` folder, create `AppDbContext.cs`:

```
using Microsoft.EntityFrameworkCore;

namespace efcoretrans.Models;
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options) { }
```

```
    public DbSet<Order> Orders { get; set; }
    public DbSet<Product> Products { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        // Define precision for decimal property
        modelBuilder.Entity<Product>()
            .Property(product => product.Price)
            .HasPrecision(18, 2);
    }
}
```

4. Configure the DbContext in Program.cs

- In `Program.cs`, configure the DbContext to use SQL Server:

```
using efcoretrans.Models;
using Microsoft.EntityFrameworkCore;

//...

builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("MyDB")));

//...
var app = builder.Build();
//...
```

5. Implement Transactional Operations

- Implement an API endpoint that demonstrates a transactional operation involving both `Order` and `Product`:

```
app.MapPost("/createorder", async (AppDbContext dbContext, Order order) =>
{
    using var transaction = await dbContext.Database.BeginTransactionAsync();

    try
    {
        dbContext.Orders.Add(order);
        await dbContext.SaveChangesAsync();

        await transaction.CommitAsync();
        return Results.Ok(order);
    }
    catch
    {
        // Rollback transaction if there are any exceptions
        await transaction.RollbackAsync();
        throw;
    }
});
```

6. Create the Database and Apply Migrations

- Configure your connection string in `appsettings.json` and `appsettings.Development.json`.

```
{
  // ..
  "ConnectionStrings": {
    "MyDB": "server=localhost; database=EfCoreLab; uid=tester;pwd=pass123; TrustServe
  },

  // ..
}
```

- Change the connection string to match your database configuration.
- Create migrations and update the database:

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

- If you have errors related to Invariant Globalization, you may disable `InvariantGlobalization` as `false` on project file, `efcoretrans.csproj`.

7. Add Scalar UI for OpenAPI (Optional)

- To enhance the API documentation, you can add Scalar UI for OpenAPI. This provides a user-friendly interface to interact with your API.
- Install the Scalar UI package:

```
dotnet add package Scalar.AspNetCore
```

- In `Program.cs`, add the following lines to configure Scalar UI:

```
using Scalar.AspNetCore;

...

if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
    app.MapScalarApiReference();
}
```

8. Build and Run the Application

- Compile and run your application using `dotnet run`.

- You can also run the application with https profile:

```
dotnet run --launch-profile https
```

9. Create the `.http` File for Testing

- Create a file `efcoretrans.http` in your project with test requests for the transactional endpoint:

```
@efcoretrans_HostAddress = http://localhost:5038

POST {{efcoretrans_HostAddress}}/createorder
Content-Type: application/json
Accept: application/json

{
    "customerName": "Ahmad Lee",
    "products": [
      { "name": "Product 1", "price": 10.99 },
      { "name": "Product 2", "price": 15.50 }
    ]
}
###
```

10. Testing the Transactional Endpoint

- Use the REST Client extension in Visual Studio Code to send requests and validate the transactional behavior.
- For adding a new product, you can modify the JSON body in the POST request as needed.
- To update and delete products, modify the PUT and DELETE requests accordingly.

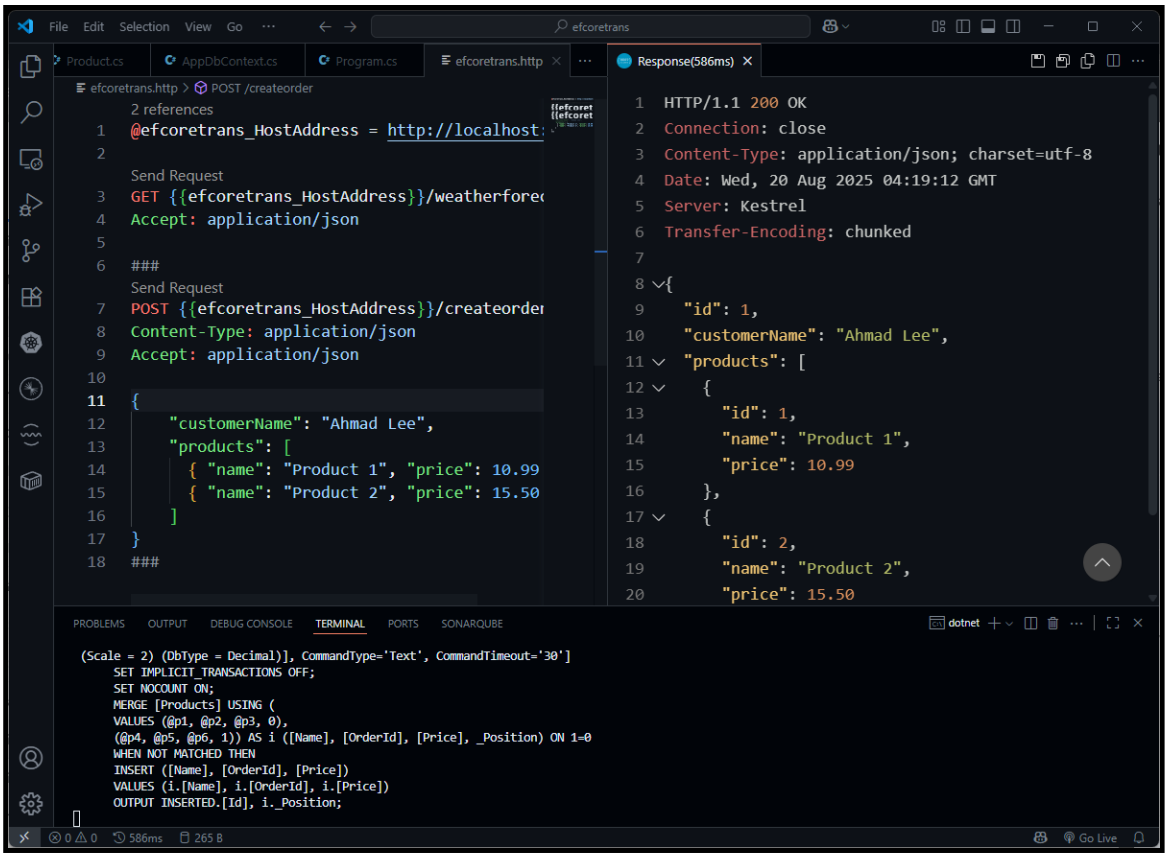


Figure 3.6 REST Client for efcoretrans program.

- We also use Scalar UI to test the API. Open browser and navigate to `https://<server>/scalar`.

3.7.4 Conclusion

In this lab, you've learned how to manage database transactions in an ASP.NET Core 9 Minimal API application using EF Core 9.0. You've implemented a transactional operation that involves multiple tables, demonstrating how to ensure data integrity in complex scenarios.

3.8 Introduction to NoSQL Databases

What is a NoSQL Database?

NoSQL databases, standing for "Not Only SQL" or "Non-SQL," represent a wide range of database technologies that were developed to handle the shortcomings of traditional relational database management systems (RDBMS). Unlike RDBMS which use tables to store data, NoSQL databases use various data models,

including document, key-value, wide-column, and graph formats. These databases are known for their flexibility, scalability, and the ability to handle large volumes of unstructured or semi-structured data.

Why Are NoSQL Databases Important?

1. **Flexibility:** NoSQL databases do not require a fixed schema, allowing the data model to evolve over time and developers to store complex data structures easily.

2. **Scalability:** They are designed to scale out by distributing data across multiple servers, making them well-suited for cloud computing and big data applications.

3. **High Performance:** NoSQL databases are optimized for specific data models and access patterns, which can lead to higher performance for certain types of applications, particularly those requiring large-scale data processing.

4. **Handling Unstructured Data:** With the increasing amount of unstructured data (such as social media content, multimedia, text), NoSQL databases provide efficient ways to store and retrieve such data.

How Are NoSQL Databases Implemented and Used?

- **Types of NoSQL Databases:**
  - **Document-Oriented:** Such as MongoDB and CouchDB, store data in JSON-like documents and are ideal for storing, retrieving, and managing document-oriented information.
  - **Key-Value Stores:** Like Redis and DynamoDB, store data as a collection of key-value pairs. They are highly efficient for lookups and are useful for caching, sessions, and simple data models.
  - **Wide-Column Stores:** Including Cassandra and HBase, use columns to store data. They are excellent for analyzing large datasets.
  - **Graph Databases:** Such as Neo4j and Amazon Neptune, are used for storing and navigating relationships. They are ideal for social networks, fraud detection, and recommendation systems.
- **Usage in Applications:**
  - NoSQL databases are often used in big data applications, real-time web applications, and in services requiring rapid development and iteration.
  - They are typically accessed through APIs provided by the database vendors, with support for various programming languages.

- **Considerations:**
  - Choose a NoSQL database based on the specific requirements of your application, such as data model, scalability needs, and consistency requirements.
  - Understand the trade-offs, as NoSQL databases often provide eventual consistency rather than the strong consistency offered by traditional RDBMS.

NoSQL databases offer a modern approach to data storage and management, addressing the challenges and requirements of contemporary application development. They provide developers with efficient ways to handle varied and voluminous data, making them an essential component of the modern technology landscape, especially in the realms of big data, real-time processing, and cloud computing. Understanding when and how to effectively utilize NoSQL technologies is crucial for developers and architects designing scalable and flexible systems.

3.9 Exercise 11: NoSQL Database and ASP.NET Core Minimal API

This lab offers practical experience in using MongoDB with ASP.NET Core, demonstrating the advantages of NoSQL databases in handling dynamic, schema-less data models and providing a foundation for building scalable and high-performance web APIs.

3.9.1 Objective

Learn to integrate MongoDB, a NoSQL database, with an ASP.NET Core 9.0 Minimal API project. This lab will cover the steps to perform CRUD operations on a MongoDB collection.

3.9.2 Requirements

- .NET 9.0 SDK installed
- MongoDB installed and running
- Visual Studio Code or another code editor
- MongoDB C# driver
- REST Client extension installed in Visual Studio Code for testing

3.9.3 Lab Steps

1. Set Up the MongoDB Server

- You can install MongoDB on your local machine or use a cloud-hosted MongoDB service.
- Download and install MongoDB Community Server from https://www.mongodb.com/try/download/community.
- In this exercise, we'll use a local MongoDB server based Docker container.
- You may use Docker Desktop. Install Docker Desktop from https://www.docker.com/products/docker-desktop.
- For demo purpose, we create docker-compose file to run MongoDB server and mongo-express.
- mongo-express is a web-based MongoDB admin interface written with Node.js, Express, and Bootstrap3.
- The following is a content of `docker-compose.yml` file

```
version: "3.8"
services:
  mongo:
    image: mongo:latest
    container_name: mongo
    environment:
        - MONGO_INITDB_ROOT_USERNAME=root
        - MONGO_INITDB_ROOT_PASSWORD=pass123
    restart: unless-stopped
    ports:
      - "27017:27017"
    networks:
      - mongonet
    volumes:
      - ./database/db:/data/db
      - ./database/dev.archive:/Databases/dev.archive
      - ./database/production:/Databases/production
  mongo-express:
    image: mongo-express
    container_name: mexpress
    environment:
      - ME_CONFIG_MONGODB_ADMINUSERNAME=root
      - ME_CONFIG_MONGODB_ADMINPASSWORD=pass123
      - ME_CONFIG_MONGODB_URL=mongodb://root:pass123@mongo:27017/?authSource=admin
      - ME_CONFIG_BASICAUTH_USERNAME=mexpress
      - ME_CONFIG_BASICAUTH_PASSWORD=pass123
    links:
      - mongo
```

```
    restart: unless-stopped
    ports:
      - "8081:8081"
    networks:
      - mongonet

networks:
  mongonet:
```

- Now you can run this using this command.

```
docker-compose up
```

- You also can run it in background.

```
docker-compose up -d
```

- After running the command, you can access mongo-express from `http://localhost:8081`.
- Enter username and password as defined in `docker-compose.yml` file.
- You can see my mongo-express below:

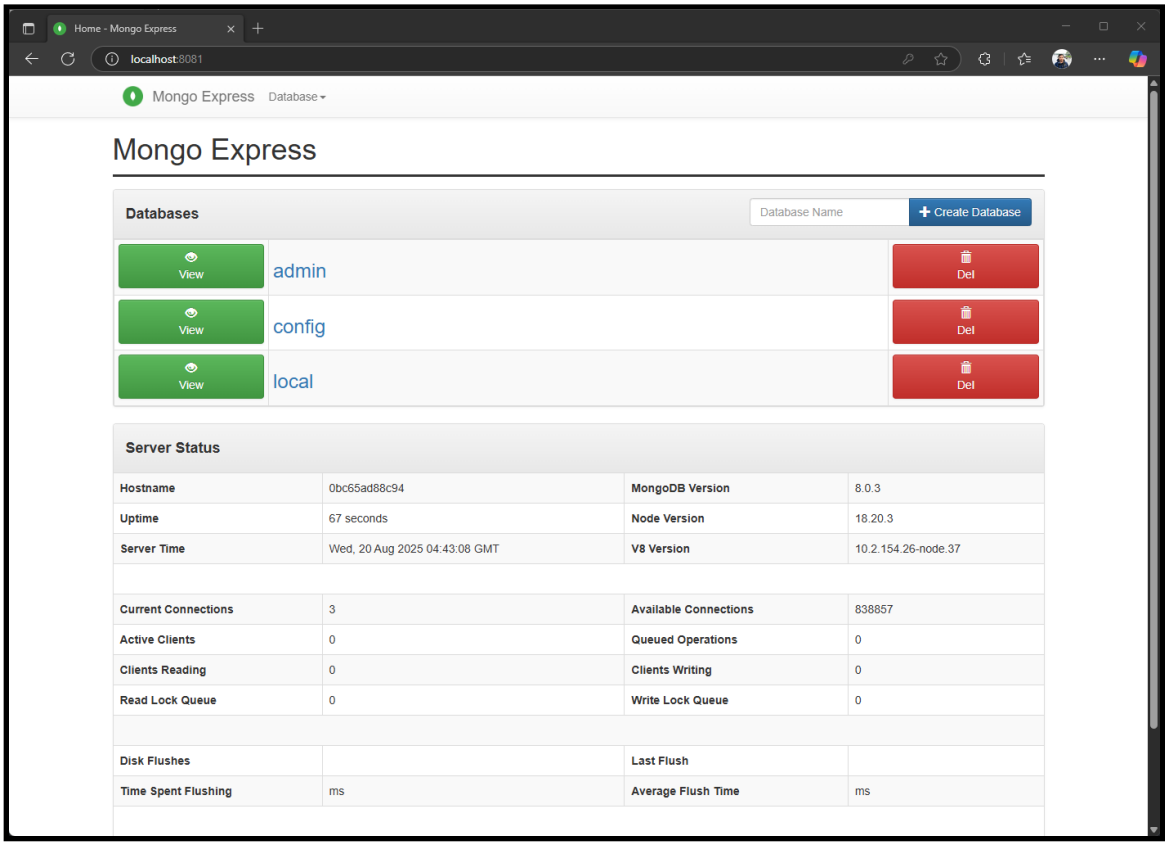


Figure 3.7 A dashboard of mongo-express application.

2. Set Up the ASP.NET Core Project

- Open a command prompt or terminal.
- Navigate to your desired working directory or create a new one.

```
mkdir mongodbapp
cd mongodbapp
```

- Create a new ASP.NET Core Minimal API project:

```
dotnet new webapi
```

- Open the project in Visual Studio Code:

```
code .
```

3. Install MongoDB C# Driver

- Install the MongoDB C# driver via NuGet:

```
dotnet add package MongoDB.Driver
```

4. Define a Model

- Create a `Models` folder.
- Inside `Models`, create a `Product` class:

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;

namespace mongodbapp.Models;

public class Product
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string? Id { get; set; }

    public string Name { get; set; } = "";
    public decimal Price { get; set; }
}
```

5. Create a MongoDB Context

- Inside `Models`, create `MongoDbContext.cs`:

```
using MongoDB.Driver;

namespace mongodbapp.Models;
public class MongoDbContext
{
    private readonly IMongoDatabase _database;

    public MongoDbContext(IConfiguration configuration)
    {
        var client = new MongoClient(configuration.GetConnectionString("MongoDb"));
        _database = client.GetDatabase("MongoDbDemo");
    }

    public IMongoCollection<Product> Products => _database.GetCollection<Product>("Pr
}
```

6. Configure the Service in Program.cs

- In `Program.cs`, add the MongoDB context as a service:

```
using mongodbapp.Models;
using MongoDB.Driver;
using MongoDB.Bson;

// ...

builder.Services.AddSingleton<MongoDbContext>();
// ...
```

- Configure your connection string in `appsettings.json` and `appsettings.Development.json`.

```
{
  // ..
  "ConnectionStrings": {
    "MongoDb": "mongodb://root:pass123@localhost:27017/?authSource=admin"
  },

  // ..
}
```

- Change the connection string `root` and `pass123` to match your database configuration.

7. Implement CRUD Operations

- Implement endpoints in `Program.cs` to perform CRUD operations:

```
// POST: Add a new product
app.MapPost("/products", async (MongoDbContext dbContext, Product product) =>
{
    await dbContext.Products.InsertOneAsync(product);
```

```
    return Results.Created($"/products/{product.Id}", product);
});

// GET: Retrieve all products
app.MapGet("/products", async (MongoDbContext dbContext) =>
    await dbContext.Products.Find(product => true).ToListAsync());

// GET: Retrieve a single product by ID
app.MapGet("/products/{id}", async (MongoDbContext dbContext, string id) =>
{
    var product = await dbContext.Products.Find(p => p.Id == id).FirstOrDefaultAsync(
    return product is not null ? Results.Ok(product) : Results.NotFound();
});

// Additional endpoints for PUT and DELETE
```

- The following is the code for PUT and DELETE endpoints:

```
// PUT: Update a product
app.MapPut("/products/{id}", async (MongoDbContext dbContext, Product product, string
{
    var filter = Builders<Product>.Filter.Eq(p => p.Id, id);
    var update = Builders<Product>.Update
        .Set(p => p.Name, product.Name)
        .Set(p => p.Price, product.Price);

    await dbContext.Products.UpdateOneAsync(filter, update);
    return Results.Ok(product);
});

// DELETE: Delete a product
app.MapDelete("/products/{id}", async (MongoDbContext dbContext, string id) =>
{
    var filter = Builders<Product>.Filter.Eq(p => p.Id, id);
    await dbContext.Products.DeleteOneAsync(filter);
    return Results.Ok();
});
```

8. Add Scalar UI for OpenAPI (Optional)

- To enhance the API documentation, you can add Scalar UI for OpenAPI. This provides a user-friendly interface to interact with your API.
- Install the Scalar UI package:

```
dotnet add package Scalar.AspNetCore
```

- In `Program.cs`, add the following lines to configure Scalar UI:

```
using Scalar.AspNetCore;

...
```

```
if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
    app.MapScalarApiReference();
}
```

9. Build and Run the Application

- Use `dotnet run` to start the application.
- You can also run the application with https profile:

```
dotnet run --launch-profile https
```

10. Create the `.http` File for Testing

- Create a `mongodbapp.http` file in your project with test requests for CRUD operations if it's not exists:

```
@mongodbapp_HostAddress = http://localhost:5103

### Add a New Product
POST {{mongodbapp_HostAddress}}/products
Content-Type: application/json

{
    "name": "Sample Product 1",
    "price": 9.99
}

### Get All Products
GET {{mongodbapp_HostAddress}}/products

### Get Product by ID
GET {{mongodbapp_HostAddress}}/products/657599bc6d53d83cc2780da5

### Update Product
PUT {{mongodbapp_HostAddress}}/products/657599bc6d53d83cc2780da5
Content-Type: application/json
Accept: */*

{
    "id": "657599bc6d53d83cc2780da5",
    "name": "Updated Product",
    "price": 19.99
}

### Delete Product
DELETE {{mongodbapp_HostAddress}}/products/657599bc6d53d83cc2780da5
```

- Change the port number to match your application's port number.
- You may change the product ID `657599bc6d53d83cc2780da5` to match your data.

11. Testing the API

- You also can use Scalar UI to test the API. Open browser and navigate to `https://<server>/scalar`.
- You can see my Scalar UI below:

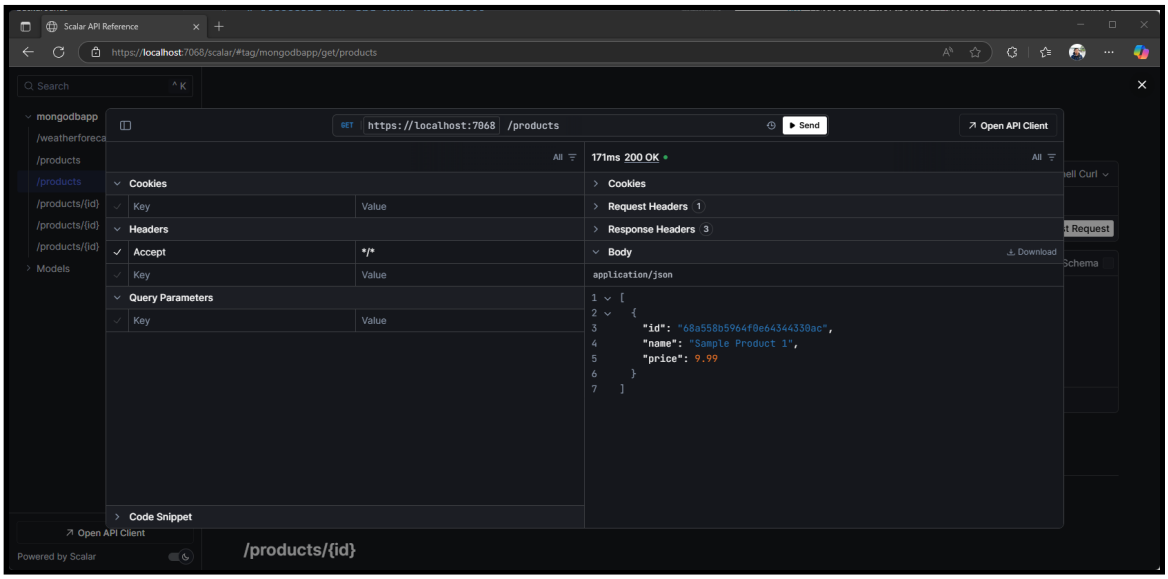


Figure 3.8 Scalar UI for mongodbapp program.

- Click `Try it out` button and click `Execute` button.
- Use the REST Client extension in Visual Studio Code to send requests.
- Click `Send Request` button.
- You can see my output from REST client below:

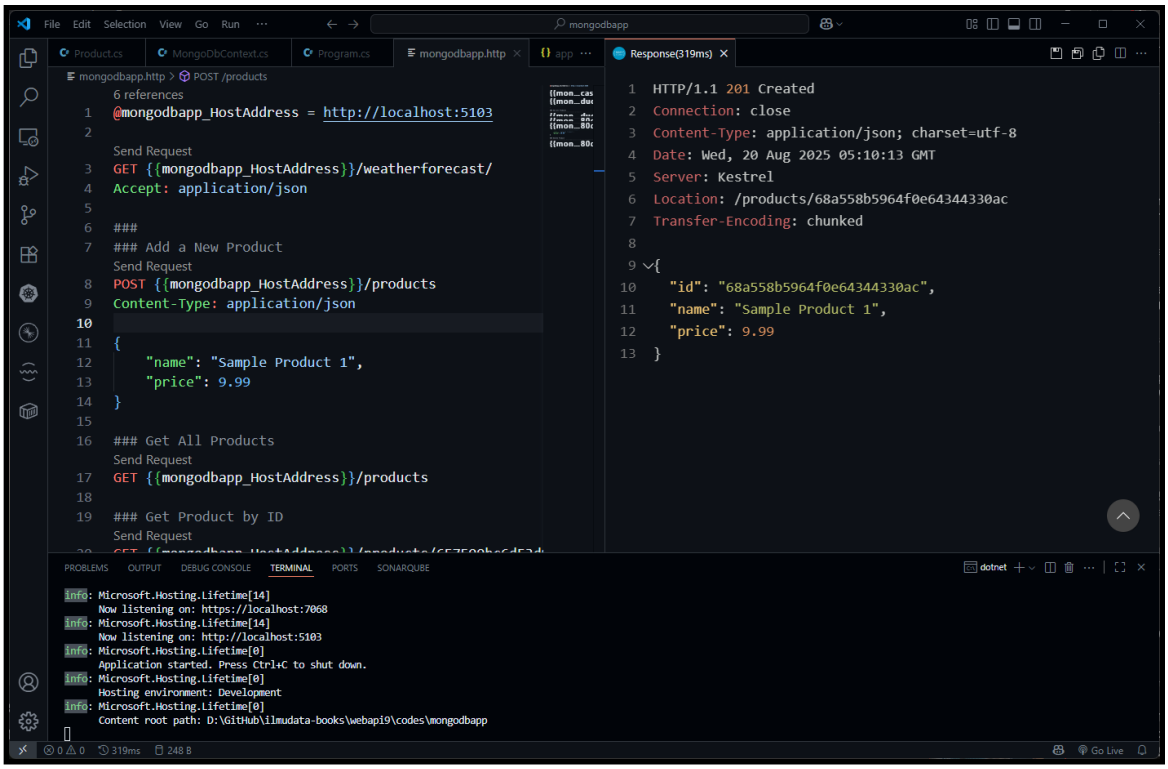


Figure 3.9 REST Client for mongodbapp program.

3.9.4 Conclusion

In this lab, you've integrated MongoDB with an ASP.NET Core 9.0 Minimal API project and performed CRUD operations using MongoDB as the data store. This lab provides a hands-on experience in working with NoSQL databases in .NET applications, showcasing MongoDB's flexibility and ease of use for managing unstructured data.

4 Deep Dive into Web Security

4.1 Introduction

ASP.NET Core Minimal APIs, introduced in ASP.NET Core 6, offer a streamlined method for creating lightweight HTTP APIs. They are designed to simplify and speed up the process of setting up APIs, reducing the need for boilerplate code. Security in ASP.NET Core Minimal APIs encompasses authentication, authorization, data protection, and ensuring secure communication and data handling.

Why is Security in ASP.NET Core Minimal APIs Important?

1. **Protect Sensitive Data:** Securing APIs is crucial to protect sensitive data from unauthorized access and breaches.

2. **Compliance and Trust:** Proper security measures ensure compliance with legal and regulatory requirements and build trust with users.

3. **Prevent Attacks:** Security mechanisms help in preventing various types of attacks like SQL injection, cross-site scripting (XSS), and others.

4. **Secure Communication:** Implementing security protocols like HTTPS ensures that the data exchanged between the client and server is encrypted and secure.

How to Implement Security in ASP.NET Core Minimal APIs?

1. **Authentication and Authorization:**
    - **Authentication:** Determine the user's identity using various methods like JWT tokens, OAuth, or basic authentication.
    - **Authorization:** Ensure that an authenticated user has the right permissions to access resources. This can be role-based, policy-based, or claim-based authorization.
    - Example: Implementing JWT token-based authentication and securing endpoints with `[Authorize]` attribute.

2. **Data Protection:**
    - Protect sensitive data using ASP.NET Core's Data Protection APIs.
    - Encrypt sensitive information in configuration files.
3. **Securing Data Transfer:**
    - Enforce the use of HTTPS to secure data in transit.
    - Implement proper CORS policies to control cross-origin requests.
4. **Input Validation and Sanitization:**
    - Validate and sanitize user input to prevent injection attacks.
    - Use model validation to enforce data integrity.
5. **Dependency Management:**
    - Regularly update dependencies to incorporate security patches.
    - Use only trusted libraries and packages.
6. **Logging and Monitoring:**
    - Implement logging to record significant events and potential security incidents.
    - Monitor API usage and behavior to detect anomalies.
7. **Rate Limiting:**
    - Implement rate limiting to prevent abuse and denial-of-service attacks.

Security in ASP.NET Core Minimal APIs is not just an optional feature but a fundamental aspect that needs to be ingrained throughout the API development lifecycle. From authenticating users to protecting data and ensuring secure communication, each aspect plays a crucial role in safeguarding the API from various security threats. As such, understanding and implementing robust security practices is essential for developing reliable and secure ASP.NET Core Minimal APIs.

4.2 Exercise 12: Authentication and Authorization

In this lab, we'll implement authentication and authorization in an ASP.NET Core Minimal API application. This lab covers user registration, login with basic authentication, token generation, and accessing user profiles with token authentication. We'll use Entity Framework Core with a code-first approach for user data and bcrypt for password hashing.

4.2.1 Objective

Implement authentication and authorization in an ASP.NET Core Minimal API application. This lab covers user registration, login with basic authentication, token generation, and accessing user profiles with token authentication. We'll use Entity Framework Core with a code-first approach for user data and bcrypt for password hashing.

4.2.2 Requirements

- .NET 9.0 SDK installed
- Visual Studio Code or another code editor
- Entity Framework Core
- A package for bcrypt, like `BCrypt.Net-Next`
- A package for JWT (JSON Web Tokens), like `Microsoft.AspNetCore.Authentication.JwtBearer`

4.2.3 Lab Steps

1. Set Up the ASP.NET Core Project

- Open a command prompt or terminal.
- Navigate to your desired working directory or create a new one.

```
mkdir secrestapi
cd secrestapi
```

- Create a new ASP.NET Core Minimal API project:

```
dotnet new webapi
```

- Open the project in Visual Studio Code:

```
code .
```

2. Install Required Packages

- Install EF Core, bcrypt, and JWT packages:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package BCrypt.Net-Next
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

- Install the Scalar UI package:

```
dotnet add package Scalar.AspNetCore
```

3. Define Models and DbContext

- Create a `Models` folder and define a `ApiUser` model on `ApiUser.cs` file:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace secrestapi.Models;

public class ApiUser
{
    [Key]
    [DatabaseGenerated( DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    public string Username { get; set; } = "";
    public string Password { get; set; } = "";
    public string Name { get; set; } = "";
    public string Email { get; set; } = "";
}
```

  We also create DTOs (Data Transfer Objects) for user registration and login. These DTOs are used to transfer data between the client and server.

- Inside `Models` folder, write `UserLogin.cs` file and write the following code:

```
namespace secrestapi.Models;

public class UserLogin
{
    public string UserName { get; set; } = "";
    public string Password { get; set; } = "";
}
```

- We also create `UserToken.cs` file as DTO for JWT token. This DTO is used to transfer JWT token between the client and server.

```
namespace secrestapi.Models;

public class UserToken
{
    public string Token { set; get; } = "";
    public string ExpiredAt { set; get; } = "";
    public string Message { set; get; } = "";
}
```

- Inside a `Models` folder and implement `AppDbContext` on `AppDbContext.cs` file:

```
using Microsoft.EntityFrameworkCore;

namespace secrestapi.Models;

public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)

    public DbSet<ApiUser> Users { get; set; }
}
```

4. Configure DbContext and JWT in Program.cs

- Configure DbContext and add JWT authentication in `Program.cs`.
- In `Program.cs`, configure the DbContext:

```
using secrestapi.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;
using Microsoft.IdentityModel.Tokens;
using System.Text;
using BC = BCrypt.Net.BCrypt;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using Microsoft.OpenApi.Models;
using Scalar.AspNetCore;
// ...

builder.Services.AddDbContext<AppDbContext>(options =>
   options.UseSqlServer( builder.Configuration.GetConnectionString("MyDB")));
// ...
```

- To generate JWT tokens, we need to add a secret key to the configuration.
- Then, we add authentication for bearer JWT token
- Write this code before `builder.Build()`

```
// configure jwt
var key = builder.Configuration["AppSettings:Secret"];
var keyBytes = Encoding.ASCII.GetBytes(key ?? "aaaaabbbbbcccccddddd11234df4444s
builder.Services.AddAuthentication(o =>
{
    o.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
```

```
    o.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(o =>
{
    o.RequireHttpsMetadata = false;
    o.SaveToken = true;
    o.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(keyBytes),
        ValidateIssuer = false,
        ValidateAudience = false
    };
});
```

- We add authorization to enable authorization using the `[Authorize]` attribute.

```
var multiPolicyAuthorization = new AuthorizationPolicyBuilder(
      JwtBearerDefaults.AuthenticationScheme)
  .RequireAuthenticatedUser()
  .Build();

builder.Services.AddAuthorization( o => o.DefaultPolicy = multiPolicyAuthorizat
```

- Since we activate the authorization, we need to add the OpenAPI to our application.

- We modify the `builder.Services.AddOpenApi()` method as follows:

```
builder.Services.AddOpenApi(options =>
{
    options.AddDocumentTransformer((document, context, cancellationToken) =>
    {
        document.Info.Title = "Secure REST API";
        document.Info.Version = "v1";
        document.Info.Description = "A secure REST API with JWT authentication"

        // Add JWT security scheme
        document.Components ??= new OpenApiComponents();
        document.Components.SecuritySchemes["Bearer"] = new OpenApiSecuritySche
        {
            Type = SecuritySchemeType.Http,
            Scheme = "bearer",
            BearerFormat = "JWT",
            Description = "Enter JWT Bearer token"
        };

        return Task.CompletedTask;
    });
});
```

- We also add Scalar UI to our application. Write this code before `builder.Build()`

- Last, we activate Scalar UI, uthentication and authorization in our application. Write this code after `builder.Build()`

```
var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
    app.MapScalarApiReference(options =>
    {
        options.Title = "Secure REST API";
        options.Theme = ScalarTheme.Purple;
        options.ShowSidebar = true;
        options.DefaultHttpClient = new(ScalarTarget.CSharp, ScalarClient.HttpC
    });
}


// ...
app.UseHttpsRedirection();
// add these lines
app.UseAuthentication();
app.UseAuthorization();
```

5. Create the Database and Apply Migrations

- Configure your connection string and secret key of JWT token in `appsettings.json` and `appsettings.Development.json`.

```
{
  // ..
  "ConnectionStrings": {
    "MyDB": "server=localhost; database=TrainingDB; uid=tester; pwd=pass123; Tru
  },
  "AppSettings": {
    "Secret": "aaaaabbbbbcccccddddd11234df4444sd"
  }
  // ..
}
```

- Change the connection string to match your database configuration.

- You can also change the secret key of JWT token. Recommend to use a random string with 32 characters.
- Create migrations and update the database:

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

- If you have errors related to Invariant Globalization, you may disable `InvariantGlobalization` as `false` on project file, `secrestapi.csproj`.

6. Implement Registration and Login Endpoints

- Implement a registration endpoint to create new users with hashed passwords.
- In `Program.cs`, write this code for registration endpoint

```
app.MapPost("/register", async (AppDbContext dbContext, ApiUser usr) =>
{
    var user = new ApiUser
    {
        Username = usr.Username,
        Password = BC.HashPassword( usr.Password),
        Email = usr.Email,
        Name = usr.Name
    };
    dbContext.Users.Add(user);
    await dbContext.SaveChangesAsync();
    return Results.Ok();
});
```

- Implement a login endpoint that authenticates users with database authentication and returns a JWT token.

```
app.MapPost("/login", (AppDbContext dbContext, IConfiguration configuration, Us
{
    // ambil user
    var usr = dbContext.Users.Where(o => o.Username == model.UserName).FirstOrD
    if (usr != null)
    {
        if (BC.Verify(model.Password, usr.Password))
        {
            var key = configuration.GetValue<string>("AppSettings:Secret");
            var keyBytes = Encoding.ASCII.GetBytes(key ?? "aaaaabbbbbcccccddddd

            // generate token + expired
            var expiredAt = DateTime.Now.AddDays(2);
            var tokenHandler = new JwtSecurityTokenHandler();
```

```
            var tokenDescriptor = new SecurityTokenDescriptor
            {
                Subject = new ClaimsIdentity(
                        new Claim[]
                        {
                                    new Claim(ClaimTypes.Name,usr.Username)
                        }),
                Expires = expiredAt,
                SigningCredentials = new SigningCredentials(
                    new SymmetricSecurityKey(keyBytes),
                    SecurityAlgorithms.HmacSha256Signature)

            };
            var token = tokenHandler.CreateToken(tokenDescriptor);
            var userToken = new UserToken
            {
                Token = tokenHandler.WriteToken(token),
                ExpiredAt = expiredAt.ToString(),
                Message = ""
            };
            return userToken;
        }
    }

    return new UserToken { Message = "Username or password is invalid" };
});
```

Explanation:

This code snippet appears to be part of an ASP.NET Core application, specifically from an API endpoint for user authentication. It authenticates users and generates a JSON Web Token (JWT) upon successful login. Here’s a breakdown of its functionality:

- **User Authentication:**
  - The snippet begins by attempting to retrieve a user from the database (`dbContext`) based on the username provided in the `model` (likely a login request model).
  - It checks if the user exists (`usr != null`). If the user is found, it proceeds to verify the password.
- **Password Verification:**
  - The password verification is done using the BCrypt library (`BC.Verify`). It compares the provided password (`model.Password`) with the hashed password stored in the database (`usr.Password`).

- **JWT Token Generation:**
    - If the password verification is successful, the code proceeds to generate a JWT token.
    - It retrieves a secret key from the application's configuration (`configuration.GetValue<string>("AppSettings:Secret")`). This key is used to sign the JWT token and should be kept confidential.
    - A new JWT token is created with an expiration date set to 2 days from the current time (`DateTime.Now.AddDays(2)`).
    - The `JwtSecurityTokenHandler` is used to create a token with the specified claims, expiration, and signing credentials.
    - The token includes a claim for the username (`ClaimTypes.Name`), which identifies the user.
- **Token and Response Creation:**
    - A new instance of `UserToken` (presumably a custom class) is created to hold the generated JWT token, its expiration date, and a message field (empty in this case).
    - This `UserToken` instance is returned, containing the JWT token as a string (`tokenHandler.WriteToken(token)`) and its expiration date.
- **Error Handling:**
    - If the user is not found or if the password verification fails, the method returns a `UserToken` instance with a message indicating that the username or password is invalid.

- Save all change codes.

7. Implement Profile Endpoint

- Create an endpoint to retrieve the user profile, which requires a valid JWT token to access.

```
app.MapGet("/profile", [Authorize] async (HttpContext httpContext, AppDbContext
{
    var username = httpContext.User.Identity?.Name;
    var user = await dbContext.Users.FirstOrDefaultAsync(u => u.Username == use
    return user != null ? Results.Ok(new {
        user.Username,
```

```
        user.Name,
        user.Email
    }) : Results.NotFound();
});
```

8. Build and Run the Application

- Compile and run your application.
- Use `dotnet run` to start the application.
- You can also run the application with https profile:

```
dotnet run --launch-profile https
```

9. Create the `.http` File for Testing

- Create an `secrestapi.http` file for testing the registration, login, and profile endpoints.

```
@secrestapi_HostAddress = http://localhost:5124
@token =
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bmlxdWVfbmFtZSI6InVzZXIxIiwibmJmIjoxNz
AyMjg1NjMxLCJleHAiOjE3MDI0NTg0MzEsImlhdCI6MTcwMjI4NTYzMX0.mE7s2rSIIT78b-bjp-
5hhbgGEPGr1eJTww8Wg0DpRMo

### register
POST {{secrestapi_HostAddress}}/register
Accept: application/json
Content-Type: application/json

{
  "Email": "user1@email.com",
  "Password": "pass123",
  "Username": "user1",
  "Name": "User 1"
}

### login
POST {{secrestapi_HostAddress}}/login
Accept: application/json
Content-Type: application/json

{
  "Password": "pass123",
  "Username": "user1"
}

###

GET {{secrestapi_HostAddress}}/profile
Accept: application/json
Content-Type: application/json
```

```
Authorization: Bearer {{token}}

###
```

- Change the `@secrestapi_HostAddress` variable to match your application's address.

- The `@token` variable is used to store the JWT token generated by the login endpoint. This token is then used to access the profile endpoint.

10. Testing the API

- Test user registration, login, and profile access using Scalar UI or the REST Client extension in Visual Studio Code.
- For Scalar UI, navigate to `https://localhost:<port>/scalar` and try the endpoints.

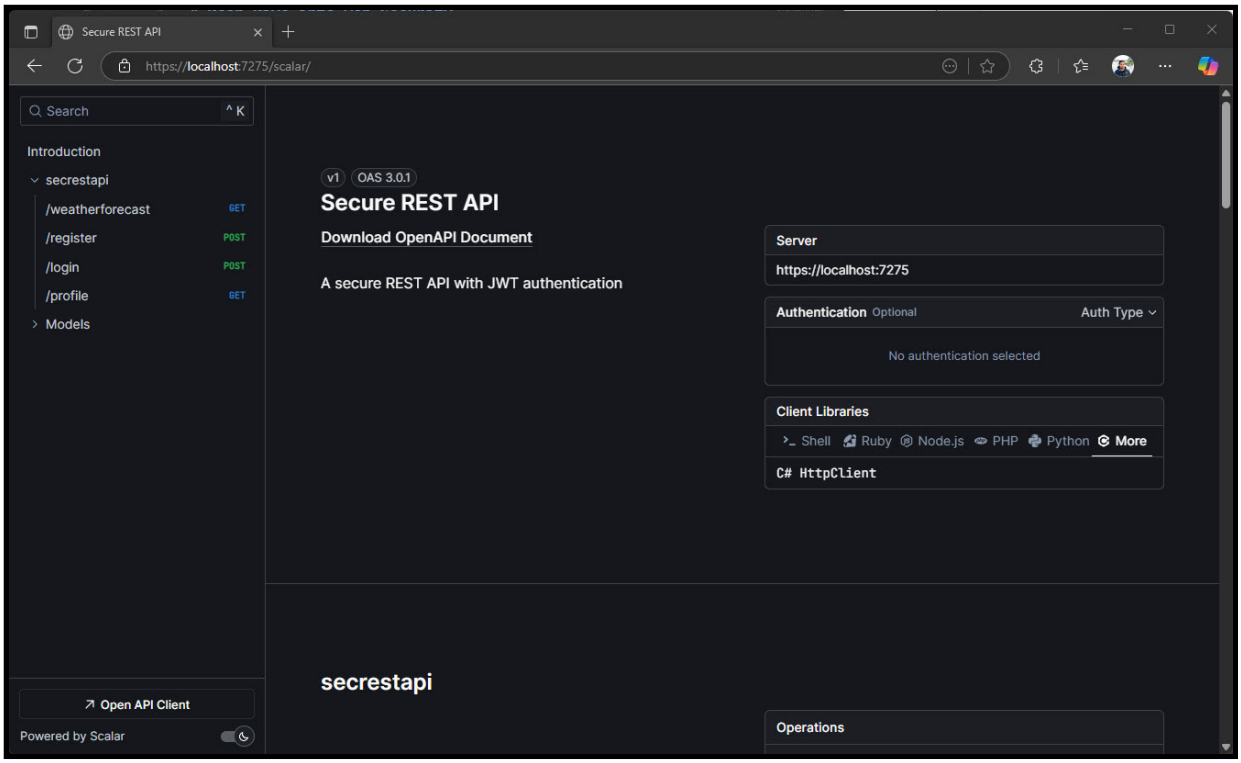


Figure 4.1 Scalar UI.

- Firstly, create user by using `register` endpoint.
- Then, login by using `login` endpoint. Copy the JWT token from the response of `login` endpoint.

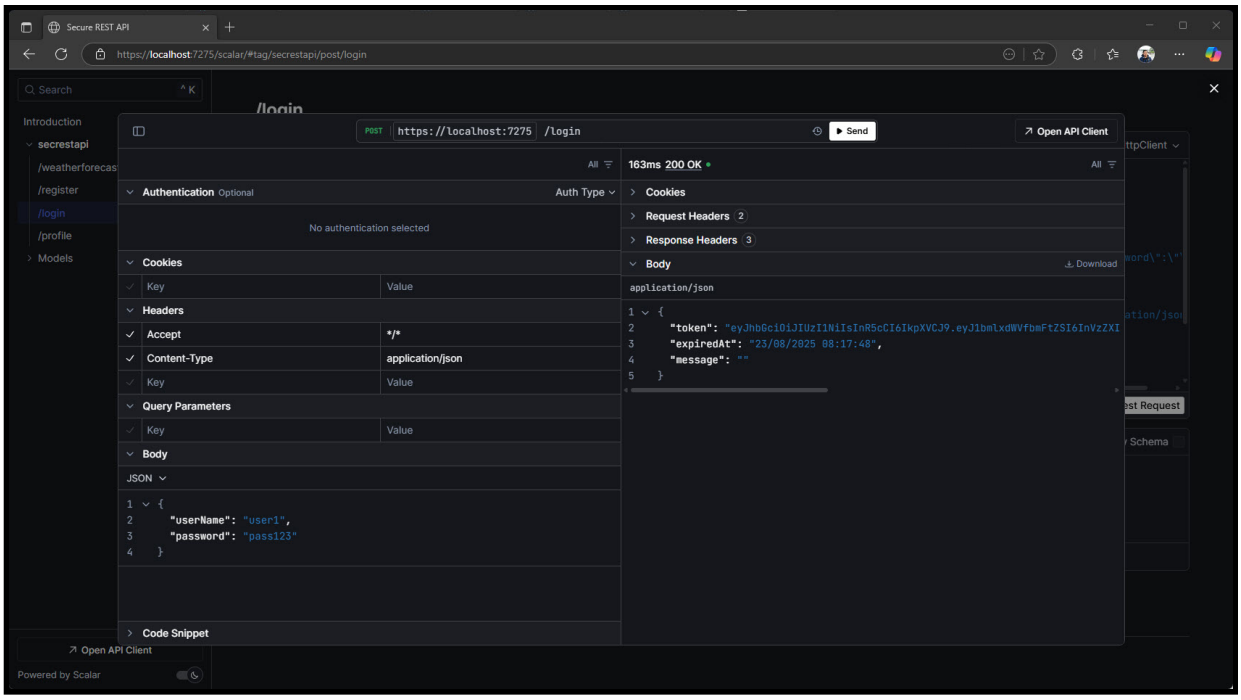


Figure 4.2 Perform login on Scalar UI.

- Finally, access the `profile` endpoint by using the JWT token.
- Add the JWT token to the `Authorization` header in Scalar UI.
- Put as `Bearer <token>` where `<token>` is the JWT token you copied from the response of `login` endpoint.
- Then, click the `Send` button to send the request to the `profile` endpoint.

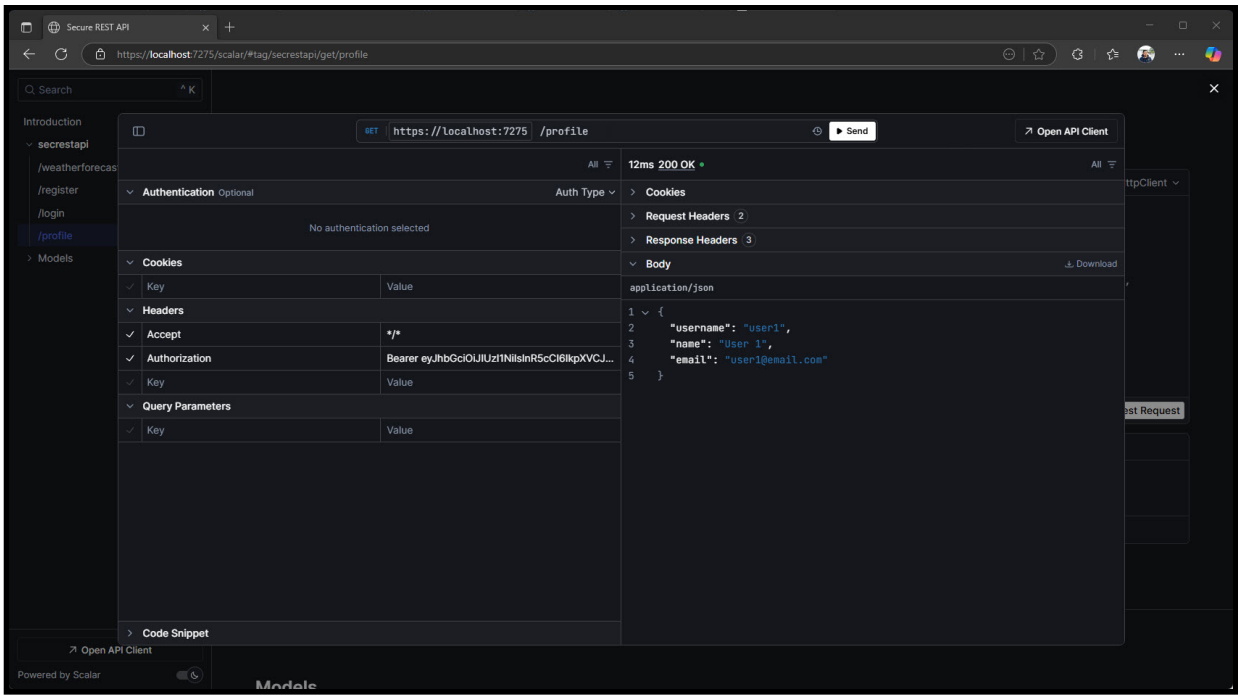


Figure 4.3 Authorize with JWT token.

- Now you can access the `profile` endpoint.
- Another option, you can set a token to Scalar Authentication.
- Click `Bearer` for Auth Type and paste the JWT token to the `Token` field.

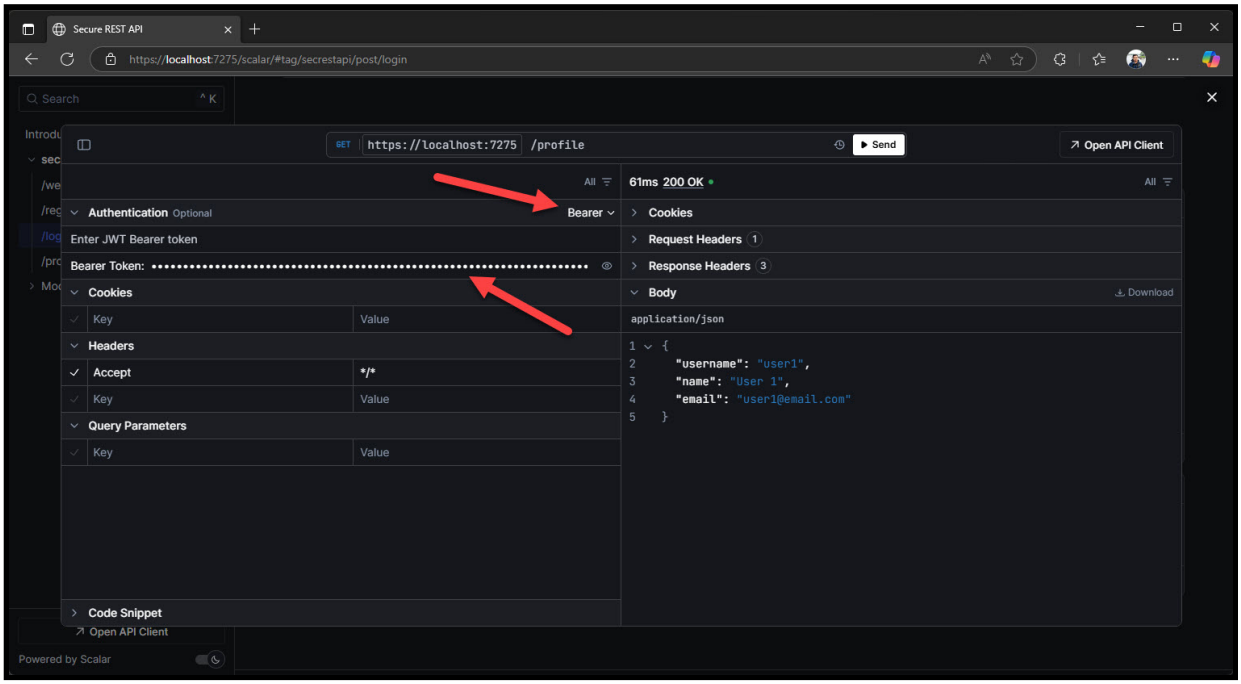


Figure 4.4 Add JWT token on Scalar UI.

- You also use the REST Client extension in Visual Studio Code to test the endpoints.
- Click the Send Request button to send the request to the endpoint.

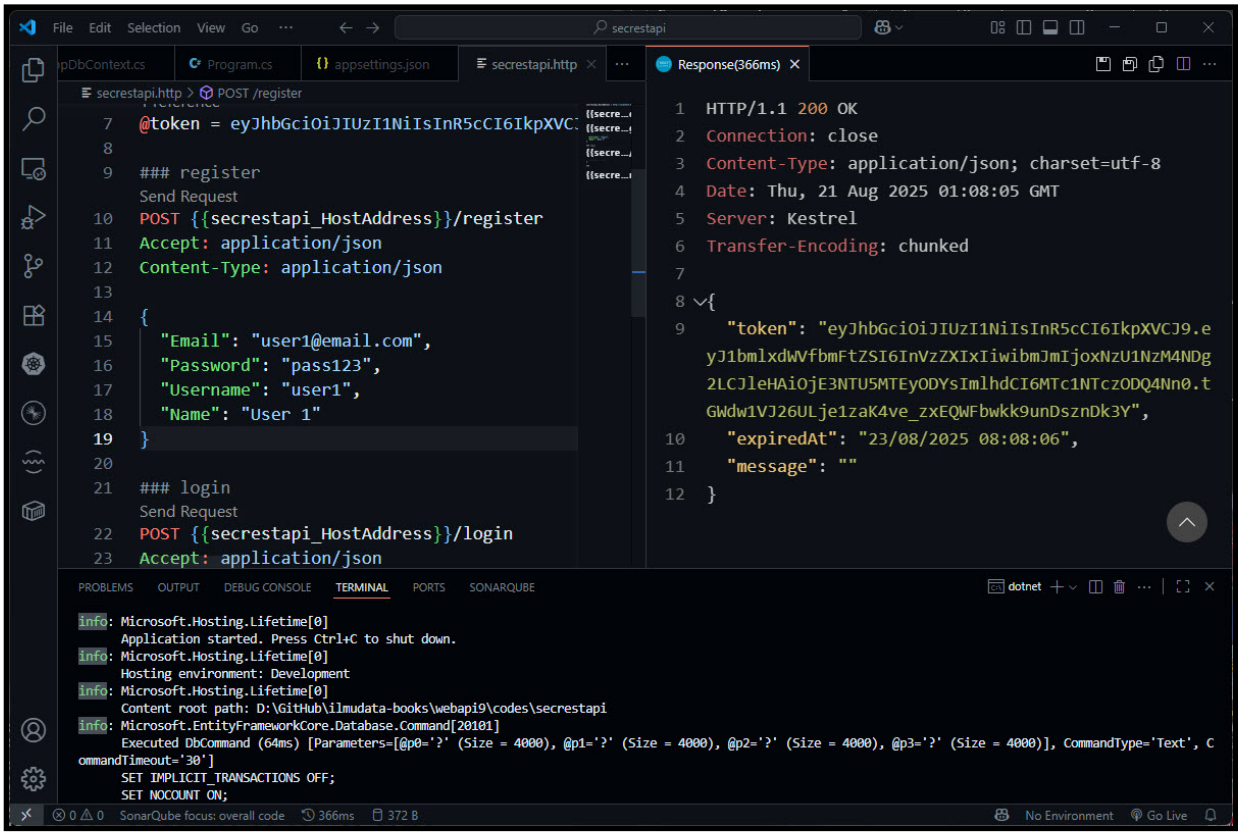


Figure 4.5 REST Client extension in Visual Studio Code.

- Afer signed in, copy `token` value from the response and paste it to the `@token` variable in the `.http` file.

4.2.4 Conclusion

This lab demonstrates how to implement a basic authentication and authorization system in an ASP.NET Core Minimal API application using Entity Framework Core for user management, bcrypt for password hashing, and JWT for token generation and validation.

4.3 Exercise 13: Role-Based Access Control (RBAC)

In this lab, we'll implement Role-Based Access Control (RBAC) in an ASP.NET Core Minimal API application using JWT tokens for authorization. This lab will guide you through creating roles, assigning them to users, and securing API endpoints based on these roles.

4.3.1 Objective

Implement Role-Based Access Control (RBAC) in an ASP.NET Core 9.0 Minimal API application using JWT tokens for authorization. This lab will guide you through creating roles, assigning them to users, and securing API endpoints based on these roles.

4.3.2 Requirements

- .NET 9.0 SDK installed
- Visual Studio Code or another code editor
- JWT authentication setup in the ASP.NET Core project
- REST Client extension in Visual Studio Code for testing

4.3.3 Lab Steps

1. Set Up the ASP.NET Core 9.0 Minimal API Project

- Open a command prompt or terminal.
- Navigate to your desired working directory or create a new one.

```
mkdir rbacapp
cd rbacapp
```

- Create a new ASP.NET Core Minimal API project:

```
dotnet new webapi
```

- Open the project in Visual Studio Code:

```
code .
```

2. Install Required Packages

- Install EF Core, bcrypt, and JWT packages:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package BCrypt.Net-Next
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

- Install the Scalar UI package:

```
dotnet add package Scalar.AspNetCore
```

3. Define Models and DbContext

- Create a `Models` folder and define a `ApiUser` model on `ApiUser.cs` file:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace rbacapp.Models;

public class ApiUser
{
    [Key]
    [DatabaseGenerated( DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    public string Username { get; set; } = "";
    public string Password { get; set; } = "";
    public string Name { get; set; } = "";
    public string Email { get; set; } = "";
}
```

We also create DTOs (Data Transfer Objects) for user registration and login. These DTOs are used to transfer data between the client and server. - Inside `Models` folder, write `UserLogin.cs` file and write the following code:

```
namespace rbacapp.Models;

public class UserLogin
{
    public string UserName { get; set; } = "";
    public string Password { get; set; } = "";
}
```

- We also create `UserToken.cs` file as DTO for JWT token. This DTO is used to transfer JWT token between the client and server.

```
namespace rbacapp.Models;

public class UserToken
```

```
{
    public string Token { set; get; } = "";
    public string ExpiredAt { set; get; } = "";
    public string Message { set; get; } = "";
}
```

- To implement RBAC, we need to create a `Role` model and a `UserRole` model.
- Every user can have one or more roles.
- The `Role` model represents a role in the application, such as “Admin” or “Manager”.
- The following code snippet shows the `Role` model, `Role.cs` in the `Models` folder:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace rbacapp.Models;

public class Role
{
    [Key]
    [DatabaseGenerated( DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public List<ApiUser>? Users { get; set; }
}
```

- The following code snippet shows the `UserRole` model, `UserRole.cs` in the `Models` folder:

```
using Microsoft.AspNetCore.Identity;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace rbacapp.Models;

public class UserRole
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    public ApiUser? User { get; set; } = null!;
    public Role? Role { get; set; } = null!;
}
```

- Inside a `Models` folder and implement `AppDbContext` on `AppDbContext.cs` file:

```
using Microsoft.EntityFrameworkCore;

namespace rbacapp.Models;

public class AppDbContext : DbContext
{
    public AppDbContext( DbContextOptions<AppDbContext> options) : base(options

    public DbSet<ApiUser> Users { get; set; }
    public DbSet<Role> Roles { get; set; }
    public DbSet<UserRole> UserRoles { get; set; }
}
```

4. Configure DbContext and JWT in Program.cs

- In general, we use the same configuration as the previous lab.
- Configure DbContext and add JWT authentication in `Program.cs`.
- In `Program.cs`, configure the DbContext:

```
using rbacapp.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;
using Microsoft.IdentityModel.Tokens;
using System.Text;
using BC = BCrypt.Net.BCrypt;
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using Microsoft.OpenApi.Models;
using Microsoft.Extensions.Options;
using Scalar.AspNetCore;
// ...

builder.Services.AddDbContext<AppDbContext>(options =>
   options.UseSqlServer( builder.Configuration.GetConnectionString("MyDB")));
// ...
```

- To generate JWT tokens, we need to add a secret key to the configuration.
- Then, we add authentication for bearer JWT token
- Write this code before `builder.Build()`

```
// configure jwt
var key = builder.Configuration["AppSettings:Secret"];
var keyBytes = Encoding.ASCII.GetBytes(key ?? "aaaaabbbbbcccccddddd11234df4444s
builder.Services.AddAuthentication(o =>
{
    o.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    o.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(o =>
{
    o.RequireHttpsMetadata = false;
    o.SaveToken = true;
    o.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(keyBytes),
        ValidateIssuer = false,
        ValidateAudience = false,
    };
});
```

- We add authorization to enable authorization using the `[Authorize]` attribute.

```
var multiPolicyAuthorization = new AuthorizationPolicyBuilder(
   JwtBearerDefaults.AuthenticationScheme )
  .RequireAuthenticatedUser()
  .Build();
builder.Services.AddAuthorization(o =>
{
    o.DefaultPolicy = multiPolicyAuthorization;
});
```

- Since we activate the authorization, we need to add the OpenAPI to our application.

- We modify the `builder.Services.AddOpenApi()` method as follows:

```
builder.Services.AddOpenApi(options =>
{
    options.AddDocumentTransformer((document, context, cancellationToken) =>
    {
        document.Info.Title = "Secure REST API";
        document.Info.Version = "v1";
        document.Info.Description = "A secure REST API with JWT authentication"

        // Add JWT security scheme
        document.Components ??= new OpenApiComponents();
        document.Components.SecuritySchemes["Bearer"] = new OpenApiSecuritySche
        {
            Type = SecuritySchemeType.Http,
            Scheme = "bearer",
            BearerFormat = "JWT",
```

```
            Description = "Enter JWT Bearer token"
        };

        return Task.CompletedTask;
    });
});
```

- We also add Scalar UI to our application. Write this code before `builder.Build()`

- Last, we activate Scalar UI, uthentication and authorization in our application. Write this code after `builder.Build()`

```
var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
    app.MapScalarApiReference(options =>
    {
        options.Title = "Secure REST API";
        options.Theme = ScalarTheme.Purple;
        options.ShowSidebar = true;
        options.DefaultHttpClient = new(ScalarTarget.CSharp, ScalarClient.HttpC
    });
}



// ...
app.UseHttpsRedirection();
// add these lines
app.UseAuthentication();
app.UseAuthorization();
```

5. Create the Database and Apply Migrations

- Configure your connection string and secret key of JWT token in `appsettings.json` and `appsettings.Development.json`.

```
{
  // ..
  "ConnectionStrings": {
    "MyDB": "server=localhost; database=Training2DB; uid=tester; pwd=pass123; T
  },
  "AppSettings": {
    "Secret": "aaaaabbbbbcccccddddd11234df4444sd"
  }
  // ..
}
```

- Change the connection string to match your database configuration.
- You can also change the secret key of JWT token. Recommend to use a random string with 32 characters.
- Create migrations and update the database:

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

- If you have errors related to Invariant Globalization, you may disable `InvariantGlobalization` as `false` on project file, `rbacapp.csproj`.

6. Set up Roles

- In this lab, we will create two roles: “Admin” and “Manager”.
- We create `/setuproles` API endpoint to create roles into the database.

```
app.MapGet("/setuproles", async (AppDbContext dbContext) =>
{
    var roles = dbContext.Roles.ToList();
    if(roles.Count <= 0)
    {
        dbContext.Roles.Add(new Role { Name = "Admin" });
        dbContext.Roles.Add(new Role { Name = "Manager" });
        await dbContext.SaveChangesAsync();
    }
    return Results.Ok(new { Message = "Roles was created"});
});
```

- To address duplicate roles, we check if the roles already exist in the database. If not, we create them.

7. Implement Registration and Login Endpoints

- Implement a registration endpoint to create new users with hashed passwords.
- In `Program.cs`, write this code for registration endpoint

```
app.MapPost("/register", async (AppDbContext dbContext, ApiUser usr) =>
{
    var user = new ApiUser
    {
        Username = usr.Username,
```

```
        Password = BC.HashPassword( usr.Password),
        Email = usr.Email,
        Name = usr.Name
    };
    dbContext.Users.Add(user);
    await dbContext.SaveChangesAsync();
    return Results.Ok();
});
```

- Implement a login endpoint that authenticates users with database authentication and returns a JWT token.

```
app.MapPost("/login", (AppDbContext dbContext, IConfiguration configuration, Us
{
    // ambil user
    var usr = dbContext.Users.Where(o => o.Username == model.UserName).FirstOrD
    if (usr != null)
    {
        if (BC.Verify(model.Password, usr.Password))
        {
            List<Claim> claims = new List<Claim>();
            // get roles by userid
            var roles = from userRole in dbContext.UserRoles
                        join role in dbContext.Roles on userRole.Role!.Id equal
                        where userRole.User!.Id == usr.Id
                        select role.Name;

            foreach (var roleName in roles)
            {
                claims.Add(new Claim(ClaimTypes.Role, "" + roleName));
            }
            claims.Add(new Claim(ClaimTypes.Name, usr.Username));

            // generate token
            var key = configuration.GetValue<string>("AppSettings:Secret");
            var keyBytes = Encoding.ASCII.GetBytes(key ?? "aaaaabbbbbcccccddddd
            var symKey = new SymmetricSecurityKey(keyBytes); // Use a secure ke
            var creds = new SigningCredentials( symKey, SecurityAlgorithms.Hmac
            var expiry = DateTime.Now.AddDays(2);

            var token = new JwtSecurityToken(
                claims: claims,
                expires: expiry,
                signingCredentials: creds
            );

            var userToken = new UserToken
            {
                Token = new JwtSecurityTokenHandler().WriteToken(token),
                ExpiredAt = expiry.ToString(),
                Message = ""
            };
            return userToken;
        }
    }
```

```
    return new UserToken { Message = "Username or password is invalid" };
});
```

- Modify secret key `aaaaabbbbbcccccddddd11234df4444sd` with your secret key.
- After user login, we get the user's roles from the database.
- Use these roles to create claims for the JWT token.
- We put user roles and username as claims in the JWT token.

8. Assign a Role to an User

- In this lab, we don't privide a dashboard to manage roles and users.
- We provide an API endpoint to assign a role to an user.
- We create `/addrole/{username}/role/{rolename}` API endpoints to assign a role to an user.
- `{username}` is existing username in the database.
- `{rolename}` is existing role name in the database.
- The following is a complete code snippet for the `/addrole/{username}/role/{rolename}` endpoint:

```
// add role
app.MapGet("/addrole/{username}/role/{rolename}",
    async (AppDbContext db, string username, string rolename) =>
{
    var role = db.Roles.Where(a => a.Name == rolename).FirstOrDefault();
    if (role is null) return Results.NotFound();

    var user = db.Users.Where(a => a.Username == username).FirstOrDefault();
    if (user is null) return Results.NotFound();

    var userRole = new UserRole { Role = role, User = user };
    await db.UserRoles.AddAsync(userRole);

    await db.SaveChangesAsync();

    return Results.Ok($"Role has been added. ID: {userRole.Id}");
});
```

- After the role is assigned to the user, we can see the role in the database.
- Call the `/addrole/{username}/role/{rolename}` endpoint to assign a role to an user.

9. Resource Endpoints

- We provide some resource endpoints to test the role-based access control.
- We define `/profile` endpoint to get user profile.
- Any user can access this endpoint but the user must have a valid JWT token.
- This endpoint is secured by JWT token.
- The following is a complete code snippet for the `/profile` endpoint:

```
app.MapGet("/profile", [Authorize] async (HttpContext httpContext, AppDbContext
{
    var username = httpContext.User.Identity?.Name;
    var user = await dbContext.Users.FirstOrDefaultAsync(u => u.Username == use
    return user != null ? Results.Ok(new {
        user.Username,
        user.Name,
        user.Email
    }) : Results.NotFound();
});
```

- ASP.NET Core provides a built-in `[Authorize]` attribute to secure an endpoint.
- User can access this endpoint if the user has a valid JWT token.
- We also define `/admin`, `/manager` and `/adminmanager` endpoints to test the role-based access control.
- `/admin` endpoint is secured by JWT token and only can be accessed by user with “Admin” role.
- We use `[Authorize(Roles = "Admin")]` attribute to secure this endpoint.

```
app.MapGet("/admin", [Authorize(Roles = "Admin")]() =>
{
    return Results.Ok(new
    {
      Message="This content is only for admin"
    });
});
```

- `/manager` endpoint is secured by JWT token and only can be accessed by user with “Manager” role.
- We use `[Authorize(Roles = "Manager")]` attribute to secure this endpoint.

```
app.MapGet("/manager", [Authorize(Roles = "Manager")] () =>
{
    return Results.Ok(new
    {
        Message = "This content is only for manager"
    });
});
```

- `/adminmanager` endpoint is secured by JWT token and only can be accessed by user with “Admin” or “Manager” role.
- We use `[Authorize(Roles = "Admin,Manager")]` attribute to secure this endpoint.

```
app.MapGet("/adminmanager",[Authorize(Roles = "Admin,Manager")]  () =>
{
    return Results.Ok(new
    {
        Message = "This content is only for admin and manager"
    });
});
```

10. Create the `.http` File for Testing

- Prepare an `rbacapp.http` file with requests for the role-secured endpoints.

```
@rbacapp_HostAddress = http://localhost:5289
@token = <token>
```

- Change the `@rbacapp_HostAddress` variable to match your application’s address.
- The `@token` variable is used to store the JWT token generated by the login endpoint. This token is then used to access the profile endpoint.

- We can call the `/setuproles` endpoint to create roles in the database.

```
### setup roles
GET {{rbacapp_HostAddress}}/setuproles
Accept: application/json
Content-Type: application/json
```

- We can call the `/register` endpoint to create a new user.

```
### register
POST {{rbacapp_HostAddress}}/register
Accept: application/json
Content-Type: application/json

{
  "Email": "user3@email.com",
  "Password": "pass123",
  "Username": "user3",
  "Name": "User 3"
}
```

- You may change the username, password, email, and name.
- We can call the `/login` endpoint to get JWT token.

```
### login
POST {{rbacapp_HostAddress}}/login
Accept: application/json
Content-Type: application/json

{
  "Password": "pass123",
  "Username": "user3"
}
```

- Change the username and password with your username and password.
- After user login, we get the JWT token from the response.
- Copy the JWT token and paste it to the `@token` variable.
- We can call resource endpoints to perform role-based access control.

```
###
GET {{rbacapp_HostAddress}}/profile
Accept: application/json
Content-Type: application/json
Authorization: Bearer {{token}}

###
GET {{rbacapp_HostAddress}}/manager
Accept: application/json
Content-Type: application/json
```

```
Authorization: Bearer {{token}}

###
GET {{rbacapp_HostAddress}}/admin
Accept: application/json
Content-Type: application/json
Authorization: Bearer {{token}}

###
GET {{rbacapp_HostAddress}}/adminmanager
Accept: application/json
Content-Type: application/json
Authorization: Bearer {{token}}
```

- Save all change codes.

11. Build and Run the Application

- Compile and run your application.
- Use `dotnet run` to start the application.
- You can also run the application with https profile:

```
dotnet run --launch-profile https
```

12. Testing the API with Roles

- Test the role-based secured endpoints using JWT tokens with appropriate role claims.
- Use a REST Client or Postman to send requests with JWT tokens to your endpoints.
- If we use Scalar UI, we can test the endpoints as follows.
- Open the Scalar UI at `https://localhost:<port>/scalar`.

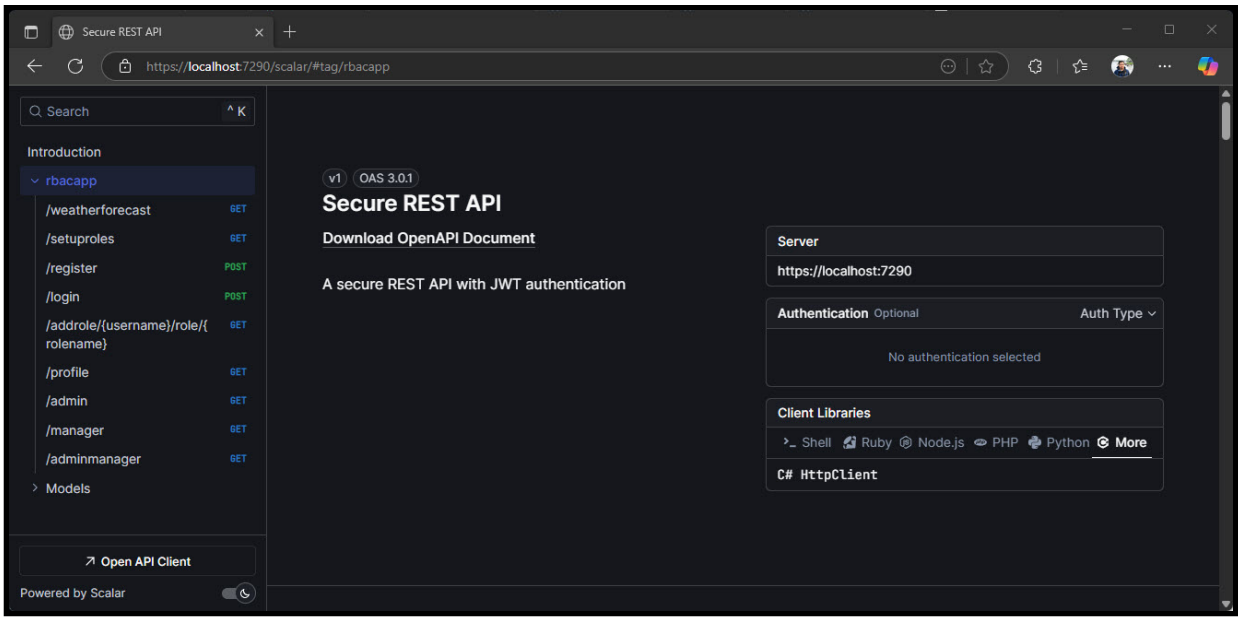


Figure 4.6 Scalar UI from RBAC app.

- You may need to change the port number.
- Firstly, call the `/setuproles` endpoint to create roles in the database.
- Then, call the `/register` endpoint to create a new user.
- Create three users with different roles:
    - User1 with “Admin” role
    - User2 with “Manager” role
    - User3 with no role (or any other role)
- Assign a role to the user by calling the `/addrole/{username}/role/{rolename}` endpoint.

RBAC Testing Steps:

- After user login, we get the JWT token from the response.
- Copy the JWT token and paste it to the Authorize input field.
- Now we can call resource endpoints to perform role-based access control.
- Try to sign in an user with **Admin** role.
- Access the `/profile`, `/manager` and `/adminmanager` endpoints.

- You should be able to access the `/profile` and `/admin` endpoints but not the `/adminmanager` endpoint.
- Now try to access `/manager` endpoint.
- You should get an error message (HTTP Code 403) because you don't have the **Manager** role.

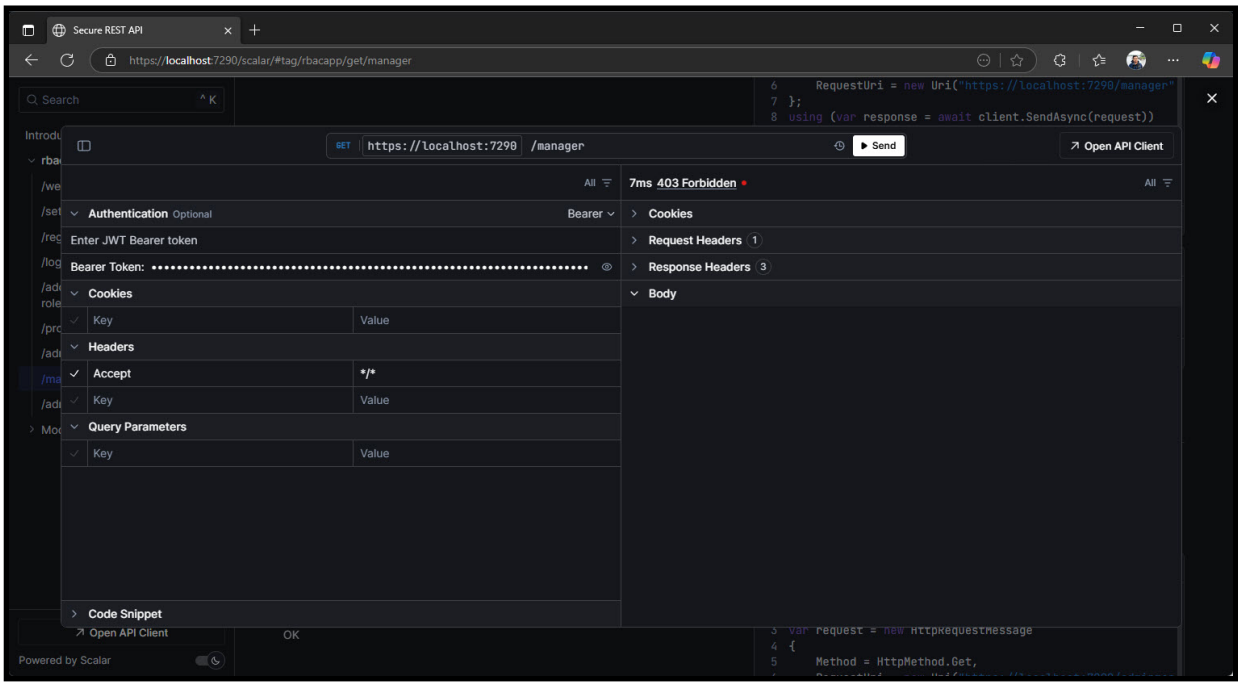


Figure 4.7 Access denied on Scalar UI from RBAC app.

- You can explore another endpoints on Scalar UI.
- For REST Client, we can test the endpoints as follows.
- Open `rbacapp.http` file.
- Click `Send Request` for `/login`.

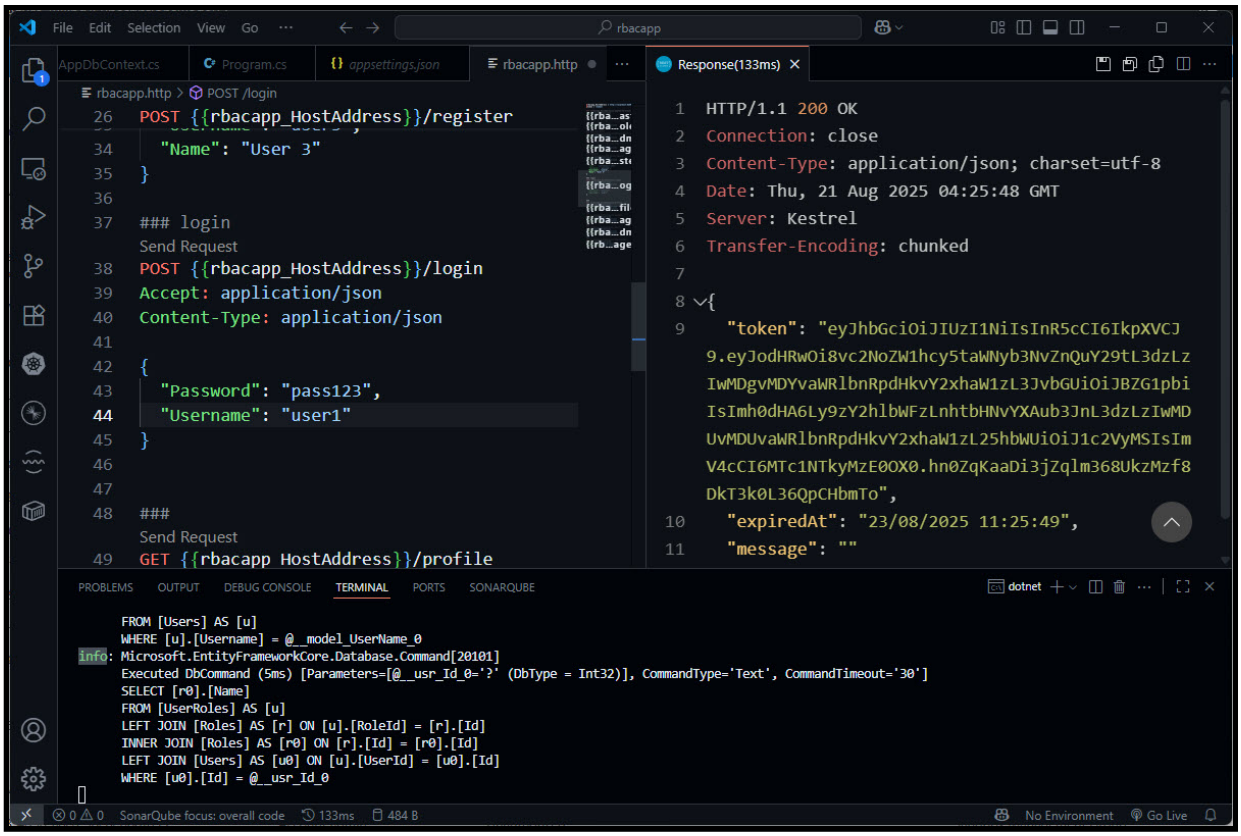


Figure 4.8 Perform calling login.

- You have a token after performed a login.
- Copy this token to `@token` variable.
- Try to access resource endpoints.
- If you signed with “Admin” role, then you access `/manager` endpoint.
- You will get error “HTTP code 403” because you are not “Manager” role.

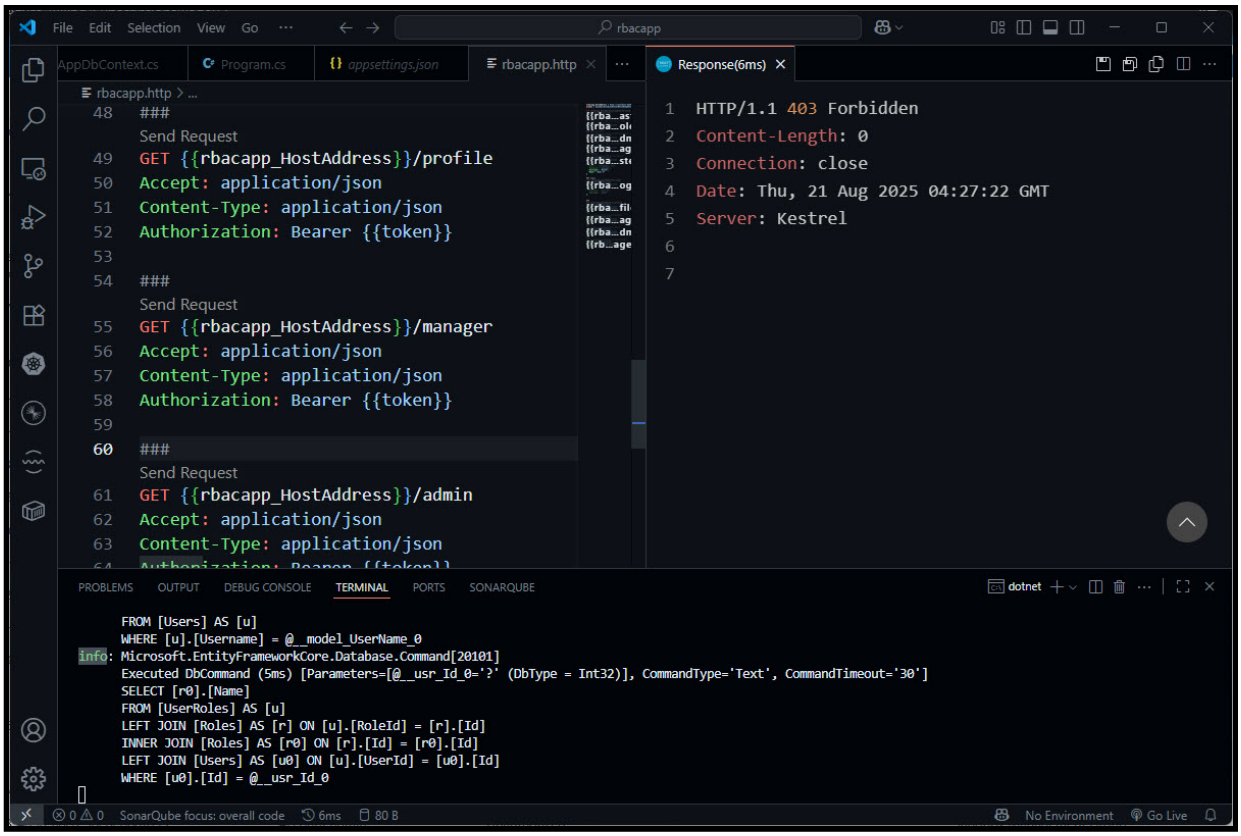


Figure 4.9 Access denied on REST API client from RBAC app.

- Try to perform any endpoint.

4.3.4 Conclusion

This lab provides a hands-on experience in implementing Role-Based Access Control in an ASP.NET Core Minimal API application using JWT tokens. By completing this lab, you'll learn how to create roles, assign them to users, and secure API endpoints based on these roles, ensuring that only authorized users can access specific resources.

4.4 Exercise 14: Data Privacy and Protection

In this lab, we learn how to protect sensitive data in an ASP.NET Core Minimal API application using Entity Framework Core and ASP.NET Core Data Protection APIs. We'll implement data protection for an `Employee` model by encrypting sensitive fields such as email, phone, and birthdate before

storing them in SQL Server. We'll also mask these fields when retrieving the list of employees.

4.4.1 Objective

Implement data protection for an `Employee` model in an ASP.NET Core 9.0 Minimal API application. Encrypt sensitive fields such as email, phone, and birthdate before storing in SQL Server, and mask these fields when retrieving the list of employees.

4.4.2 Requirements

- .NET 9.0 SDK installed
- Visual Studio Code or another code editor
- SQL Server installed and accessible
- Entity Framework Core
- ASP.NET Core Data Protection APIs

4.4.3 Lab Steps

1. Set Up the ASP.NET Core 9.0 Minimal API Project

- Open a command prompt or terminal.
- Navigate to your desired working directory or create a new one.

```
mkdir privdata
cd privdata
```

- Create a new ASP.NET Core Minimal API project:

```
dotnet new webapi
```

- Open the project in Visual Studio Code:

```
code .
```

2. Install Required Packages

- Install EF Core and data protection packages:

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet add package Microsoft.AspNetCore.DataProtection
dotnet add package Microsoft.AspNetCore.DataProtection.EntityFrameworkCore
```

- Install the Scalar UI package:

```
dotnet add package Scalar.AspNetCore
```

3. Define the `Employee` Model

- In the `Models` folder, create `Employee.cs`:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace privdata.Models;

public class Employee
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public string Email { get; set; } = ""; // To be encrypted
    public string Phone { get; set; } = "";// To be encrypted
    public string Birthdate { get; set; } = "";// To be encrypted
}
```

4. Implement `AppDbContext`

- We use SQL Server to store data protection keys.

- In the `Models` folder, create `AppDbContext.cs`:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.DataProtection.EntityFrameworkCore;

namespace privdata.Models;

public class AppDbContext : DbContext, IDataProtectionKeyContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)

    public DbSet<Employee> Employees { get; set; }
    public DbSet<DataProtectionKey> DataProtectionKeys { get; set; }
}
```

- In `AppDbContext.cs`, implement `IDataProtectionKeyContext` to store data protection keys in the database.

- `DataProtectionKey` is a built-in model for storing data protection keys in the database.

- In `Program.cs`, add and configure `AppDbContext` as previously described.

```
using privdata.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.DataProtection;
using Scalar.AspNetCore;

var builder = WebApplication.CreateBuilder(args);

// ...
// add database
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer( builder.Configuration.GetConnectionString("MyDB")));
```

5. Configure Data Protection Services

- Since we store data protection keys in the database, we need to configure data protection services.

- We create a custom data protection provider to handle data protection keys in the database.

- Create `SqlServerDataProtectionProvider.cs` file inside `Models` folder and write these codes

```
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.EntityFrameworkCore;

namespace privdata.Models;
public class SqlServerDataProtectionProvider : IDataProtectionProvider
{
    private readonly AppDbContext _dbContext;

    public SqlServerDataProtectionProvider(AppDbContext dbContext)
    {
        _dbContext = dbContext;
    }

    public IDataProtector CreateProtector(string purpose)
    {
        // Implement logic to retrieve or create a new key from the database
        var key = _dbContext.DataProtectionKeys.Where(o => o.FriendlyName == pu
```

```
        if(key == null)
        {
            key = CreateNewKey(purpose);
        }
        var protector = DataProtectionProvider.Create(key.FriendlyName ?? purpo
            .CreateProtector(purpose);
        return protector;
    }

    private DataProtectionKey CreateNewKey(string purpose)
    {
        var key = new DataProtectionKey { FriendlyName = purpose };
        _dbContext.DataProtectionKeys.Add(key);
        _dbContext.SaveChanges();
        return key;
    }
}
```

- `CreateProtector()` is used to create a data protector for a specific purpose.
- `CreateNewKey()` is used to create a new key if the key doesn't exist in the database.
- We also create a `SensitiveDataService` class to handle encryption and masking logic.
- Create `SensitiveDataService.cs` file inside `Models` folder
- In `Program.cs`, add and configure Data Protection services as previously described.

```
// ...
// Add data protection services
builder.Services.AddDataProtection()
    .PersistKeysToDbContext<AppDbContext>();

// add custom data protection provider
builder.Services.AddTransient<IDataProtectionProvider, SqlServerDataProtectionP

// ...
```

6. Implement Encryption and Masking Logic

- We create a `SensitiveDataService` class to handle encryption and masking logic.

- Create `SensitiveDataService.cs` file inside `Models` folder
- Following is a complete code snippet for the `SensitiveDataService` class:

```
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.DataProtection.EntityFrameworkCore;

namespace privdata.Models;

public class SensitiveDataService
{
    private readonly IDataProtector _protector;

    public SensitiveDataService(IDataProtectionProvider provider)
    {
        _protector = provider.CreateProtector("EmployeeDataProtector");
    }

    public Employee EncryptEmployeeData(Employee employee)
    {
        employee.Email = _protector.Protect(employee.Email);
        employee.Phone = _protector.Protect(employee.Phone);
        employee.Birthdate = _protector.Protect(employee.Birthdate);
        return employee;
    }

    public Employee MaskEmployeeData(Employee employee)
    {

        employee.Email = MaskEmail(_protector.Unprotect(employee.Email));
        employee.Phone = MaskPhone(_protector.Unprotect(employee.Phone));
        employee.Birthdate = "*****"; // Simple mask for birthdate
        return employee;
    }

      public static string MaskEmail(string email)
    {
        var atIndex = email.IndexOf('@');
        if (atIndex == -1 || atIndex == 0) return email; // Invalid or empty ema

        var accountPart = email.Substring(0, atIndex);
        var domainPart = email.Substring(atIndex);

        var maskedLength = accountPart.Length / 2;
        var maskedPart = new string('*', maskedLength);
        var visiblePart = accountPart.Substring(maskedLength);

        return maskedPart + visiblePart + domainPart;
    }

    private string MaskPhone(string phone)
    {
        // Implement phone masking logic
        return "*******" + phone.Substring(phone.Length - 4); // Example
```

```
    }
}
```

- `EncryptEmployeeData()` is used to encrypt sensitive employee data before storing it in the database.
- We encrypt the email, phone, and birthdate fields using the data protector.
- `MaskEmployeeData()` is used to mask sensitive employee data when sending it to clients.
- We mask the email and phone fields using the `MaskEmail()` and `MaskPhone()` methods.
- `MaskEmail()` masks the email address by replacing the first half of the account name with asterisks `*****`.
- `MaskPhone()` masks the phone number by replacing the first 7 digits with asterisks `*******`.
- For birthdate, we simply replace the value with asterisks `*****`.
- Since we use the data protector to encrypt and decrypt sensitive data, we need to inject the `SensitiveDataService` class into the API endpoints.

```
...
builder.Services.AddTransient<SensitiveDataService>();
...
```

7. Create the Database and Apply Migrations

- Configure your connection string and secret key of JWT token in `appsettings.json` and `appsettings.Development.json`.

```
{
  // ..
  "ConnectionStrings": {
    "MyDB": "server=localhost; database=Training3DB; uid=tester; pwd=pass123; T
  },
  "AppSettings": {
    "Secret": "aaaaabbbbbcccccddddd11234df4444sd"
  }
```

```
  // ..
}
```

- Change the connection string to match your database configuration.
- You can also change the secret key of JWT token. Recommend to use a random string with 32 characters.
- Create migrations and update the database:

```
dotnet ef migrations add InitialCreate
dotnet ef database update
```

- If you have errors related to Invariant Globalization, you may disable `InvariantGlobalization` as `false` on project file, `privdata.csproj`.

8. Create API Endpoints

- Implement API endpoints for adding and retrieving `Employee` data:

```
app.MapPost("/employees", (AppDbContext dbContext, SensitiveDataService service,
{
    dbContext.Employees.Add(service.EncryptEmployeeData(employee));
    dbContext.SaveChanges();
    return Results.Ok();
});

app.MapGet("/employees", (AppDbContext dbContext, SensitiveDataService service)
{
    var employees = dbContext.Employees.AsEnumerable().Select(service.MaskEmploy
    return Results.Ok(employees);
});
```

9. Create the `.http` File for Testing

- Create a `privdata.http` file with requests for adding and retrieving employee data.

```
@privdata_HostAddress = http://localhost:5252

### Create Employee
POST {{privdata_HostAddress}}/employees
Accept: application/json
Content-Type: application/json

{
  "name": "user 5",
```

```
  "email": "user5@email.com",
  "phone": "08134455483",
  "birthdate": "11-des-1996"
}

### Get All Employees
GET {{privdata_HostAddress}}/employees
Accept: application/json
```

10. Configure Scalar UI

- In `Program.cs`, configure Scalar UI to provide a user-friendly interface for testing the API endpoints.

```
using Scalar.AspNetCore;


...
builder.Services.AddOpenApi();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
    app.MapScalarApiReference();
}
```

11. Build and Run the Application

- Compile and run your application.
- Use `dotnet run` to start the application.
- You can also run the application with https profile:

```
dotnet run --launch-profile https
```

12. Testing the API

- We use Scalar UI to test the API.
- Open the Scalar UI at `https://localhost:<port>/scalar`.

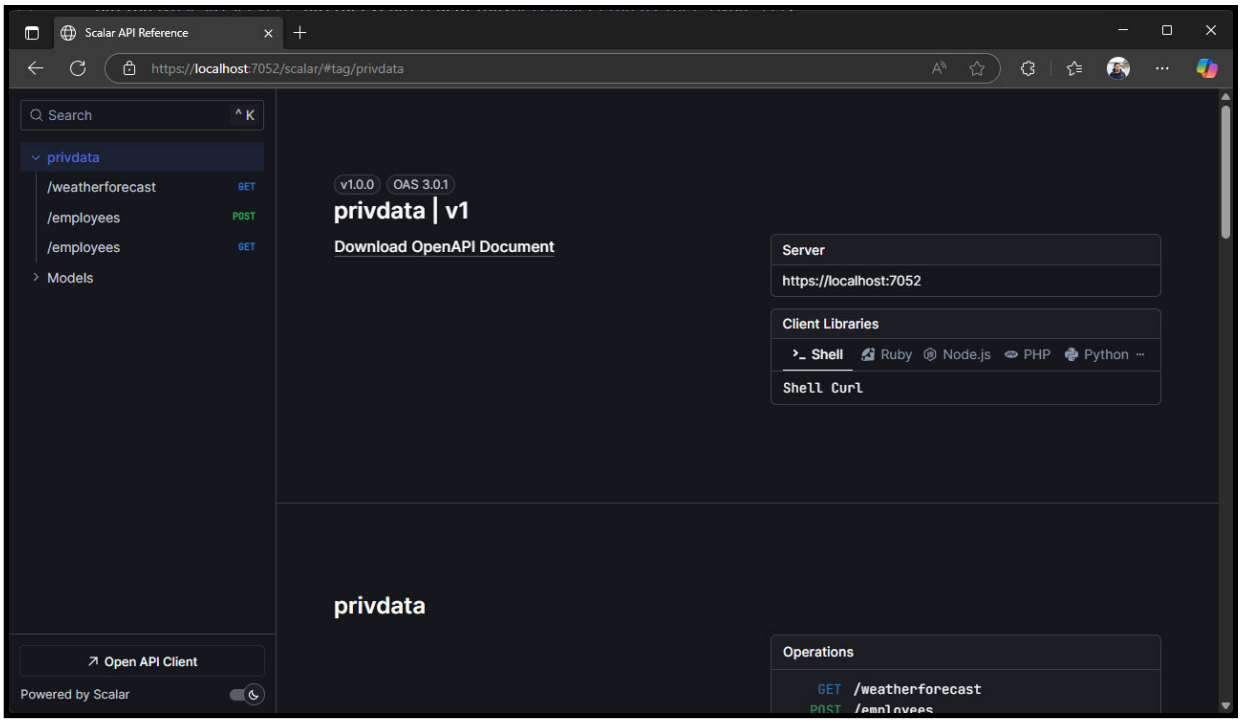


Figure 4.10 Scalar UI from PrivData app.

- Try to add some employees.
- Then, open the database and see the employee data.

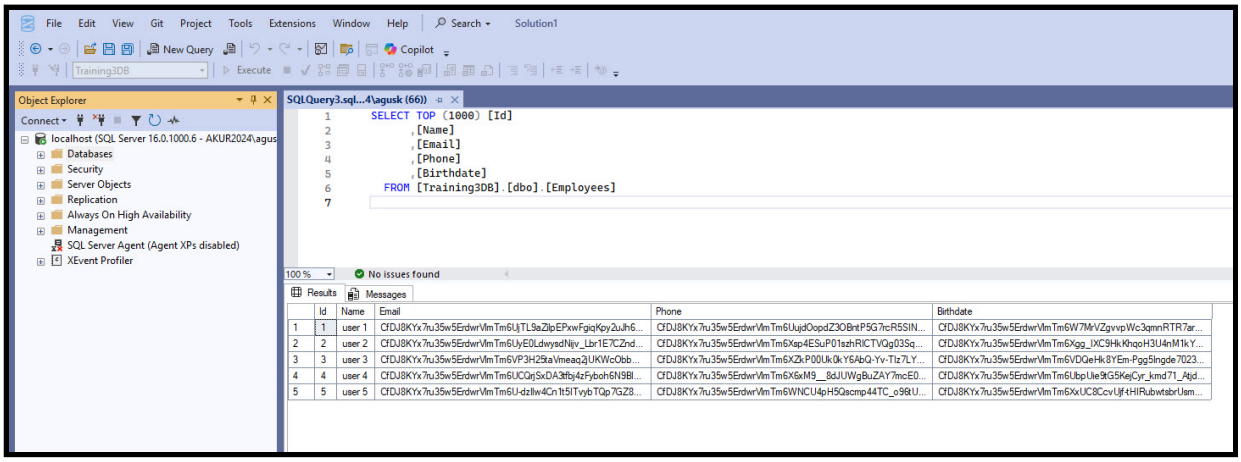


Figure 4.11 Employee data in the database.

- You can see the email, phone, and birthdate fields are encrypted.
- Now, try to get all employees.
- You can see the email, birthdate and phone fields are masked.

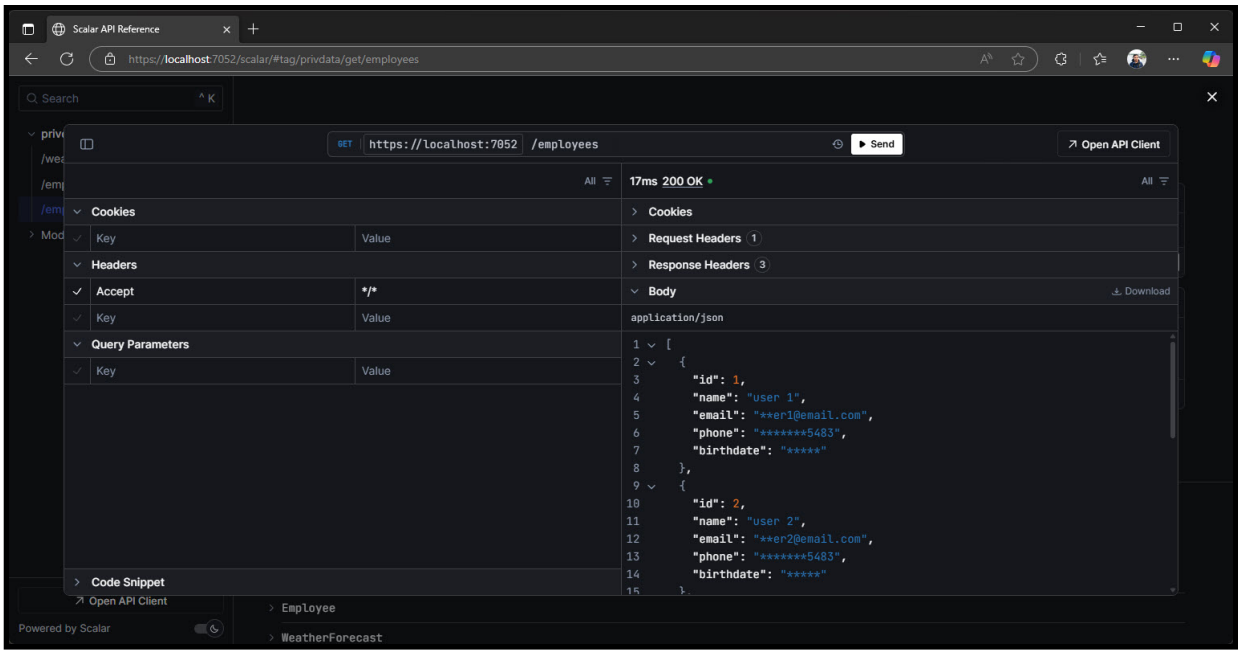


Figure 4.12 Employee data in the database.

- We also use the REST Client extension to test adding and retrieving encrypted and masked employee data.
- Click `Send Request` for POST `/employees` to add an employee.
- Click `Send Request` for GET `/employees` to get all employees.

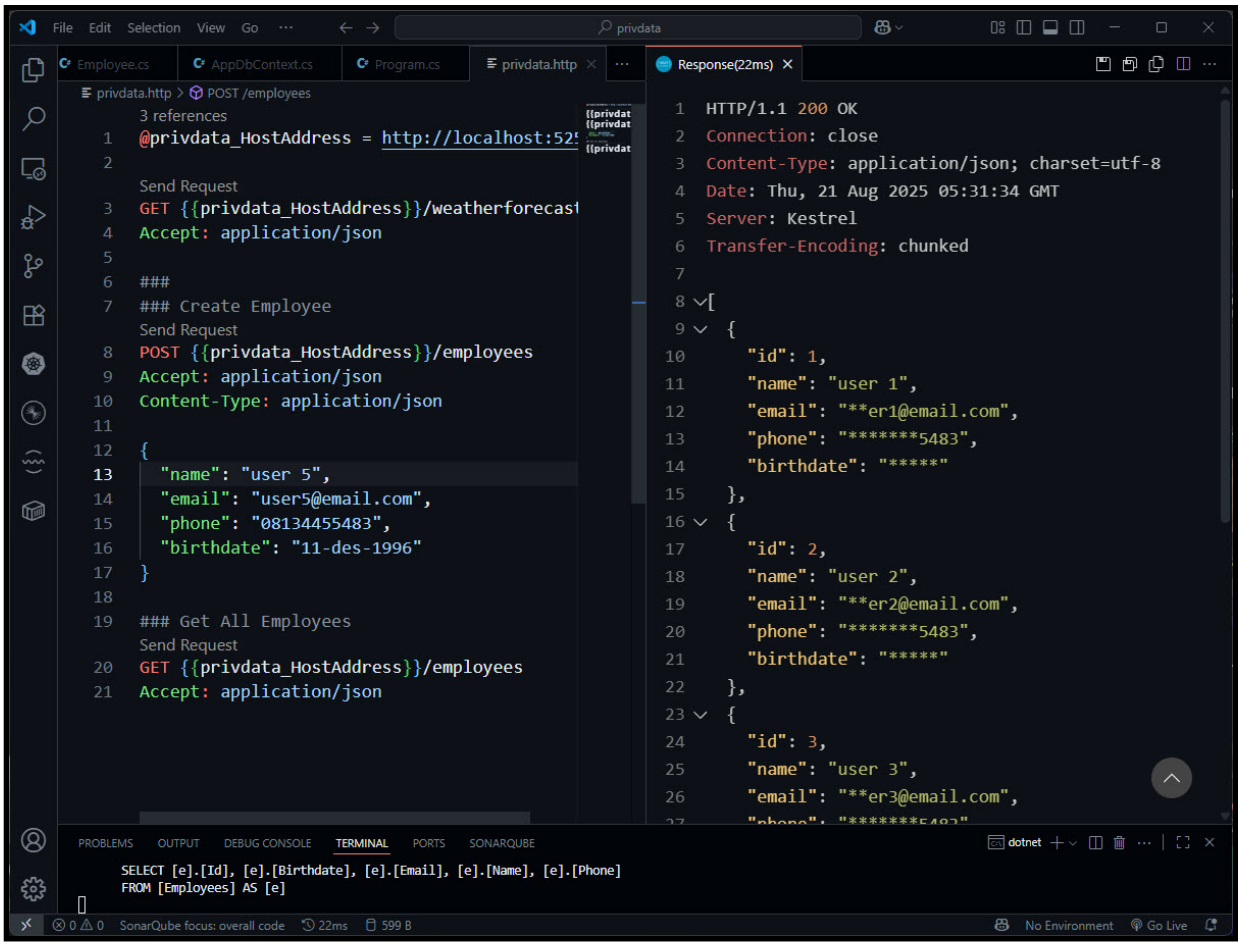


Figure 4.13 A list of employee data with masked data.

4.4.4 Conclusion

This lab demonstrates how to encrypt sensitive employee data before storing it in SQL Server and how to mask this data when sending it to clients in an ASP.NET Core 9.0 Minimal API application. By completing this lab, you gain practical experience in handling sensitive data securely.

4.5 Exercise 15: Rate Limiting and Throttling

In this lab, we learn how to implement rate limiting and throttling in an ASP.NET Core Minimal API application. We'll use the `AspNetCoreRateLimit` package to manage the rate of requests to our API and protect it from abuse and overuse.

4.5.1 Objective

Implement rate limiting and throttling in an ASP.NET Core 9.0 Minimal API to control the rate of requests a user can send to the API. This lab will cover setting up middleware to manage request rates and protect the API from abuse and overuse.

4.5.2 Requirements

- .NET 9.0 SDK installed
- Visual Studio Code or another code editor
- A package for rate limiting, such as `AspNetCoreRateLimit`
- REST Client extension in Visual Studio Code for testing

4.5.3 Lab Steps

1. Set Up the ASP.NET Core 9.0 Minimal API Project

- Open a command prompt or terminal.
- Navigate to your desired working directory or create a new one.

```
mkdir ratelimitapi
cd ratelimitapi
```

- Create a new ASP.NET Core Minimal API project:

```
dotnet new webapi
```

- Open the project in Visual Studio Code:

```
code .
```

2. Install the Rate Limiting Package

- Install `AspNetCoreRateLimit`, a library to implement rate limiting:

```
dotnet add package AspNetCoreRateLimit
```

- Install the Scalar UI package:

```
dotnet add package Scalar.AspNetCore
```

3. Configure Rate Limiting in Program.cs

- In `Program.cs`, configure rate limiting services and middleware:

```
using Scalar.AspNetCore;
using AspNetCoreRateLimit;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddMemoryCache();
builder.Services.Configure<IpRateLimitOptions>(builder.Configuration.GetSection
builder.Services.AddInMemoryRateLimiting();
builder.Services.AddSingleton<IRateLimitConfiguration, RateLimitConfiguration>(

var app = builder.Build();

// Configure the HTTP request pipeline.
app.UseIpRateLimiting();

if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
    app.MapScalarApiReference();
}

// ... other middleware ...
```

- Add rate limiting settings in `appsettings.json`:

```
"IpRateLimiting": {
  "EnableEndpointRateLimiting": true,
  "StackBlockedRequests": false,
  "GeneralRules": [
    {
      "Endpoint": "*",
      "Period": "1m",
      "Limit": 10
    }
  ]
}
```

4. Add Test API Endpoints

- We use existing endpoints from the webapi project template `/weatherforecast`.

5. Build and Run the Application

- Run the application using `dotnet run`.
- We also run the application with https profile:

```
dotnet run --launch-profile https
```

6. Create the `.http` File for Testing

- Prepare a `ratelimitapi.http` file with requests to test the rate-limited endpoints:

```
@ratelimitapi_HostAddress = http://localhost:5205

GET {{ratelimitapi_HostAddress}}/weatherforecast
Accept: application/json

###
```

7. Testing Rate Limiting

- We can test the rate limiting using Scalar UI.
- Open the Scalar UI at `https://localhost:<port>/scalar`.
- Click `Try it out` and `Execute` button for `/weatherforecast` endpoint.
- Try to click more than 10 times until you get an error message.

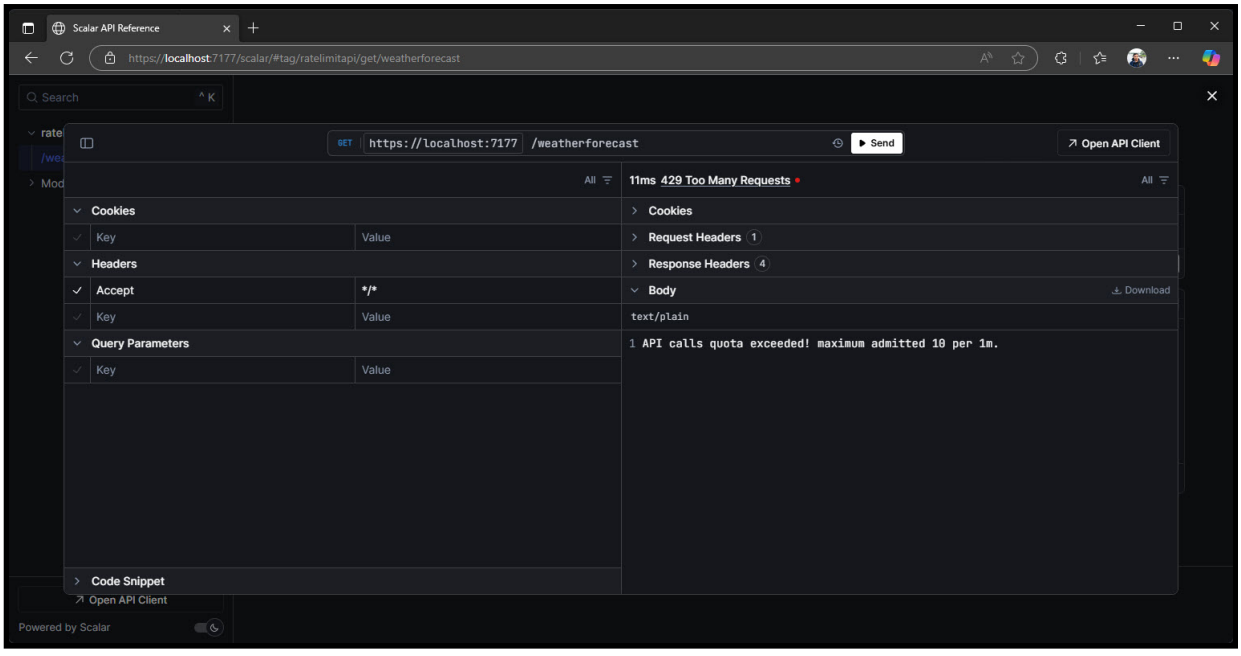


Figure 4.14 Scalar UI from RateLimitApi app.

- You should receive 429 Too Many Requests responses.
- Use the REST Client extension to send multiple requests to your API and observe the rate limiting in action. After exceeding the limit, you should receive 429 Too Many Requests responses.

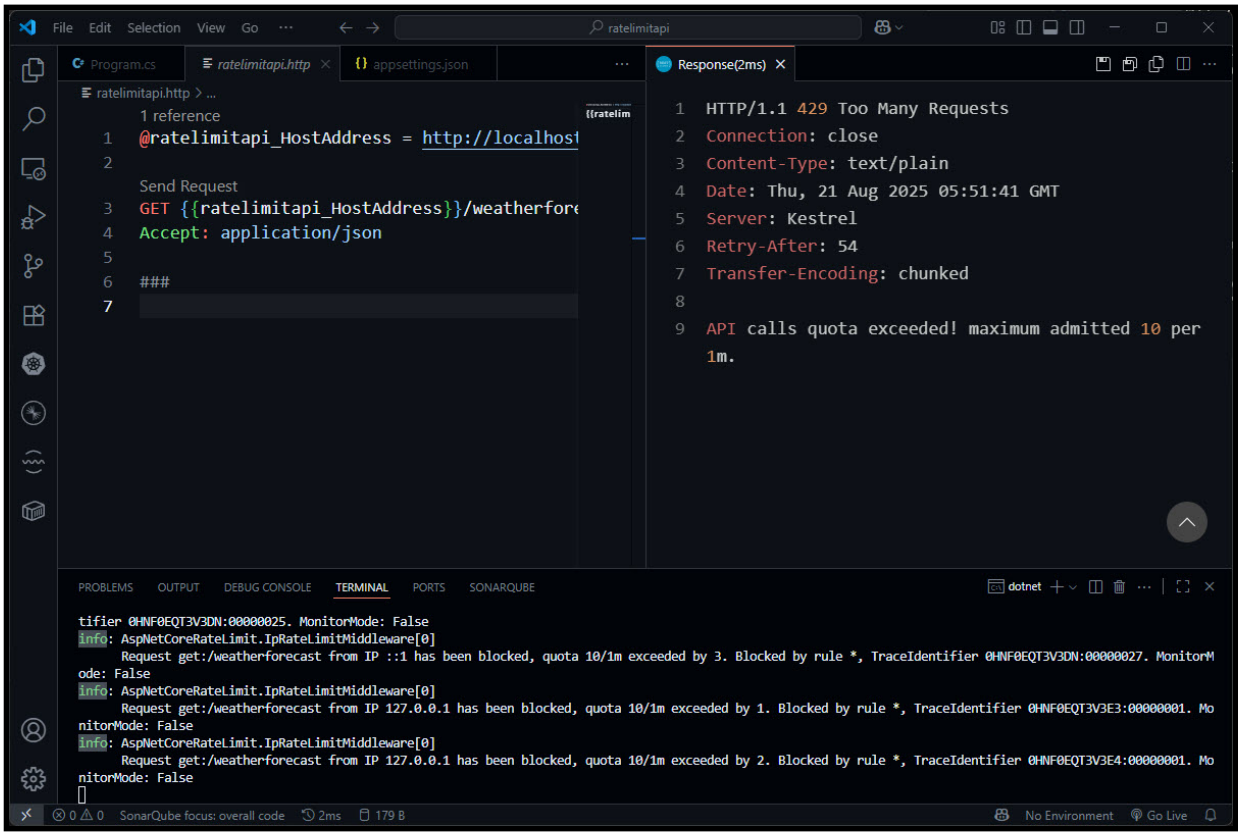


Figure 4.15 REST Client from RateLimitApi app.

4.5.4 Conclusion

This lab provides practical experience in implementing rate limiting in an ASP.NET Core Minimal API. Through the setup of `AspNetCoreRateLimit`, you'll understand how to manage and restrict the rate of requests to your API, which is crucial for maintaining service stability and preventing abuse.

4.6 Exercise 16: Configuring CORS in ASP.NET Core 9.0 Minimal API

In this lab, we learn how to configure and use Cross-Origin Resource Sharing (CORS) in an ASP.NET Core Minimal API application. We'll configure CORS to allow requests from specific origins and test its functionality.

4.6.1 Objective

Learn how to configure and use Cross-Origin Resource Sharing (CORS) in an ASP.NET Core 9.0 Minimal API application. This lab covers setting up CORS to allow requests from specific origins and testing its functionality.

4.6.2 Requirements

- .NET 9.0 SDK installed
- Visual Studio Code or another code editor
- Basic understanding of CORS (Cross-Origin Resource Sharing)
- REST Client extension in Visual Studio Code or a front-end application for testing

4.6.3 Lab Steps

1. Set Up the ASP.NET Core 9.0 Minimal API Project

- Open a command prompt or terminal.
- Navigate to your desired working directory or create a new one.

```
mkdir corsapi
cd corsapi
```

- Create a new ASP.NET Core Minimal API project:

```
dotnet new webapi
```

- Open the project in Visual Studio Code:

```
code .
```

2. Configure CORS in Program.cs

- In `Program.cs`, configure CORS to allow specific origins, methods, and headers:

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

// Add CORS services
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowSpecificOrigin", builder =>
        builder.WithOrigins(["http://example.com", "http://test.com"])
                .AllowAnyMethod()
                .AllowAnyHeader());
});

var app = builder.Build();

// Use CORS with the specified policy
app.UseCors("AllowSpecificOrigin");

// ... other configurations like app.MapGet ...
```

3. Add Test API Endpoints

- Implement some test API endpoints to demonstrate CORS functionality
- We use existing endpoints from the webapi project template `/weatherforecast`.

4. Build and Run the Application

- Run your application using `dotnet run`.
- You can also run the application with https profile:

```
dotnet run --launch-profile https
```

5. Create Client App

- We use a simple HTML page to test CORS.
- Create `index.html` file in the root folder of the project.
- Write these codes

```
<!DOCTYPE html>
<html>
```

```
<head>
    <title>CORS Test</title>
</head>
<body>
    <h1>CORS Request Test</h1>
    <button id="test-cors">Test CORS</button>
    <script>
        document.getElementById('test-cors').addEventListener('click', () => {
            fetch(' https://localhost:7140/weatherforecast')
                .then(response => response.text())
                .then(data => console.log(data))
                .catch(error => console.error('CORS Error:', error));
        });
    </script>
</body>
</html>
```

- Change the port number to match your application's address.
- Project folder should be put on web server.
- We also run this project using Python or PHP built-in web server.
- For Python, run this command inside project folder

```
python -m http.server 8000
```

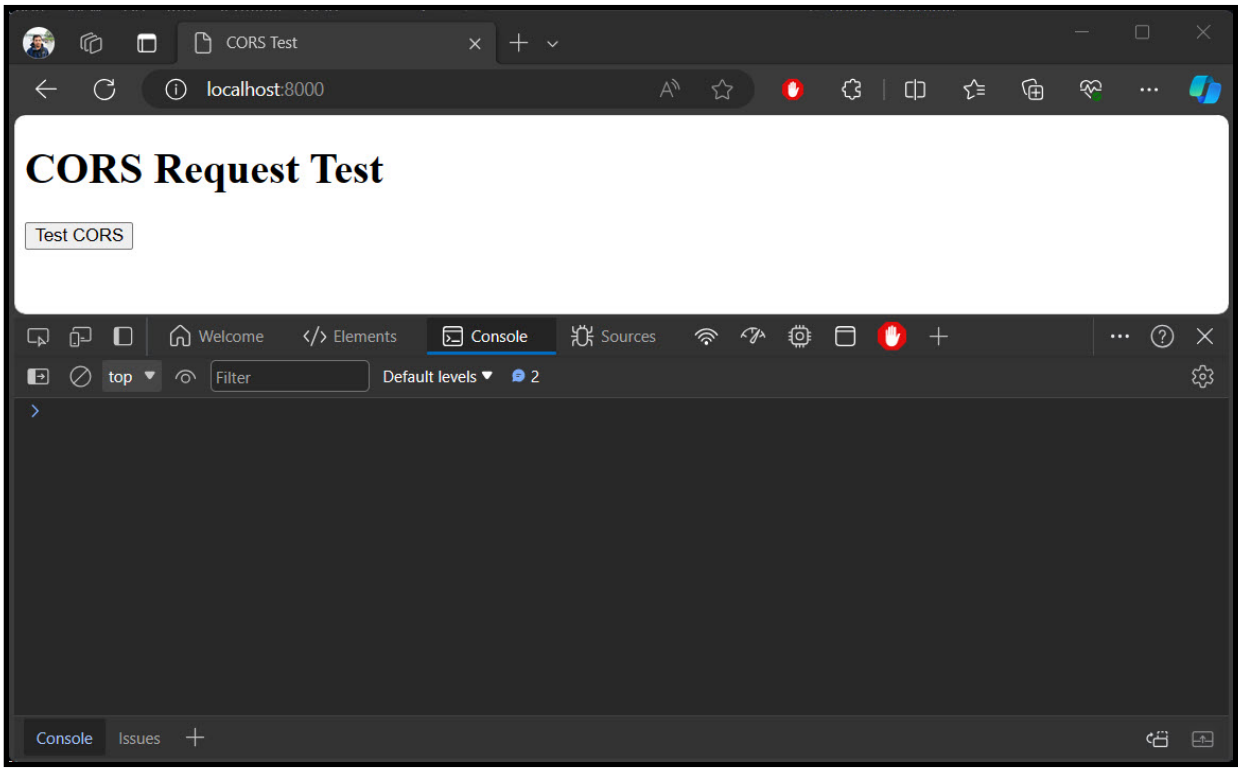


Figure 4.16 CORS Test page.

- Open Developer Tools so you can see the console output.

6. Test CORS Configuration

- After running the application, open the CORS test page in your browser.
- Click the `Test CORS` button to send a request to your API.

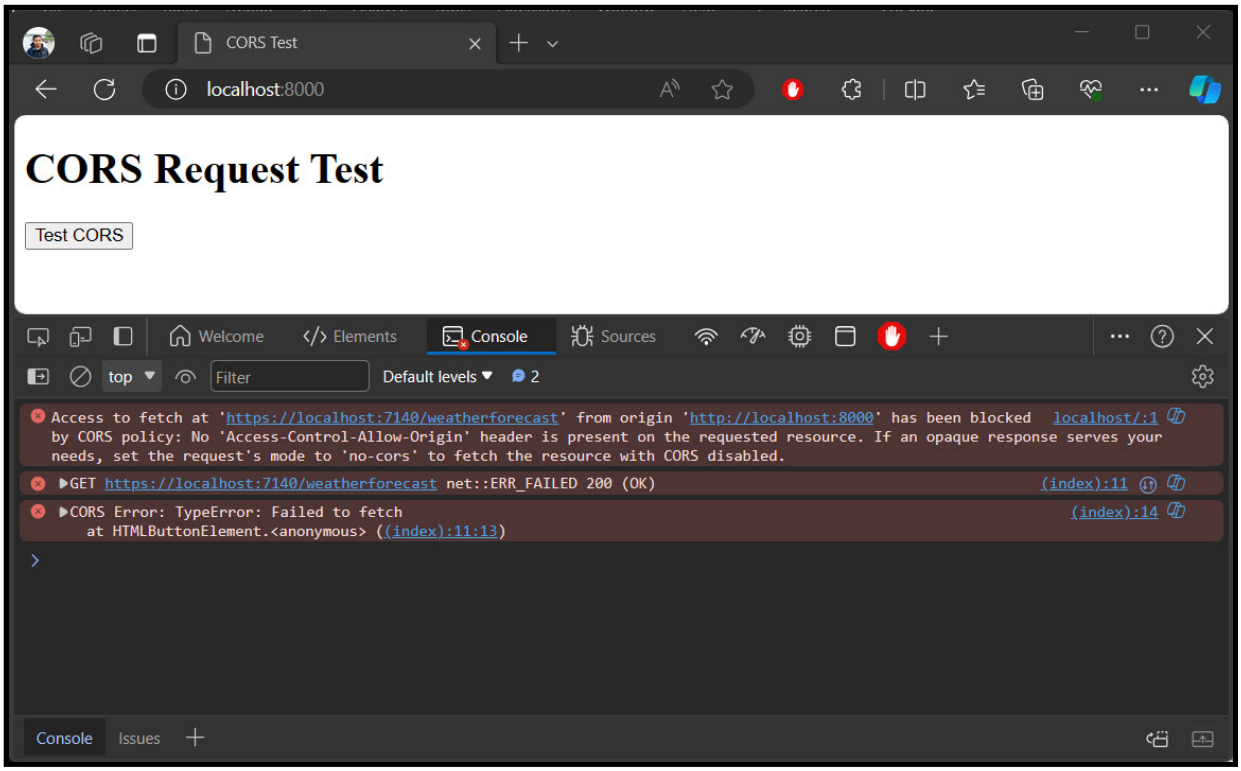


Figure 4.17 Get CORS error.

- You should see a CORS error in the console.
- Now, we configure CORS to allow requests from the test page.
- In `Program.cs`, change the CORS policy to allow all origins, methods, and headers
- We add `"http://localhost:8080"`into `WithOrigins` method.

```
// Add CORS services
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowSpecificOrigin", builder =>
        builder.WithOrigins(["http://example.com", "http://test.com", "http://l
               .AllowAnyMethod()
```

```
                .AllowAnyHeader());
});
```

- Run the application again and test the CORS request.

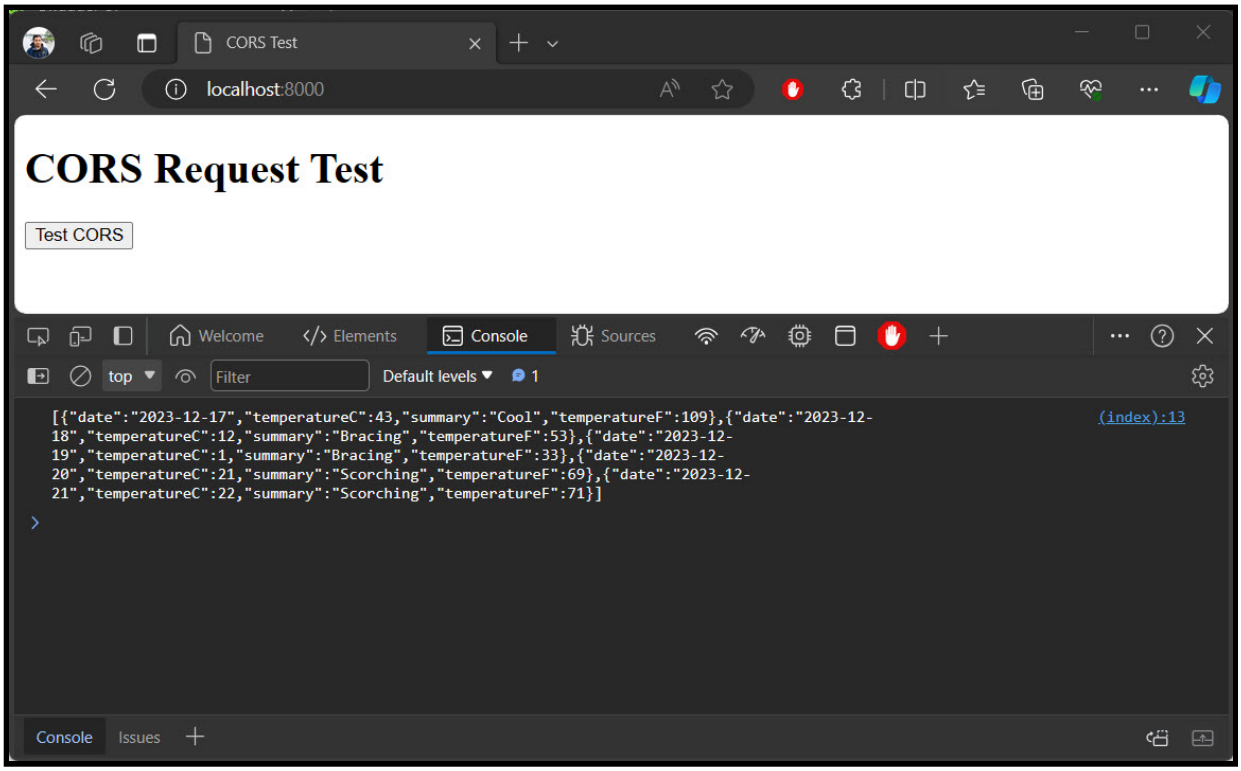


Figure 4.18 CORS request is successful.

- You should see the response from the API in the console.

4.6.4 Conclusion

This lab guides you through setting up and testing CORS in an ASP.NET Core 9.0 Minimal API application. By completing this lab, you'll understand how to configure CORS policies to control how resources in your API can be accessed from a different domain.

5 Monitoring and Deployment

5.1 Introduction

5.1.1 Monitoring in ASP.NET Core 9.0 Minimal API

What is Monitoring?

Monitoring in the context of ASP.NET Core 9.0 Minimal API involves observing and tracking the application's performance, health, and activities. It includes logging, metrics collection, and health checks.

Key Aspects of Monitoring:

1. **Logging:** ASP.NET Core provides built-in support for logging. You can log information about application events, errors, and other significant actions. It supports various logging providers like console, debug, event source, and third-party providers like Serilog or NLog.

2. **Health Checks:** ASP.NET Core offers health check APIs that can be used to check the health of the application and its dependencies, like databases or external services.

3. **Performance Metrics:** Using tools like Application Insights or Prometheus, you can gather performance metrics (response times, request rates, failure rates, etc.) to understand how well the application is performing.

4. **Distributed Tracing:** For microservices architecture, distributed tracing tools like OpenTelemetry or Jaeger can be used to trace requests across different services.

5.1.2 Deployment of ASP.NET Core 9.0 Minimal API

What is Deployment? Deployment is the process of installing, configuring, and enabling a specific version of an application on a server or cloud environment.

Deployment Strategies:

1. **IIS Hosting:** Host your ASP.NET Core app on a Windows Server using Internet Information Services (IIS). Ensure to install the .NET Core Hosting Bundle and configure IIS for ASP.NET Core.

2. **Docker Containers:** Containerize your app with Docker, creating a portable and consistent environment. This allows easy deployment to container orchestration platforms like Kubernetes.

3. **Cloud Platforms:** Deploy your application to cloud services like Azure App Service, AWS Elastic Beanstalk, or Google Cloud Run. These platforms offer easy scaling, management, and additional services like databases, caching, etc.

4. **Linux Hosting:** Deploy the application on a Linux server using reverse proxies like Nginx or Apache. Ensure the server has the .NET runtime installed.

5. **CI/CD Pipeline:** Implement continuous integration and continuous deployment using tools like GitHub Actions, Jenkins, or Azure DevOps. Automate the testing and deployment process to ensure reliable and frequent deployments.

Best Practices:

- **Environment-Specific Configuration:** Use appsettings.json, environment variables, or secret management tools to manage different configurations for development, staging, and production environments.

- **Database Migrations:** Automate database updates using Entity Framework Core migrations to ensure the database schema is up-to-date with the application.

- **Security:** Implement security practices like HTTPS enforcement, CORS policies, and securing sensitive data.

- **Testing:** Prior to deployment, conduct thorough testing including unit testing, integration testing, and load testing.

Monitoring and deployment are crucial aspects of the development lifecycle of ASP.NET Core 9.0 Minimal API applications. Effective monitoring ensures the application runs smoothly and efficiently, while a well-planned deployment strategy enables reliable and scalable delivery of the application to end-users.

5.2 Exercise 17: Health Check and Monitoring

In this lab, you will learn how to implement health checks and monitoring in an ASP.NET Core 9.0 Minimal API application. You will set up health checks for various components of your application and monitor their status.

5.2.1 Objective

Implement health checks and monitoring in an ASP.NET Core 9.0 Minimal API application. This lab will guide you through setting up health checks for various components of your application and monitoring their status.

5.2.2 Requirements

- .NET 9.0 SDK installed
- Visual Studio Code or another code editor
- Basic understanding of ASP.NET Core and its middleware
- REST Client extension in Visual Studio Code for testing

5.2.3 Lab Steps

1. Set Up the ASP.NET Core 9.0 Minimal API Project

- Open a command prompt or terminal.

- Navigate to your desired working directory or create a new one.

```
mkdir healthcheckapi
cd healthcheckapi
```

- Create a new ASP.NET Core Minimal API project:

```
dotnet new webapi
```

- Open the project in Visual Studio Code:

```
code .
```

2. Add Health Checks

- In `Program.cs`, add the health checks middleware:

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

// Add health checks
builder.Services.AddHealthChecks()
    .AddCheck("MyApp", () =>
        HealthCheckResult.Healthy("The check of the sample is OK!"),
        tags: ["myapp"]);

// ... other configurations ...

var app = builder.Build();

// Map health check endpoints
app.MapHealthChecks("/health");

// ... other app configurations ...
```

3. Build and Run the Application

- Compile and run your application.
- Use `dotnet run` to start the application.
- You can also run the application with https profile:

```
dotnet run --launch-profile https
```

4. Create the `.http` File for Testing

- Prepare an `HealthCheckApi.http` file with a request to the health check endpoint:

```
@healthcheckapi_HostAddress = http://localhost:5145

GET {{healthcheckapi_HostAddress}}/health
Accept: application/json

###
```

5. Testing the Health Checks

- Use the REST Client extension to test the health check endpoint.
- Click the **Send Request** button to send the request to the health check endpoint. You should receive a response indicating the health status of the application components.

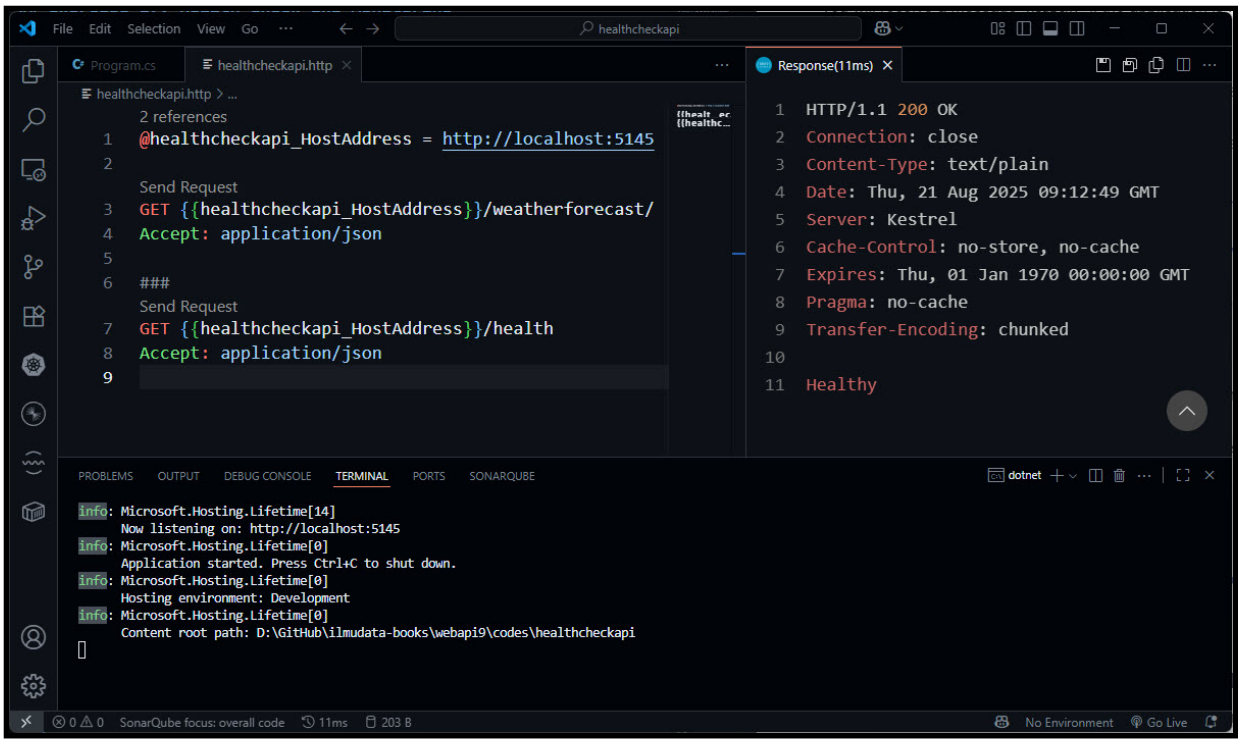


Figure 5.1 Testing the health check endpoint.

- You will receive a message `Healthy` if the application is running correctly.

6. Configure Additional Health Checks (Optional)

- For more advanced scenarios, add health checks for databases, external services, or custom components.

- For example, to add a health check for a SQL Server database, use the following code:

```
builder.Services.AddHealthChecks()
    .AddSqlServer(builder.Configuration["ConnectionStrings:MyDB"],
                  name: "SQL Server",
                  tags: new[] { "db", "sql", "sqlserver" });
```

- Since we call `AddSqlServer()`, we need to add the following package to the project:

```
dotnet add package AspNetCore.HealthChecks.SqlServer
```

- If you have errors related to Invariant Globalization, you may disable `InvariantGlobalization` as `false` on project file, `healthcheckapi.csproj`.

- Configure your database connection string in `appsettings.json` and `appsettings.Development.json`.

```
{
  // ..
  "ConnectionStrings": {
    "MyDB": "server=localhost; database=TrainingDB; uid=tester; pwd=pass123; Tru
  }
  // ..
}
```

- Change the connection string to match your database configuration.

- Now you can test the health check endpoint again. You should receive a response indicating the health status of the application components.

- Try to stop the SQL Server and test the health check endpoint again. You should receive a response indicating the health status of the application components.

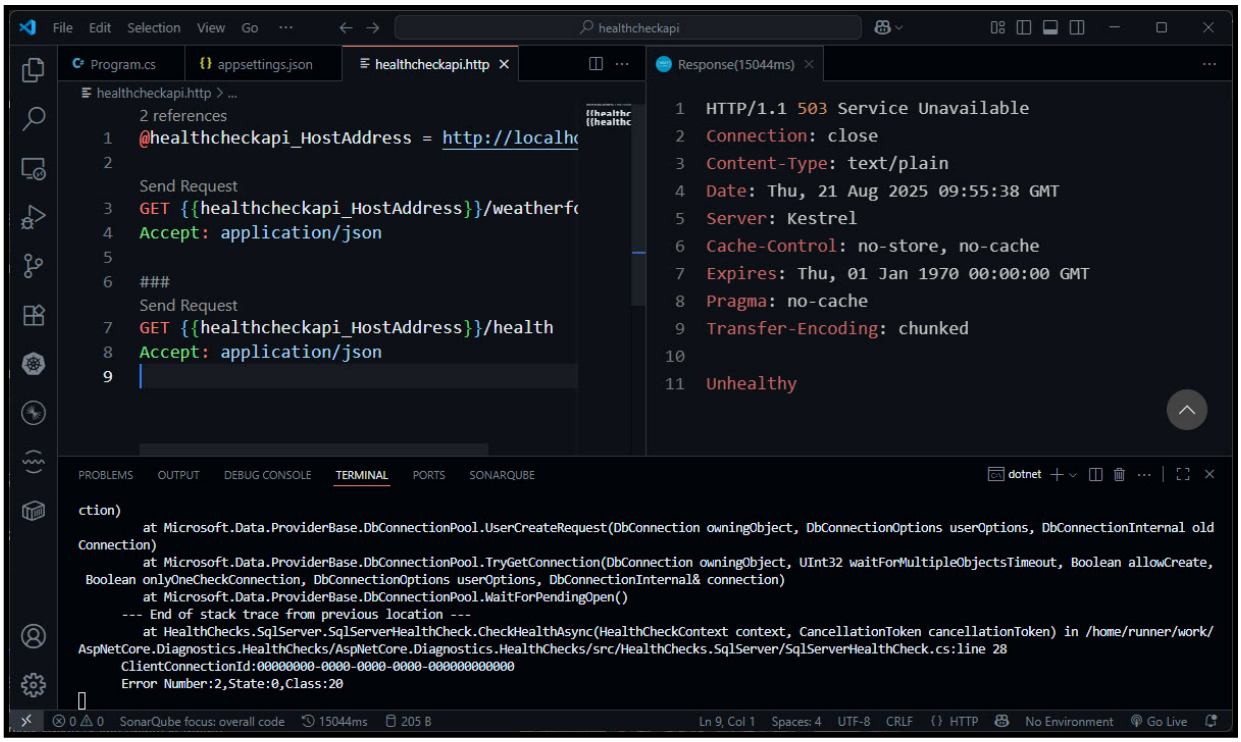


Figure 5.2 Testing the health check endpoint included checking SQL Server.

5.2.4 Implement Custom Health Checks (Advanced)

To create a custom health check in ASP.NET Core that verifies the uptime of an external website (e.g., `http://localhost:9090`, `https://www.google.com`), you will need to implement a custom `IHealthCheck` class. This class will make an HTTP request to the specified URL and determine the health based on the response.

Here's how you can create and register this custom health check:

1. Create a Custom Health Check Class

Create a new class that implements the `IHealthCheck` interface. This class will make an HTTP request to the external website and return `Healthy` if it receives a successful response:

```
using Microsoft.Extensions.Diagnostics.HealthChecks;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

public class ExternalEndpointHealthCheck : IHealthCheck
{
    private readonly string _externalUrl;
```

```
    public ExternalEndpointHealthCheck(string externalUrl)
    {
        _externalUrl = externalUrl;
    }

    public async Task<HealthCheckResult> CheckHealthAsync(HealthCheckContext context,
    {
        using (var httpClient = new HttpClient())
        {
            try
            {
                var response = await httpClient.GetAsync(_externalUrl, cancellationTo
                if (response.IsSuccessStatusCode)
                {
                    return HealthCheckResult.Healthy($"The check for {_externalUrl} i
                }

                return HealthCheckResult.Unhealthy($"The check for {_externalUrl} fai
            }
            catch
            {
                return HealthCheckResult.Unhealthy($"The check for {_externalUrl} fai
            }
        }
    }
}
```

2. Register the Custom Health Check

In your `Program.cs` or `Startup.cs`, register your custom health check with the dependency injection container:

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

builder.Services.AddHealthChecks()
    .AddCheck<ExternalEndpointHealthCheck>("ExternalEndpointHealthCheck",
        null,
        new[] { "external_endpoint" });

var app = builder.Build();

// Map health check endpoint
app.MapHealthChecks("/health");

// ... other configurations ...
```

3. Configure the Custom Health Check

In the registration step, pass the URL of the external website (`https://www.google.com`) to the constructor of your custom health check:

```
builder.Services.AddSingleton<ExternalEndpointHealthCheck>(_ =>
    new ExternalEndpointHealthCheck("https://www.google.com"));
```

4. Test the Health Check

Run your application and access the `/health` endpoint. It should now include the status of the external website in the health check response.

5.2.5 Conclusion

This lab provides a hands-on approach to implementing health checks in an ASP.NET Core 9.0 Minimal API application. By setting up health checks, you can monitor the status of various components of your application and ensure its overall health.

The custom health check provides a simple way to monitor the availability of an external service or website that your application depends on. The health check makes an HTTP request to the specified URL and reports the service as healthy if it receives a successful HTTP response, and unhealthy otherwise. This can be particularly useful in microservices architectures where your service might depend on other external services.

5.3 Exercise 18: Deploying to Web Server IIS

In this lab, you will learn how to deploy an ASP.NET Core 9.0 Minimal API application to an Internet Information Services (IIS) web server. You will cover the necessary steps to prepare, publish, and configure your ASP.NET Core application for IIS deployment.

5.3.1 Objective

Learn how to deploy an ASP.NET Core 9.0 Minimal API application to an Internet Information Services (IIS) web server. This lab covers the necessary steps to prepare, publish, and configure your ASP.NET Core application for IIS deployment.

5.3.2 Requirements

- A Windows server with IIS installed
- .NET 9.0 SDK and Runtime installed on the server
- ASP.NET Core Hosting Bundle installed on the server
- Visual Studio Code or another code editor
- An ASP.NET Core 9.0 Minimal API project

For demo, I use Windows Server 2022 with IIS 10.0 on Virtual Machine.

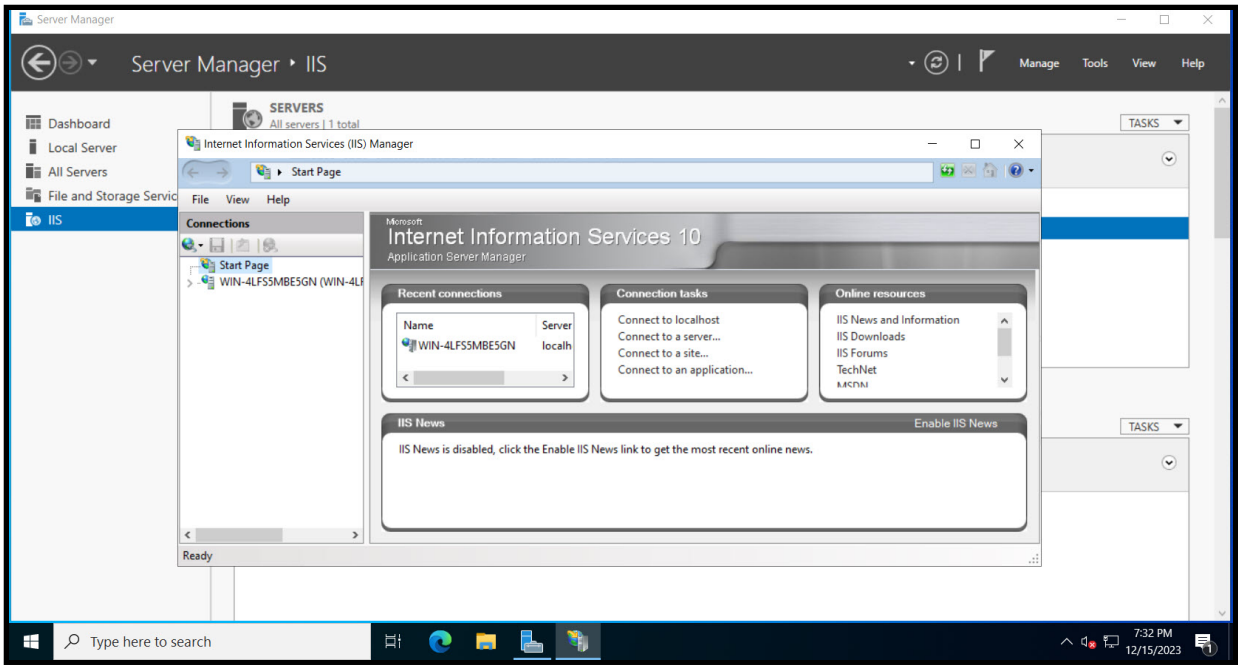


Figure 5.3 Windows Server 2022 with IIS 10.0.

5.3.3 Lab Steps

1. Set Up the ASP.NET Core Project

- Open a command prompt or terminal.
- Navigate to your desired working directory or create a new one.

```
mkdir dotnetapp
cd dotnetapp
```

- Create a new ASP.NET Core Minimal API project:

```
dotnet new webapi
```

- Open the project in Visual Studio Code:

```
code .
```

2. Prepare the Application for IIS

- Ensure that the application is configured to run behind a reverse proxy (IIS acts as a reverse proxy):
    - In `Program.cs`, add the forward headers middleware:

```
using Microsoft.AspNetCore.HttpOverrides;

// ...
var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedH
});

var app = builder.Build();

app.UseForwardedHeaders();

// ... other middleware and configurations ...
```

3. Publish the Application

- Use the .NET CLI to publish your application:

```
dotnet publish -c Release -o ./publish
```

- This command compiles the application and places the output in the `./publish` directory.
- You can see the output of `publush` directory as follows:

4. Install ASP.NET Core Runtime 9.0.0

- Download the ASP.NET Core Runtime 9.0.0 from https://dotnet.microsoft.com/download/dotnet/9.0.
- Copy the downloaded file to the server and install it.

- Install the ASP.NET Core Runtime 9.0.0 on the server.

5. Configure and Deploy the Published Application to IIS

- Copy the contents of the `./publish` directory to your IIS server.
- For instance, you can copy the files to `C:\inetpub\wwwroot\dotnetapp`.
- On the IIS server, click Application Pools and create a new application pool for your application, for instance, `dotnetapp`.
- Ensure that the application pool is configured to use the .NET CLR version `No Managed Code`, as ASP.NET Core runs in its own runtime.

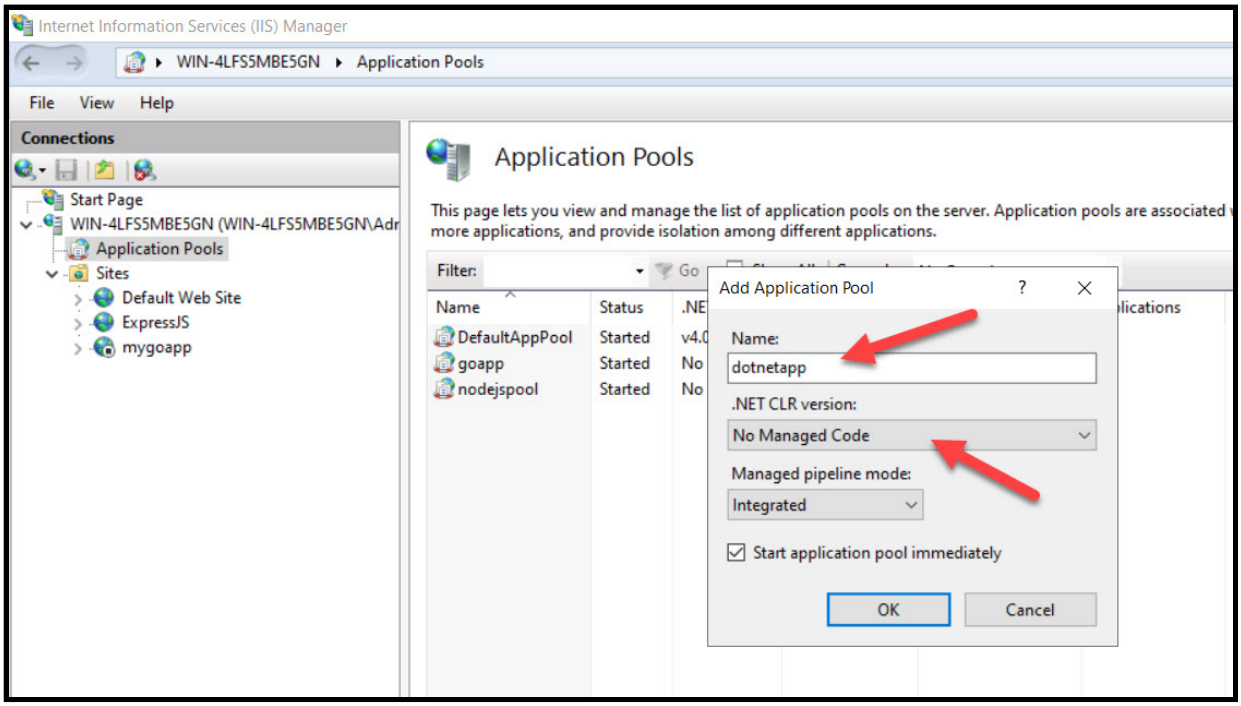


Figure 5.4 Creating a new application pool for the application.

- On the IIS server, create a new website or application in the IIS Manager, pointing the physical path to where you've placed the published files, `C:\inetpub\wwwroot\dotnetapp`.
- Set the application pool to the one you created in the previous step.
- Set port 8081 as the binding port for the website.

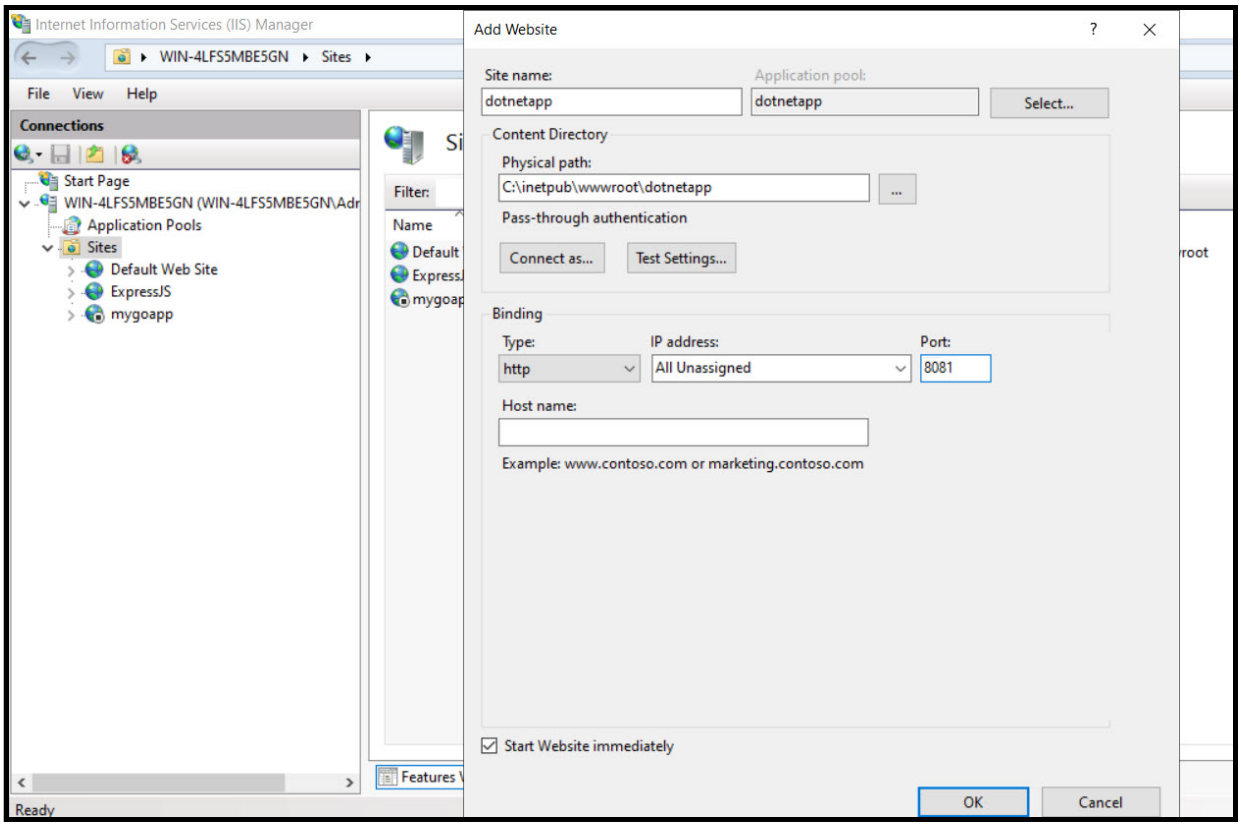


Figure 5.5 Creating a new website for the application.

- Click **OK** to save the website configuration.

6. Test the Deployment

- After setting up the site in IIS, navigate to the application URL in a web browser or use a tool like `curl` to test the endpoints.
- Navigate to `http://localhost:8081/WeatherForecast` to test the application.
- Verify that the application is accessible and functioning as expected.

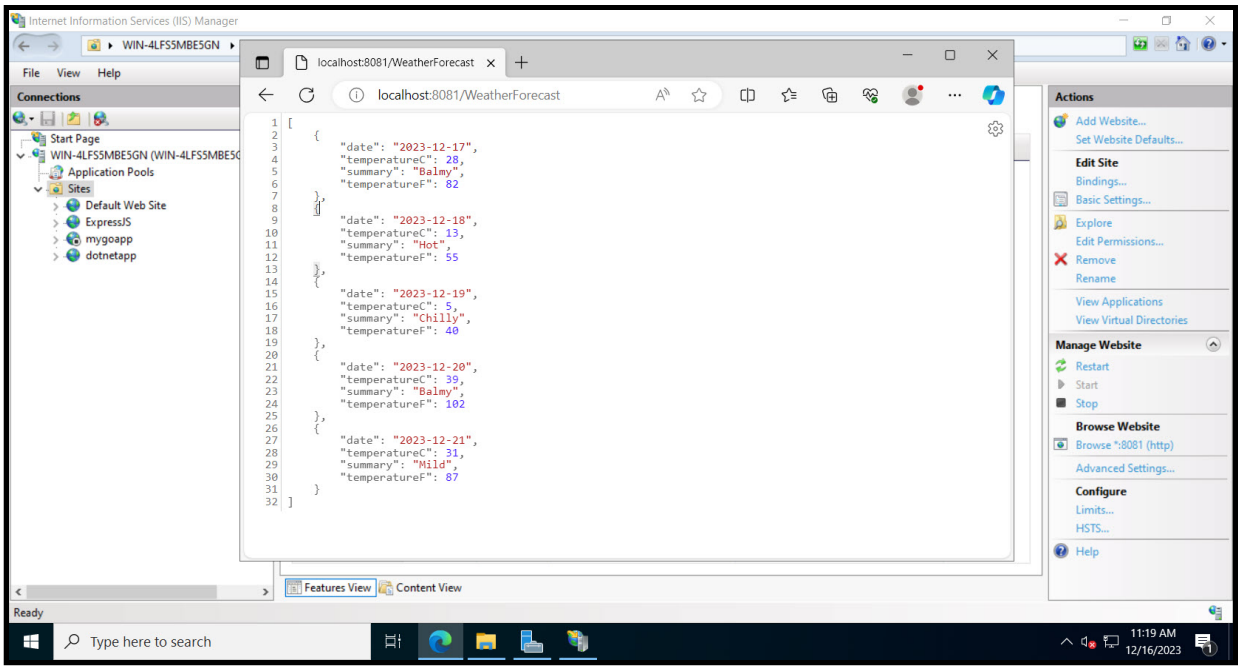


Figure 5.6 Testing the application in a web browser.

7. Troubleshooting (if needed)

- If the application doesn’t work immediately, check the Windows Event Viewer for any application-related errors.
- Ensure that the server’s firewall allows traffic on your application’s port.

5.3.4 Conclusion

This lab provides a practical guide to deploying an ASP.NET Core 9.0 Minimal API application on an IIS web server. By following these steps, you can successfully deploy and run your application in a Windows server environment.

Note: Deployment to a production environment often requires additional considerations, such as setting up secure connections (HTTPS), configuring proper logging, and ensuring that the environment is secure. Always test thoroughly in a staging environment before deploying to production.

5.4 Exercise 19: Deploying to Linux Server with Nginx

In this lab, you will learn how to deploy an ASP.NET Core 9.0 Minimal API application on a Linux server (Ubuntu) using Nginx as a reverse proxy. You will configure .NET Core to run as a daemon on Ubuntu for a production-ready setup.

5.4.1 Objective

Learn how to deploy an ASP.NET Core 9.0 Minimal API application on a Linux server (Ubuntu) using Nginx as a reverse proxy. Configure .NET Core to run as a daemon on Ubuntu for a production-ready setup.

5.4.2 Requirements

- A Linux server running Ubuntu
- Basic knowledge of Linux commands and Nginx
- .NET 9.0 SDK and runtime installed on the server
- Nginx installed on the server
- An ASP.NET Core 9.0 Minimal API project

For demo, I use Ubuntu 22.04 on Virtual Machine.

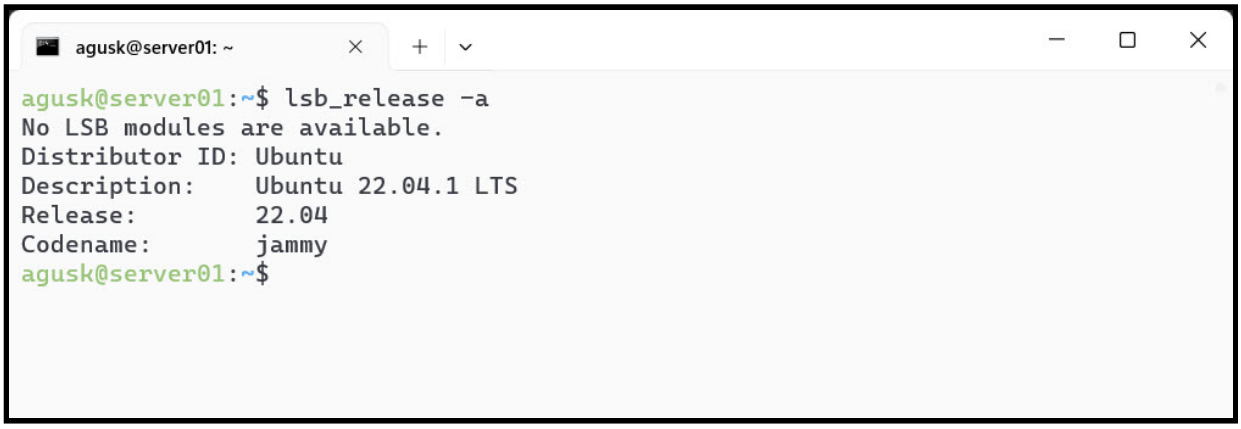


Figure 5.7 Ubuntu 22.04 on Virtual Machine.

5.4.3 Lab Steps

1. Set up .NET on Linux

- You can follow the instructions on the official Microsoft documentation to install .NET on Ubuntu: https://learn.microsoft.com/en-us/dotnet/core/install/linux

- For demo, I use Ubuntu Server 22.04 on Virtual Machine.

- Firstly, we register the Microsoft key and feed, https://learn.microsoft.com/en-us/dotnet/core/install/linux-ubuntu#supported-distributions:

```
# Get Ubuntu version
declare repo_version=$(if command -v lsb_release &> /dev/null; then lsb_release

# Download Microsoft signing key and repository
wget https://packages.microsoft.com/config/ubuntu/$repo_version/packages-micros

# Install Microsoft signing key and repository
sudo dpkg -i packages-microsoft-prod.deb

# Clean up
rm packages-microsoft-prod.deb

# Update packages
sudo apt update
```

- You can install .NET 9.0 SDK using the following commands:

```
sudo apt-get update && \
sudo apt-get install -y dotnet-sdk-9.0
```

- Verify the installation:

```
dotnet --version
```

- You should see your dotnet version

- If you prefer to install .NET 9.0 Runtime, you can use the following commands:

```
sudo apt-get update && \
sudo apt-get install -y aspnetcore-runtime-9.0
```

2. Prepare the ASP.NET Core Application

- Develop your ASP.NET Core 9.0 Minimal API application on your development machine.
- Ensure it runs correctly locally before proceeding with the deployment.
- Navigate to your desired working directory or create a new one.

```
mkdir dotnetapp
cd dotnetapp
```

- Create a new ASP.NET Core Minimal API project:

```
dotnet new webapi
```

- Open the project in Visual Studio Code:

```
code .
```

- Ensure that the application is configured to run behind a reverse proxy (IIS acts as a reverse proxy):
    - In `Program.cs`, add the forward headers middleware:

```
using Microsoft.AspNetCore.HttpOverrides;

// ...
var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedH
});

var app = builder.Build();

app.UseForwardedHeaders();

// ... other middleware and configurations ...
```

- Check to build to ensure that our application is running correctly:

```
dotnet build
```

3. Publish the Application

- Use the .NET CLI to publish your application:

```
dotnet publish -c Release -o ./publish
```

- This command will create a `publish` directory with all the necessary files to run your application.

4. Install Nginx

- Install Nginx on your Linux server:

```
sudo apt install nginx
```

- Start Nginx and enable it to start on boot:

```
sudo systemctl start nginx
sudo systemctl enable nginx
```

- Verify that Nginx is running:

```
sudo systemctl status nginx
```

5. Transfer the Published Application to the Linux Server dt - We will deploy our application to `/var/www/dotnetapp` directory. `sh sudo mkdir /var/www/dotnetapp` - Copy the contents of the `./publish` directory to your Linux server. - For instance, you can copy the files to `/var/www/dotnetapp`. - You can use `cp` command to copy the files: `sh sudo cp -r ./publish/* /var/www/dotnetapp` - You can see the output of `/var/www/dotnetapp` directory as follows: `sh ls -l /var/www/dotnetapp` - Ensure that the application files are owned by the current user: `sh sudo chown -R $USER:$USER /var/www/dotnetapp` - Ensure that the application files have the correct permissions: `sh sudo chmod -R 755 /var/www/dotnetapp`
- You can see the output of `/var/www/dotnetapp` directory as follows: `sh ls -la /var/www/dotnetapp`

- If your application is located on different machine, use `scp` or a similar tool to transfer the contents of the `publish` directory to your Linux server.

6. Configure .NET Core Application as a Daemon

- Create a systemd service file for your application:

```
sudo nano /etc/systemd/system/dotnetapp.service
```

- Add the following content to the service file:

```
[Unit]
Description=Example .NET Web API App running on Ubuntu

[Service]
WorkingDirectory=/var/www/dotnetapp
ExecStart=/usr/bin/dotnet /var/www/dotnetapp/dotnetapp.dll
Restart=always
RestartSec=10
KillSignal=SIGINT
SyslogIdentifier=dotnet-example
User=agusk
Environment=ASPNETCORE_ENVIRONMENT=Production

[Install]
WantedBy=multi-user.target
```

- Replace `/var/www/dotnetapp` with the path to your application and `dotnetapp.dll` with the name of your application DLL.

7. Start and Enable the .NET Core Service

- Start the service and enable it to start on boot:

```
sudo systemctl start dotnetapp.service
sudo systemctl enable dotnetapp.service
```

- Verify that the service is running:

```
sudo systemctl status dotnetapp.service
```

8. Install and Configure Nginx as a Reverse Proxy

- If Nginx is not installed, install it:

```
sudo apt install nginx
```

- Configure Nginx to reverse proxy to your application:

```
sudo nano /etc/nginx/sites-available/default
```

- Add the following configuration inside the `server` block:

```
location / {
    proxy_pass         http://localhost:5000;
```

```
    proxy_http_version 1.1;
    proxy_set_header   Upgrade $http_upgrade;
    proxy_set_header   Connection keep-alive;
    proxy_set_header   Host $host;
    proxy_cache_bypass $http_upgrade;
    proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header   X-Forwarded-Proto $scheme;
}
```

- Comment if you find `location /` block.

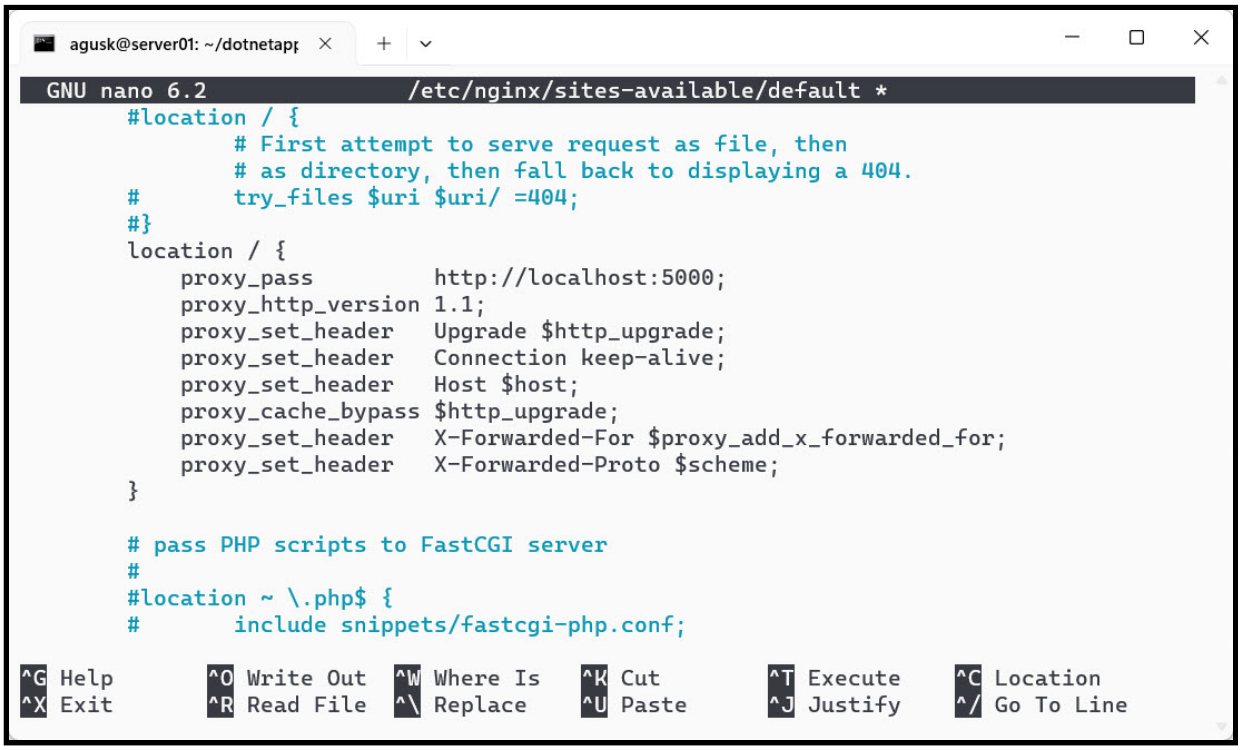


Figure 5.8 Configuring Nginx as a reverse proxy.

- Replace `http://localhost:5000` with the URL and port your .NET application is running on.
- `http://localhost:5000` is the default URL and port for ASP.NET Core applications.

8. Restart Nginx

- Restart Nginx to apply the changes:

```
sudo systemctl restart nginx
```

9. Test the Deployment

- Open a web browser and navigate to your server's IP address or domain name. You should see your ASP.NET Core application running.
- Navigate to `http://<server-ip>/WeatherForecast` to test the application.

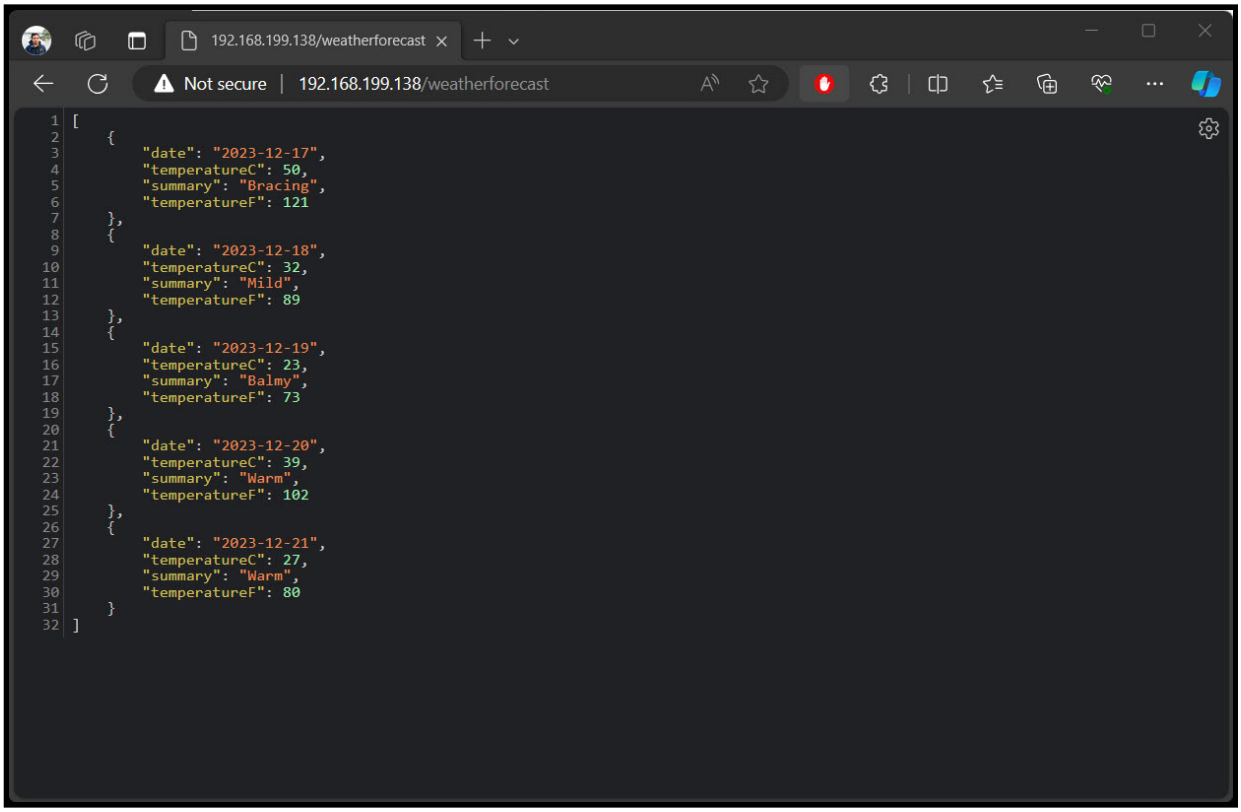


Figure 5.9 Testing the application in a web browser.

5.4.4 Conclusion

You have successfully deployed an ASP.NET Core 9.0 Minimal API application on a Linux server running Ubuntu, using Nginx as a reverse proxy. The application runs as a systemd service, ensuring it starts automatically and remains running.

Security and Maintenance Tips:

- Regularly update your Ubuntu server and installed packages.
- Secure your application and server using firewalls, SSL/TLS certificates, and by following best security practices.
- Monitor your application and server performance and logs for any issues or potential improvements.

5.5 Exercise 20: Deploying to Container Platforms

In this lab, you will learn how to containerize an ASP.NET Core 9.0 Minimal API application and deploy it to a container platform. You will cover creating a Docker container for your application and deploying it to a container orchestration platform such as Kubernetes or Docker Swarm.

5.5.1 Objective

Learn how to containerize an ASP.NET Core 9.0 Minimal API application and deploy it to a container platform. This lab will cover creating a Docker container for your application and deploying it to a container orchestration platform such as Kubernetes or Docker Swarm.

5.5.2 Requirements

- .NET 9.0 SDK installed
- Docker installed on your local machine
- Basic understanding of containerization concepts
- Access to a container orchestration platform (like Kubernetes, Docker Swarm, or a cloud-based container service)

5.5.3 Lab Steps

1. Set Up the ASP.NET Core Project

- Develop your ASP.NET Core 9.0 Minimal API application locally.
- Test the application to ensure it’s working correctly before containerizing it.
- Navigate to your desired working directory or create a new one.

```
mkdir dotnetdocker
cd dotnetdocker
```

- Create a new ASP.NET Core Minimal API project:

```
dotnet new webapi
```

- Open the project in Visual Studio Code:

```
code .
```

2. Create a Dockerfile

- In the root of your ASP.NET Core project, create a file named `Dockerfile`.
- Add the following contents to the `Dockerfile`:

```
FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

# Use SDK image to build the application
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
WORKDIR /src
COPY dotnetdocker.csproj .
RUN dotnet restore "dotnetdocker.csproj"
COPY . .
RUN dotnet build "dotnetdocker.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "dotnetdocker.csproj" -c Release -o /app/publish

# Build runtime image
FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "dotnetdocker.dll"]
```

    - Replace `dotnetdocker` with the name of your project.

3. Build and Test the Docker Image

- Build the Docker image from the root of your project:

```
docker build -t dotnetdocker .
```

- Once the image is built, run it locally to test:

```
docker run --rm -p 8080:8080 dotnetdocker
```

- Navigate to `http://localhost:8080/weatherforecast` in your browser to ensure the containerized application is running correctly.

4. Push the Image to a Container Registry

- For instance, if we want to publish to Docker Hub, we need to tag the image as follows:

```
docker tag dotnetdocker yourdockerhubusername/dotnetdocker:tag
```

- Change `yourdockerhubusername` with your Docker Hub username and `tag` with the version of your application.

- Sign in to Docker Hub:

```
docker login
```

- Push the image to Docker Hub:

```
docker push yourdockerhubusername/dotnetdocker:tag
```

5. Deploy to a Container Orchestration Platform

- Prepare your deployment and service YAML files for Kubernetes or a similar configuration for other platforms.

- Write `deployment.yaml` and `service.yaml` files as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dotnetdocker
  labels:
    app: dotnetdocker
spec:
  replicas: 1
  selector:
    matchLabels:
      app: dotnetdocker
  template:
    metadata:
      labels:
        app: dotnetdocker
    spec:
      containers:
      - name: dotnetdocker
        image: yourdockerhubusername/dotnetdocker:tag
        ports:
        - containerPort: 8080
```

- Change `yourdockerhubusername` with your Docker Hub username and `tag` with the version of your application.

- Write `service.yaml` file as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: dotnetdocker
  labels:
    app: dotnetdocker
spec:
  type: LoadBalancer
  ports:
  - port: 80 # external port
    targetPort: 8080 # internal port
  selector:
    app: dotnetdocker
```

- Deploy the application to the container platform:

```
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

- Check the deployment status to ensure everything is running correctly.

6. Access the Deployed Application

- Once deployed, access your application through the load balancer or node port provided by your container platform.

5.5.4 Conclusion

You have successfully containerized and deployed an ASP.NET Core 9.0 Minimal API application to a container platform. This deployment strategy enhances the scalability, portability, and consistency of your application across different environments.

Best Practices: - Always test your containerized application locally before deploying it to production. - Manage sensitive configuration data using environment variables or configuration management tools provided by your container platform. - Keep your container images updated and scan them for vulnerabilities. - Monitor the performance and health of your deployed application using platform-specific tools or third-party solutions.

This is the end of the book. I hope you enjoyed it. If you have any questions, please contact me.

Appendix A: C# Cheat Sheet

Basic Syntax and Structure:

1. **Hello World:**

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

2. **Variables and Data Types:**

```
int number = 10;
string text = "Hello";
bool isTrue = true;
double decimalNumber = 5.99;
char letter = 'A';
```

3. **Constants:**

```
const double PI = 3.14159;
```

4. **Arrays:**

```
int[] numbers = { 1, 2, 3, 4, 5 };
string[] names = new string[5];
```

5. **Loops:**

    - For Loop:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

- While Loop:

```
int i = 0;
while (i < 5)
{
    Console.WriteLine(i);
    i++;
}
```

- Foreach Loop:

```
string[] names = { "Anna", "Bill", "Cindy" };
foreach (string name in names)
{
    Console.WriteLine(name);
}
```

6. **Conditionals:**

- If-Else:

```
if (number > 0)
{
    Console.WriteLine("Positive");
}
else
{
    Console.WriteLine("Non-Positive");
}
```

- Switch:

```
switch (number)
{
    case 1:
        Console.WriteLine("One");
        break;
    case 2:
        Console.WriteLine("Two");
        break;
    default:
        Console.WriteLine("Other");
        break;
}
```

7. **Methods:**

```
void PrintName(string name)
{
    Console.WriteLine(name);
}
```

8. **Classes and Objects:**

   - Class Definition:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public void Greet()
    {
        Console.WriteLine("Hello " + Name);
    }
}
```

   - Creating an Object:

```
Person person = new Person();
person.Name = "Alice";
person.Greet();
```

ASP.NET Core Specific:

9. **Minimal API Endpoint:**

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();
```

10. **Dependency Injection in Controllers:**

```
public class MyController : ControllerBase
{
    private readonly MyService _service;

    public MyController(MyService service)
    {
        _service = service;
    }
}
```

11. **Entity Framework Core - Basic Query:**

```
using (var context = new MyDbContext())
{
```

```
    var users = context.Users.ToList();
}
```

12. **Middleware:**

```
app.Use(async (context, next) =>
{
    // Pre-processing logic here
    await next.Invoke();
    // Post-processing logic here
});
```

This cheat sheet provides a quick reference to some of the most commonly used features and syntax in C#. It’s a handy tool for developers working with the .NET framework and ASP.NET Core, especially when developing Minimal APIs. Remember that C# and .NET are vast, and this cheat sheet only scratches the surface of what’s possible.

Appendix B: Resources

SQL Server 2025 High Availability & Disaster Recovery: Always On Solutions Course

Dive into the world of SQL Server 2025 with our comprehensive Udemy course, **“SQL Server 2025: Build Always On HA & DR Solutions.”** This course is designed for database administrators and IT professionals who want to master high availability (HA) and disaster recovery (DR) solutions using the latest features of SQL Server 2025.

What You’ll Learn

In this course, you will learn to:

- Understand HA and DR concepts in SQL Server 2025
- Build and configure Windows Server Failover Clustering (WSFC)
- Deploy Always On Availability Groups from scratch
- Set up and manage the AG Listener for client connections
- Configure read-only routing for reporting and BI workloads
- Offload backups using Preferred Backup Replica
- Perform failover testing: automatic, manual, and forced
- Monitor and troubleshoot AG health
- Integrate real-world ASP.NET Core apps with AG Listener
- Apply best practices for performance and uptime

100% Hands-On with Real Labs

This course is not just theory. You’ll build your own lab environment using virtual machines and simulate real-world HA/DR use cases.

We guide you through every step — from cluster setup to full availability group testing. Whether you’re creating an AG with two replicas or

deploying to a multi-subnet environment, this course shows you how it works in practice.

No scripts without context. No fluff. Just practical demos you can repeat and apply at work.

Enroll today: *SQL Server 2025: Build Always On HA & DR Solutions*
https://www.udemy.com/course/sqlserverag/?referralCode=2E28F5CFD4DFBAD4EC15

Enhance Your Learning with Our Udemy Course

For those who've journeyed with us through this book, we have something special to further your understanding — a comprehensive Udemy course titled **"Red Hat NGINX Web Server: Publishing and Deploying Web Apps."**

Why Choose This Course?

1. **Specialized Knowledge**: Dive deep into the world of Red Hat and NGINX. Understand how to use NGINX on the Red Hat platform, a powerful combination for web server deployments.
2. **Hands-On Approach**: Our course isn't just about theory; we believe in the 'learn by doing' philosophy. With guided tutorials and real-world examples, grasp how to publish and deploy various web applications effectively.
3. **Expert Instructors**: Benefit from the insights and expertise of professionals who are not just educators but industry practitioners with years of experience.
4. **Flexible Learning**: Learn at your own pace. With lifetime access, you can revisit topics anytime and solidify your understanding.

Who Is This Course For? - Web developers looking to understand the deployment process on Red Hat using NGINX. - System administrators aiming to expand their knowledge in server configuration and optimization. - IT professionals transitioning to roles that require knowledge of web server setup and deployment on Red Hat.

Enroll today: Red Hat NGINX Web Server: Publishing and Deploying Web Apps https://www.udemy.com/course/rhel-nginx/?referralCode=C9CFA39AE9E332ADA9FB

While I can't directly access or view content on external sites, including Udemy, I can draft a promotional piece for your course "Mastering Docker: Publishing and Deploying Web Applications" based on the title and URL you've provided. Here's a promotional content for your course:

Dive Deeper into Containerization with Our Udemy Course

Having explored the vast realm of NGINX, it's time to take a leap into another crucial technology in the modern web infrastructure world: Docker. We're excited to introduce our Udemy course: **"Mastering Docker: Publishing and Deploying Web Applications."**

Course Highlights:

1. **Comprehensive Docker Mastery**: Navigate through the intricacies of Docker, from understanding its architecture to deploying real-world web applications.
2. **Practical & Hands-On**: Delve into practical scenarios, Dockerfile creation, container orchestration, and more. The course isn't just about theory; it's about empowering you with real-world skills.
3. **Expert Guidance**: Learn from seasoned professionals who bring their wealth of industry knowledge to the table. Each module is tailored to ensure you grasp the essence of Docker in web deployments.
4. **On-The-Go Learning**: Our course is structured for both beginners and seasoned developers. With lifetime access, dive into lessons at your convenience, and revisit modules anytime.

Who Should Enroll?

- Web developers keen on leveraging containerization for their applications.

- DevOps professionals looking to streamline their CI/CD processes using Docker.
- IT enthusiasts aiming to gain a firm grasp on the future of web application deployment.

Exclusive Features: - Detailed lessons breaking down Docker's complex topics. - Engaging quizzes to test and solidify your understanding. - A certificate of completion to enhance your professional journey. - A vibrant community forum to discuss, share, and learn from peers.

Expand your horizons beyond traditional web server technologies. Dive into Docker and understand why it's the talk of the tech world!

Join the learning journey: Mastering Docker: Publishing and Deploying Web Applications https://www.udemy.com/course/webdocker/?referralCode=E839AA8926D06B16DD61

Build Secure PHP APIs Like a Pro with Laravel 12, OAuth2, and JWT

Unlock the full potential of **Laravel 12** for REST API development! This hands-on course on Udemy teaches you how to build **robust, secure, and modern APIs** using Laravel, MySQL, OAuth2, JWT, Sanctum, and Role-Based Access Control (RBAC). Perfect for real-world applications and 2025 standards.

🚀 Highlight Topics

- What's New in Laravel 12 for API development
- Build RESTful APIs from scratch (Hello World to full CRUD)
- File upload and user data handling via REST API
- Secure authentication with **Sanctum**, **JWT**, and **OAuth2**
- Role-Based Access Control (RBAC) with middleware
- Legacy support: Laravel 8, 7.x, and 6.x projects included
- Real project codebases and testing tutorials

👨‍💻 Who Should Enroll?

- Laravel developers aiming to modernize their API skills
- Backend engineers securing APIs with token-based auth
- Teams migrating legacy Laravel APIs to newer standards
- Students and professionals building real-world Laravel apps
- Anyone preparing for backend development roles in 2025

> ***Future-proof your Laravel skills.*** *This course gives you* ***everything you need*** *to build secure, scalable, and professional REST APIs in Laravel 12. Learn by doing — with real code, live tests, and full project coverage.*

👉 **Join now and start building APIs that meet today's security demands.** *PHP REST API: Laravel 12, MySQL, OAuth2, JWT, Roles-Based* https://www.udemy.com/course/phprestapi/?referralCode=2C5B2F14100B499E9845

Master Real-World Logging & Visualization with the Full ELK Stack

Take control of your logging, search, and monitoring pipeline with this **hands-on Udemy course** covering Elasticsearch, Logstash, Kibana, and Beats. Learn how to set up, ingest, visualize, and scale log data using practical projects — all designed for developers, sysadmins, and DevOps engineers in **real production environments**.

🚀 Highlight Topics

- Cross-platform installation: Windows, Ubuntu, macOS, Docker
- Elasticsearch REST API: CRUD, mapping, queries, aggregation, SQL, geo fields
- Real-world API integration: PHP, ASP.NET Core, Node.js, Python
- Logstash ingestion: files, folders, and RDBMS (MySQL)
- Kibana Lens visualizations: charts, maps, dashboards, Canvas
- Beats agents: Filebeat, Winlogbeat, Metricbeat, Packetbeat, Heartbeat, Auditbeat
- High Availability (HA) setup for Elasticsearch and Kibana with Nginx

👨‍💻 Who Should Enroll?

- Developers and DevOps engineers building log-driven applications
- System administrators responsible for monitoring and observability
- Backend/API developers seeking integration with Elasticsearch
- Cybersecurity analysts and IT ops engineers using ELK for log auditing
- Teams adopting open-source observability tools for modern infrastructure

> ***Log smarter, visualize better, and scale with confidence.*** *Whether you're just getting started or already managing production systems, this course gives you everything you need to build and operate a* ***powerful ELK Stack pipeline****. With real-world use cases, cross-platform setups, and step-by-step guidance, you'll go beyond the basics and into expert territory.*

👉 **Enroll today** to master the ELK Stack and unlock actionable insights from your data! *Practical Full ELK Stack: Elasticsearch, Kibana and Logstash* https://www.udemy.com/course/elkstack/?referralCode=863C1036F77169C975C5

Appendix C: Source Code

You can download the source code files for this book from GitHub at https://github.com/agusk/ilmudata-book-aspnet9-minimalapi.

About

Agus Kurniawan's journey in the field of technology, spanning from 2001, is a remarkable blend of deep technical expertise and a fervent passion for sharing knowledge. As a seasoned professional, Agus has carved a niche in diverse technological domains, including software development, IoT (Internet of Things), Machine Learning, IT infrastructure, and DevOps. His experiences are not just limited to developing cutting-edge solutions but also extend to shaping the future of upcoming technologists through training and workshops.

Agus's career is marked by significant contributions to both technological innovation and community development. His recognition as a Microsoft Most Valuable Professional (MVP) from 2004 to 2022 underlines his proficiency in Microsoft technologies and his dedication to educating others. Agus has been at the forefront of delivering various training sessions and workshops, sharing his insights and helping others grow in the ever-evolving tech industry.

Agus Kurniawan's book, "Hallo .NET 9.0: Practical ASP.NET Core Minimal API," is a comprehensive guide that captures his extensive experience in the realm of software development, particularly focusing on the latest advancements in .NET technology. This book is a culmination of Agus's in-depth knowledge and practical approach to building applications using ASP.NET Core 9.0, a framework known for its efficiency and minimalism.

Contact the Author

Agus Kurniawan values the feedback, queries, and insights of his readers. Whether you have a technical question related to .NET, a suggestion for future editions of this book, or just want to share your experiences and thoughts, Agus welcomes your correspondence.

Email: aguskur@hotmail.com, agusk2007@gmail.com

LinkedIn: linkedin.com/in/agusk

Twitter: [@agusk2010]