800 XP

Create a .NET Aspire project

32 min • Module7 Units

Intermediate | Developer | Solution Architect | .NET | ASP.NET Core | .NET Aspire | Visual Studio | Visual Studio Code

Learn how to create cloud-native applications from scratch or add orchestration to an existing app by using the .NET Aspire stack in .NET 8.

Learning objectives

By the end of this module, you're going to be able to:

- Create new cloud-native apps by using the .NET Aspire templates in Visual Studio.
- Add the .NET Aspire stack to an existing .NET app for orchestration and simple cloud-native components.
- Use the .NET Aspire dashboard to diagnose connection issues between services.

Start > ⊕ Add

Prerequisites

- Experience building web applications using .NET and C#

This module is part of these learning paths

Build distributed apps with .NET Aspire

> Introduction
> 1 min

> Learn how to create a new .NET Aspire project
> 5 min

> Exercise - Create a new .NET Aspire project
> 10 min

> How to add orchestration to an existing .NET app
> 5 min

> Exercise - Integrate an existing ASP.NET Core web app
> 8 min

Knowledge check

2 min

Summary

1 min

100 XP

Introduction

1 minute

Creating a cloud-native application can be challenging, as it requires in-depth technical knowledge of both coding and infrastructure. Each microservice needs to locate and communicate with others, store data, utilize common services such as caches, and manage messaging queues. Using .NET Aspire simplifies many of these challenges, allowing you to focus more on developing your application's unique functionality.

Imagine you work for an outdoor equipment retailer. You're new to the development team, and want to see how hard it is to add .NET Aspire to the team's current cloud-native app. You believe .NET Aspire can help your team reduce development effort.

In this module, learn how to use the .NET Aspire project templates in Visual Studio to create new cloud-native apps in just a few steps. To ease further development on an existing code base, see how to enlist an existing app in .NET Aspire orchestration.

Learning objectives

By the end of this module, you're going to be able to:

- Create new cloud-native apps by using the .NET Aspire templates in Visual Studio.
- Add the .NET Aspire stack to an existing .NET app for orchestration and simple cloud-native components.

100 XP

Learn how to create a new .NET Aspire project

5 minutes

Cloud-native development can require developers to connect together different micro-services like databases, messaging queues, and caches. .NET Aspire simplifies this process by providing a set of templates you can use to create and manage the connections between these services.

In this unit, learn how to create a new .NET Aspire project and understand the differences between the two starter project templates. Then explore the structure of the solution generated.

.NET Aspire prerequisites

Before you can create a new .NET Aspire project, there are some prerequisites you need to install locally:

- .NET 8
- Visual Studio 2022 Preview
- Docker Desktop or Podman
- .NET Aspire workload in Visual Studio

In the next exercise, you'll go through the steps to install these prerequisites.

Choose the best .NET Aspire template for your project

There are two .NET Aspire starter templates currently available:

- **.NET Aspire Application**: This template is a good starting point for new projects. It only includes the **AspireSample.AppHost** and **AspireSample.ServiceDefaults** projects. This template is useful when you want to start from scratch and add your own components and services.

- **.NET Aspire Starter Application**: This template includes the **AspireSample.AppHost** and

AspireSample.ServiceDefaults projects, but also includes an example Blazor App **AspireSample.Web** and an API that provides data to it **AspireSample.ApiService**. These projects are preconfigured with service discovery and other basic examples of common .NET Aspire functionality.

Both Aspire templates provide a dashboard to monitor the health of the services and the traffic between them. The dashboard helps improve your local development experience — as at a glance it gives you an overview of the state and structure of your app.

There are also three project templates available:

- **.NET Aspire App Host**: A template that only contains an app host (orchestrator) project.
- **.NET Aspire Service Defaults**: A template that only contains the service defaults project.
- **.NET Aspire Test Project**: A template that only contains unit tests for the app host project.

Creating a new .NET Aspire project by using a .NET Aspire template

You can use the Visual Studio launch dialog to create a new .NET Aspire project, or **File** > **New** > **Project**. You can also use .NET CLI commands. To create a solution with the *.NET Aspire Application* template, you would use this command:

.NET CLI

```
dotnet new aspire
```

Or to use the *.NET Aspire Starter Application* template, you would use this command:

.NET CLI

```
dotnet new aspire-starter
```

A benefit of using Visual Studio is you choose your configuration options through dialogs. The .NET CLI commands are useful when you want to create a new .NET Aspire project with default settings, but you can alter the defaults with optional flags. In the next exercise, you'll see how to create a new .NET Aspire project using both methods.

Explore the structure of solutions generated by the .NET Aspire templates

The .NET Aspire templates generate a solution with a specific structure. This structure is a simplified diagram for the starter application, without the caching or testing options enabled:

Directory

```
└──📂 AspireSample
    ├──📂 AspireSample.ApiService
    │   ├──📂 Properties
    │   │   └── launchSettings.json
    │   ├── appsettings.json
    │   ├── AspireSample.ApiService.csproj
    │   └── Program.cs
    ├──📂 AspireSample.AppHost
    │   ├──📂 Properties
    │   │   └── launchSettings.json
    │   ├── appsettings.json
    │   ├── AspireSample.AppHost.csproj
    │   └── Program.cs
    ├──📂 AspireSample.ServiceDefaults
    │   ├── AspireSample.ServiceDefaults.csproj
    │   └── Extensions.cs
    ├──📂 AspireSample.Web
    │   ├──📂 Components
    │   │   ├──📂 Layout
    │   │   │   ├── ...
    │   │   ├──📂 Pages
    │   │   │   ├── ...
    │   │   ├── ...
    │   ├──📂 Properties
    │   │   └── launchSettings.json
    │   ├──📂 wwwroot
    │   │   ├──...
    │   ├── appsettings.json
    │   ├── AspireSample.Web.csproj
    │   ├── Program.cs
    │   └── WeatherApiClient.cs
    └── AspireSample.sln
```

Both templates add **AppHost** and **ServiceDefaults** projects. These projects are the core of an application built with .NET Aspire. The **AppHost** project is the entry point and is responsible for acting as the orchestrator.

The **ServiceDefaults** project contains the default configuration for the application. These configurations are reused across all the projects in your solution.

The above solution also includes **Web** and **ApiService** projects. The **Web** project is a Blazor WebAssembly app that has a counter and calls the **ApiService** to get forecast data. The **ApiService** is a simple API that returns forecast data.

Walk through the code structure

The **AspireSample.AppHost** project has the following code in *Program.cs*:

C#

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

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

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

builder.AddProject<Projects.AspireStarterSample_Web>("webfrontend")
    .WithReference(cache)
    .WithReference(apiService);

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

Walking through the above code, line by line:

- Create a `builder` variable that's a `IDistributedApplicationBuilder`.
- Create a `cache` variable that's a `IResourceBuilder<RedisResource>`.
- Call `AddProject` with a generic-type parameter containing the project's `IServiceMetadata` details, adding the **AspireSample.ApiService** project to the application model.

  This is a fundamental building block of .NET Aspire. The `AddProject` configures service discovery and communication between the projects in your app. The name argument **apiservice** is used to identify the project in the application model, and used later by projects that want to communicate with it.
- Calls `AddProject` again, this time adding the **AspireSample.Web** project to the

application model. It also chains multiple calls to `WithReference` passing the **cache** and **apiservice** variables.

The `WithReference` API is another fundamental API of .NET Aspire, which injects either service discovery information or connection string configuration into the project being added to the application model.

- Finally, the `builder` calls `Build` and `Run` to start the application.

---

100 XP

Exercise - Create a new .NET Aspire project

10 minutes

Before you begin working on a new service for your company's latest project, you want to check your system has all the prerequisites for .NET Aspire. The best way to check is create a new .NET Aspire project with a starter template.

In the exercise you'll install all the prerequisites, and then you'll create a new .NET Aspire Starter app. Then you'll see how to add a caching component using Redis to the app. Finally, you'll test the application and quickly explore the Aspire Dashboard.

Visual Studio

Choose this tab to see the steps in this exercise for *Visual Studio*.

Install prerequisites

We discussed the prerequisites in the previous unit. Let's walk through installing them now.

Install .NET 8

Follow this .NET 8 link, and select the correct installer for your operating system. For example, if you're using Windows 11, and a modern processor, select the x64 .NET 8 SDK for Windows.

After the download is complete, run the installer and follow the instructions. In a terminal window, run the following command to verify that the installation was successful:

.NET CLI

```
```

You should see the version number of the .NET SDK you installed. For example:

Console

```
8.0.300-preview.24203.14
```

Install Visual Studio 2022 Preview

Follow this Visual Studio 2022 Preview link, and select **Download Preview**. After the download is complete, run the installer and follow the instructions.

Install Docker Desktop

Follow this Docker Desktop link, and select the correct installer for your operating system. After the download is complete, run the installer and follow the instructions.

Open the **Docker Desktop** application and accept the service agreement.

Install the .NET Aspire workload

Install the .NET Aspire workload using Visual Studio:

1. Open the **Visual Studio Installer**.
2. Select **Modify** next to **Visual Studio**.
3. Select the **ASP.NET and web development** workload.
4. On the **Installation details** panel, select **.NET Aspire SDK (Preview)**.
5. Select **Modify** to install the .NET Aspire component.
6. Check that the latest version of .NET Aspire is installed, in a new terminal run this command:

   .NET CLI

After installing the workload, you see:

Console

```
Installed Workload Id      Manifest Version      Installation Source
----------------------------------------------------------------------------
------------------
aspire                     8.0.0/8.0.100         SDK 8.0.300-preview.24203,
VS 17.10.34902.84

Use `dotnet workload search` to find additional workloads to install.
```

Create a new .NET Aspire Starter app

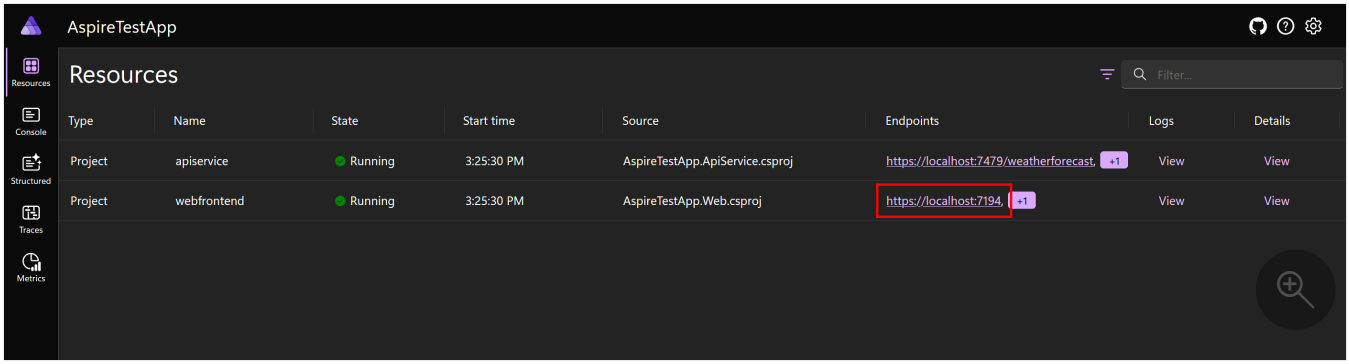
Now that the prerequisites are installed, let's create a new app.

Visual Studio

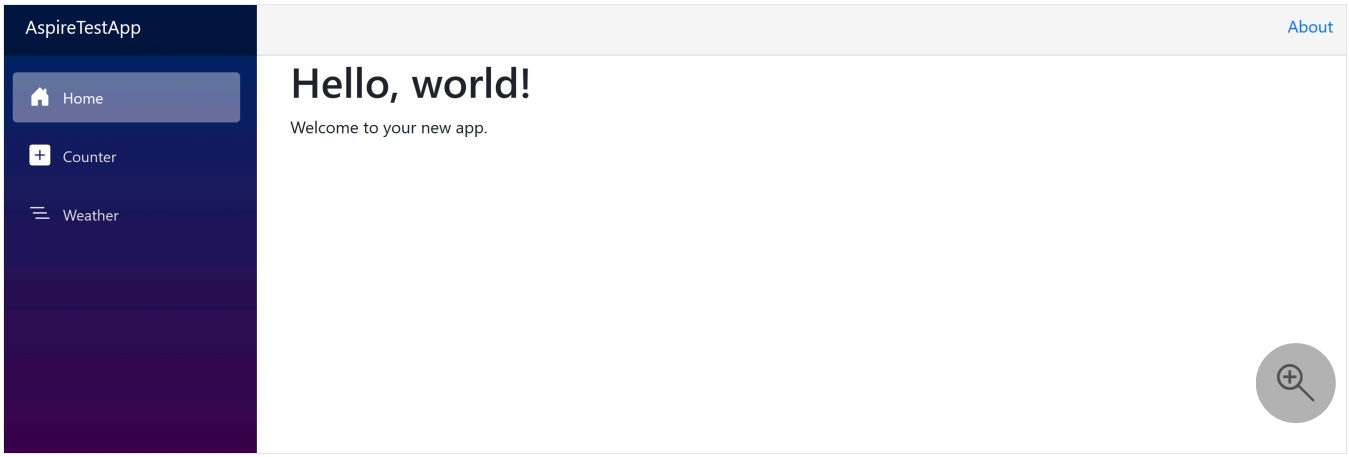
1. Open **Visual Studio**. In the dialog select **Create a new project**.
2. In the **Search for templates** box, enter **aspire**.
3. Select **.NET Aspire Starter Application**, and then select **Next**.
4. In the **Solution name** box, enter **AspireTestApp**.
5. In the **Location** box, enter the folder where you want to create the new app.
6. Select **Next**.
7. Leave the default **.NET 8.0 (Long Term Support)** selected.
8. Uncheck **Use Redis for caching (requires a supported container runtime)**.

   You'll be manually adding Redis support in the next steps.
9. Select **Create**.
10. From the menu select **Debug**, and then select **Start Debugging** (Alternatively, press `F5`).
11. If prompted to start Docker Engine, select **Yes**.

The dashboard opens in your default web browser.



Select the **webfrontend** endpoint URL. The port is randomly assigned so your dashboard might not match.



The Blazor App has a simple counter page and a Weather page that calls the backend API service to get forecast data to display.

Close the browser tabs for the Blazor App and the .NET Aspire dashboard. In Visual Studio, stop debugging.

Add a caching component to a .NET Aspire project

Now let's add a Redis caching component to the .NET Aspire project. We'll start with the app host:

Visual Studio

1. In **Solution Explorer**, right-click on the **AspireTestApp.AppHost** project, and select **Manage NuGet Packages**.
2. Select the **Browse** tab, and select **Include prerelease**.
3. Search for **aspire redis**, and select the **Aspire.Hosting.Redis** package.
4. In the right pane, for **Version** select the latest **8.0.0**, and then select **Install**.
5. In the **License Acceptance** dialog, select **I Accept**.

1. To add the Redis configuration to the app host project, open the *AspireTestApp.AppHost/Program.cs* file and add this code:

C#

```
// Add Redis
var redis = builder.AddRedis("cache");
```

This code configures the orchestration to create a local Redis container instance.

2. Change the current **webfrontend** service to use the Redis cache. Change this code:

```C#
builder.AddProject<Projects.AspireTestApp_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(apiService);
```

To this code:

```C#
builder.AddProject<Projects.AspireTestApp_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(apiService)
    .WithReference(redis);
```

The `WithReference` extension method configures the UI to use Redis automatically for output caching.

Next, we can use Redis in the consuming project.

Visual Studio

1. In **Solution Explorer**, right-click on the **AspireTestApp.Web** project, and select **Manage NuGet Packages**.
2. Select the **Browse** tab, and select **Include prerelease**.
3. Search for **aspire redis**, and select the **Aspire.StackExchange.Redis.OutputCaching** package.
4. In the right pane, for **Version** select the latest **8.0.0**, and then select **Install**.
5. In the **License Acceptance** dialog, select **I Accept**.

Now use Visual Studio to add code to use the Redis component.

1. If you need to, open the **AspireTestApp** solution in Visual Studio.

2. In **Solution Explorer**, under the **AspireTestApp.Web** project, select *Program.cs*.

3. Add this code under `var builder = WebApplication.CreateBuilder(args);`:

   C#

   ```
   // Add Redis caching
   builder.AddRedisOutputCache("cache");
   ```

   This code:

   - Configures ASP.NET Core output caching to use a Redis instance with the specified connection name.
   - Automatically enables corresponding health checks, logging, and telemetry.

4. Replace the contents of **AspireTestApp.Web/Components/Pages/Home.razor** with the following code:

   razor

   ```
   @page "/"
   @attribute [OutputCache(Duration = 10)]

   <PageTitle>Home</PageTitle>

   <h1>Hello, world!</h1>

   Welcome to your new app on @DateTime.Now
   ```

   In the preceding code, the `OutputCache` attribute specifies a 10-second duration. After the page is cached, every subsequent request within the 10-second window receives the cached output.

You can see that Aspire is designed to make it easy to add new components to your application. You add a new component to your application by adding a NuGet package, and then add a few lines of code to the *Program.cs* file in the **Web** and **AppHost** projects. Aspire then automatically configures the Redis container and the output caching for you.

Test the application

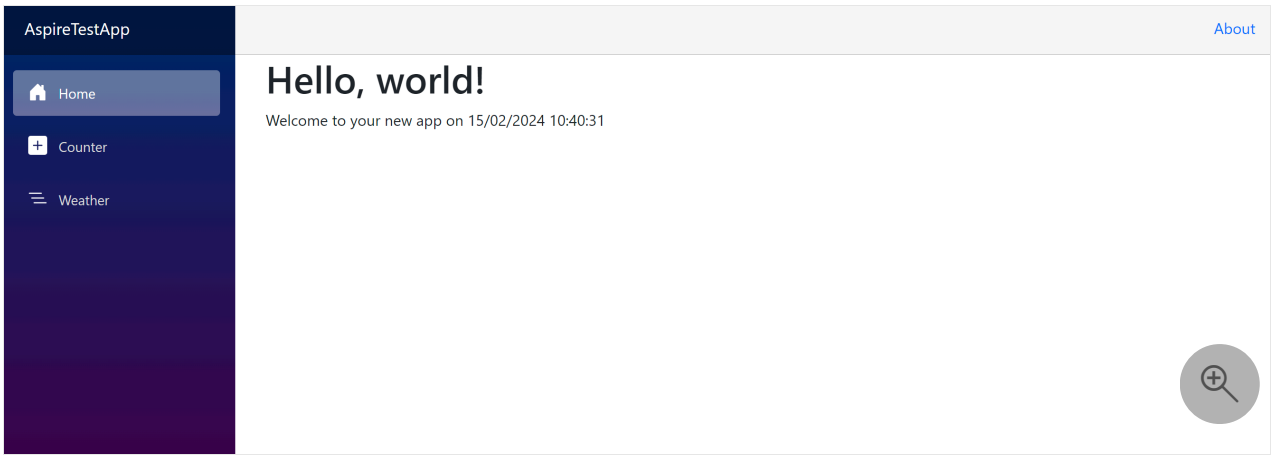
Now let's run the application to see the caching in action. In Visual Studio:

1. From the menu select **Debug**, and then select **Start Debugging** (Alternatively, press F5).

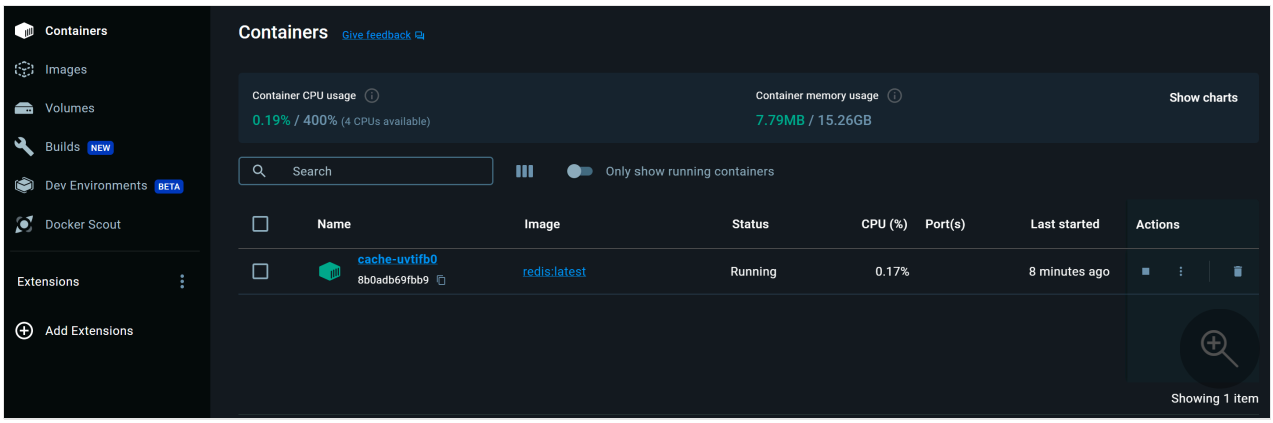
The solution builds, and the Aspire Dashboard opens in your default web browser.

2. Select the **Endpoint** URL for the **webfrontend** service to view the home page of the application.

3. In the browser, refresh the page a few times. The time on the page doesn't change within the 10 second cache duration.



The solution creates a Redis container. Open **Docker Desktop** to see the container running.



4. To stop the solution running in Visual Studio, press `Shift`+`F5`.

5. Open **Docker Desktop**, and select **Containers/Apps**. You should see the **redis:latest** is no longer running.

You've seen how easy it is to add a new caching component to an application using .NET Aspire. You added a NuGet package, and then added a few lines of code. .NET Aspire automatically configured the Redis container and the output caching for you.

100 XP

How to add orchestration to an existing .NET app

5 minutes

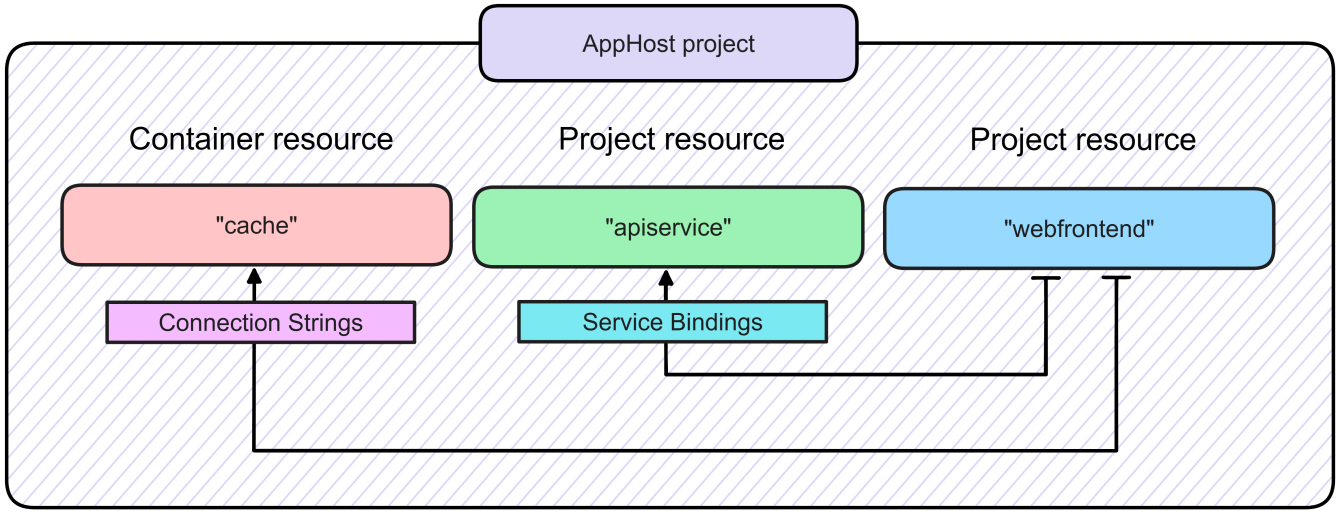
.NET Aspire can be broken down into solutions for three issues that face modern cloud-native apps. These are:

1. Managing the complexity of orchestration between microservices.
2. Simplifying how your microservices use components through NuGet packages.
3. Supporting developer velocity through tooling and templates.

In this unit, see how using .NET Aspire orchestration brings benefits to your existing cloud-native apps. Then see how to enroll your app in .NET Aspire orchestration and examine the changes made to your solution.

Benefits of .NET Aspire orchestration

Orchestration is the coordination and management of the various services within a cloud-native app. .NET Aspire provides abstractions for managing your solution's service discovery, environment variables, and container configurations. These abstractions also provide consistent setup patterns across apps with many components and services.



.NET Aspire has three base compute types it supports with orchestration:

- **ProjectResource**: A .NET project, such as ASP.NET Core web apps.

- **ContainerResource**: A container image, such as a Docker image containing Redis.
- **ExecutableResource**: An executable file.

Compare .NET Aspire with how you manage service discovery using Docker Compose. Docker Compose is excellent but starts to become unproductive when all you need to do is run several projects or executables. You need to build container images, compose the YAML to connect them, and then run apps inside of containers. Also, there are environment variable replacements (and includes) and no IntelliSense, and it's hard to determine what exactly is running and why. Debugging can also be difficult.

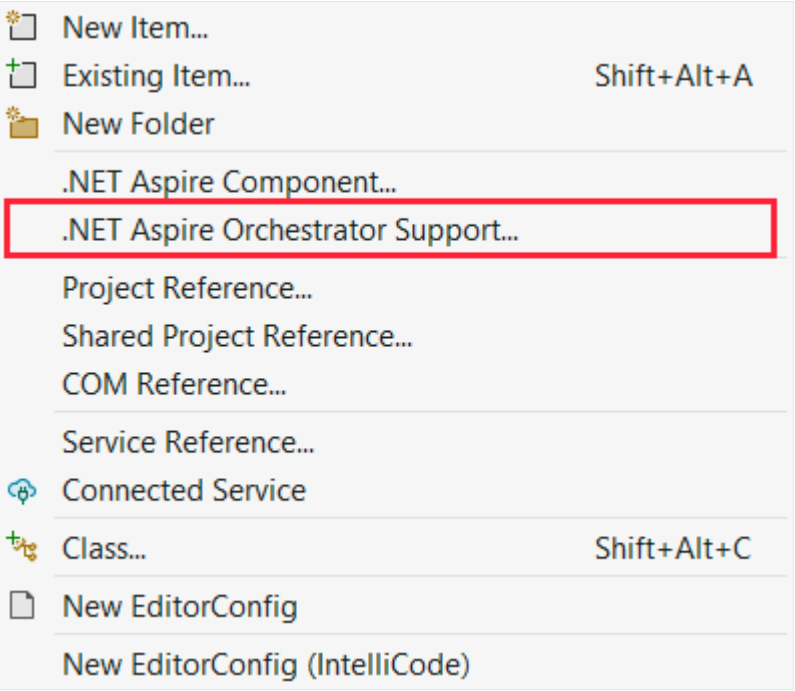
Configuration through declarative code is better. Using .NET Aspire, you don't need to learn anything beyond what you already know. .NET Aspire is a better experience that's easy to get started with and scales up to an orchestrator like Docker Compose using a real programming language.

.NET Aspire orchestration also supports your development in the following ways:

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

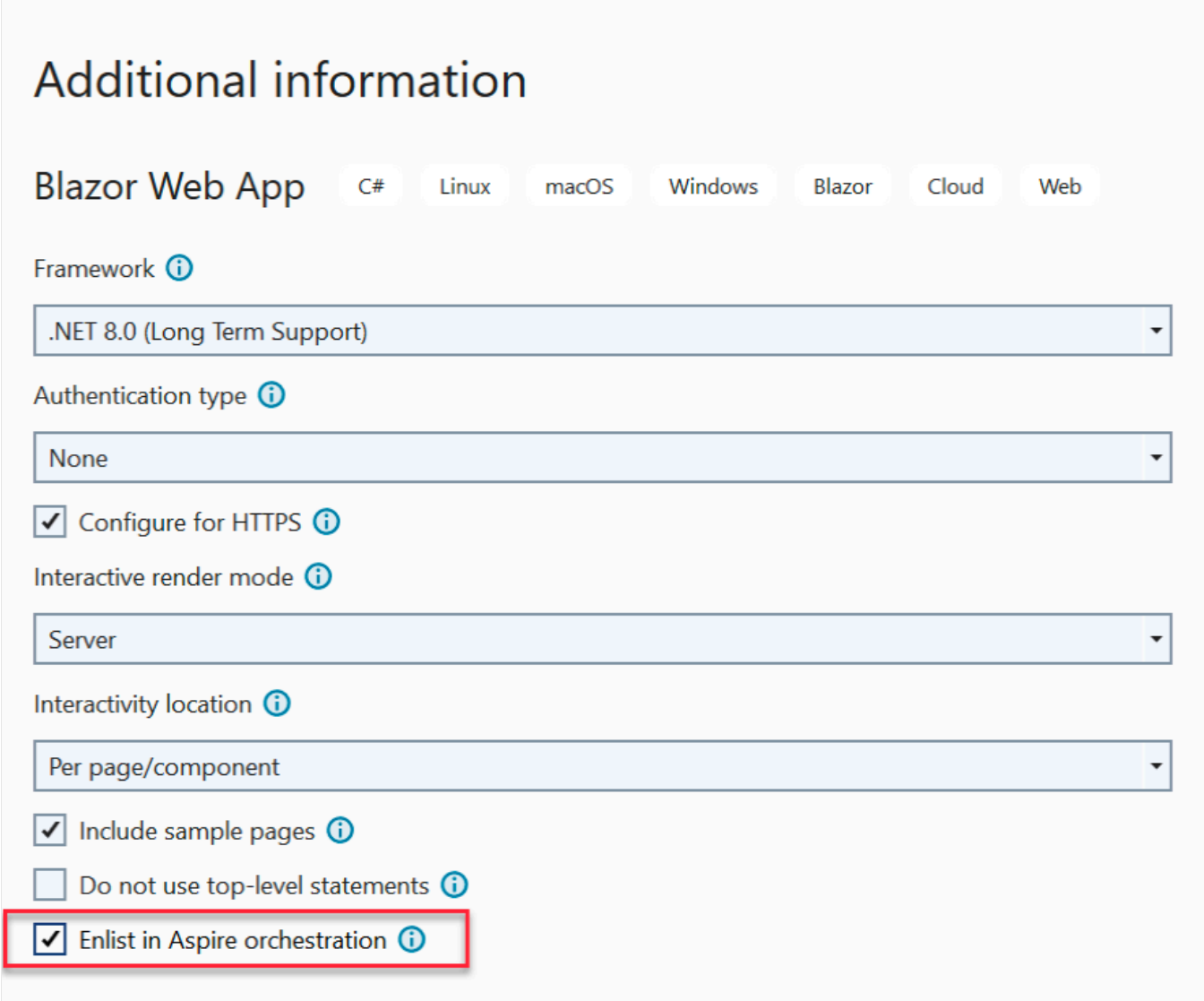
Enlisting an existing app in .NET Aspire orchestration

Visual Studio provides menus to enlist an existing project in .NET Aspire orchestration.



The first time you add orchestration to your solution a dialog asks for the project name prefix and explains that the .NET Aspire **AppHost** and **ServiceDefaults** projects are added. When you add more projects to an already orchestrated solution, the dialog notifies you that the **AppHost** project is updated to include those projects.

If you're creating a new project, during the new project workflow, Visual Studio asks if you want to enlist in .NET Aspire orchestration.



Changes Aspire makes to an existing solution

When you add .NET Aspire orchestration to your solution, the following changes happen:

- An **AppHost** project is added. The project contains the orchestration code. It becomes the entry point for your app and is responsible for starting and stopping your app. It also manages the service discovery and connection string management.
- A **ServiceDefaults** project is added. The project configures OpenTelemetry, adds default health check endpoints, and enables service discovery through `HttpClient`.
- The solution's default startup project is changed to **AppHost**.
- Dependencies on the projects enrolled in orchestration are added to the **AppHost** project.
- The .NET Aspire Dashboard is added to your solution, which enables shortcuts to access all the project endpoints in your solution.

- The dashboard adds logs, traces, and metrics for the projects in your solution.

If you add orchestration to a web app project, .NET Aspire automatically adds a reference to the **ServiceDefaults** project. It then makes the following changes to the code in *Program.cs*:

- Adds a call to `AddServiceDefaults` that enables the default OpenTelemetry, meters, and service discovery.
- Adds a call to `MapDefaultEndpoints` that enables the default endpoints, such as `/health` and `/alive`.

Next unit: Exercise - Integrate an existing ASP.NET Core web app

Continue >

100 XP

Exercise - Integrate an existing ASP.NET Core web app

8 minutes

Now let's explore how to add .NET Aspire to an existing ASP.NET Core web app. Along the way, you'll learn how to add the .NET Aspire stack to an existing ASP.NET Core web app and then run the app. You'll also see how to call microservices from the ASP.NET Core app.

Set up your environment

To add .NET Aspire to an existing demo ASP.NET Core web app, you need to first obtain the existing app.

In a terminal window:

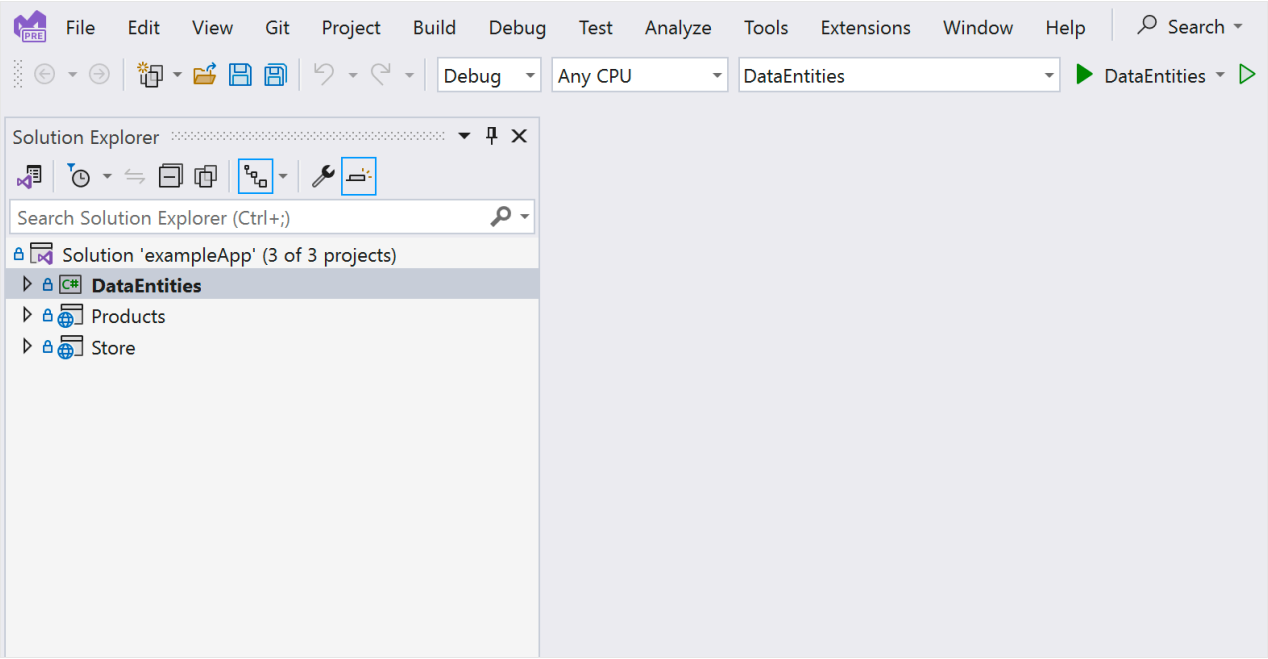
1. Set the current working directory to where you want to store your code.
2. Clone the repository into a new folder named *ExampleApp*:

```bash
git clone https://github.com/MicrosoftDocs/mslearn-aspire-starter ExampleApp
```

Explore the example app

Use Visual Studio to explore the demo app.

1. Open Visual Studio, then select **Open a project or solution**.
2. In the **Open Project/Solution** dialog, navigate to the *ExampleApp/eShopAdmin* folder, then select *EShopAdmin.sln*.
3. Select **Open**.

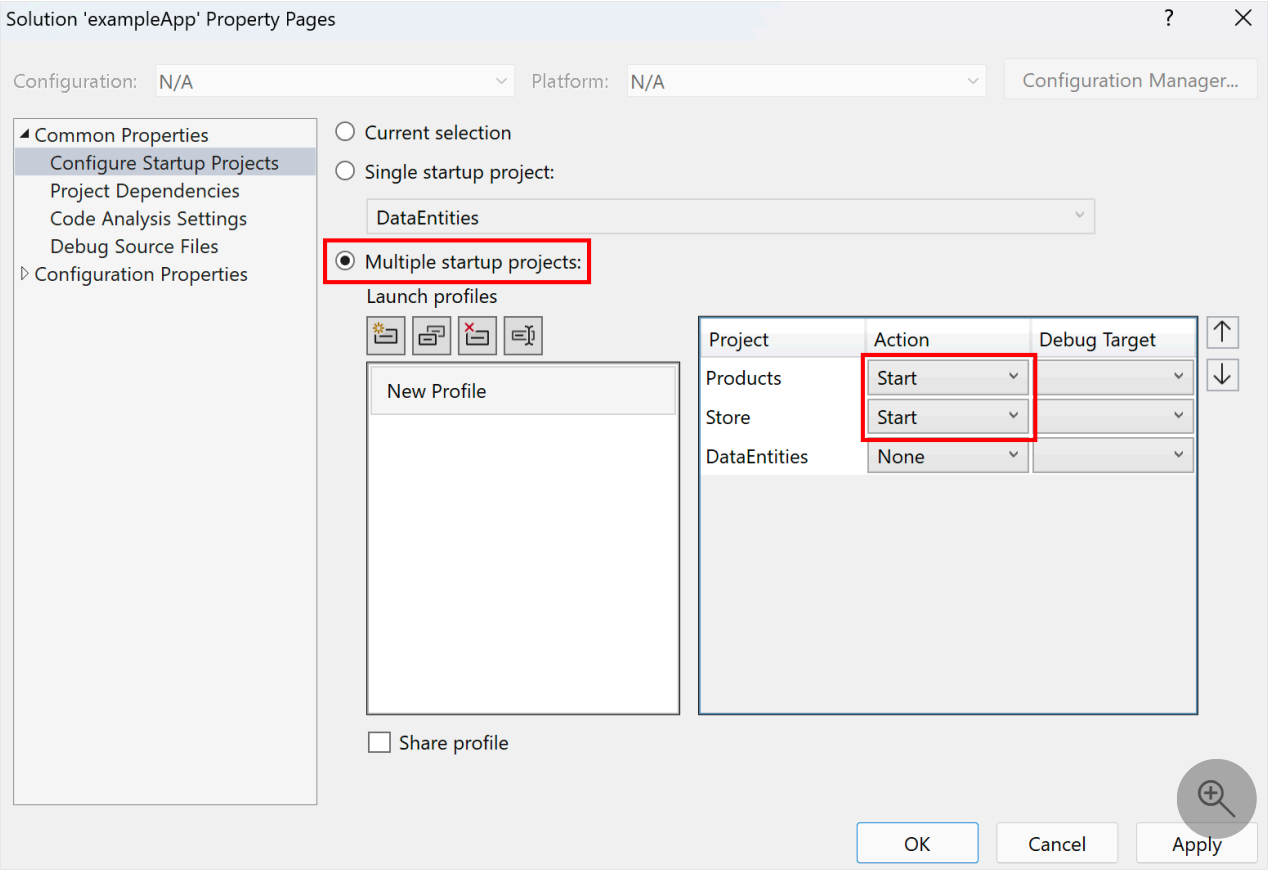


The demo app is a .NET 8 solution with three projects:

- **Data Entities**. A class library that defines the **Product** class used in the Web App and Web API.
- **Products**. A Web API that returns a list of products in the catalog with their properties.
- **Store**. A Blazor Web App displays these products to website visitors.

To successfully run the app, change the projects that start up:

1. From the menu, select **Project** > **Configure Startup Projects...**.
2. In the **Solution Property Pages** dialog, select **Multiple startup projects**.



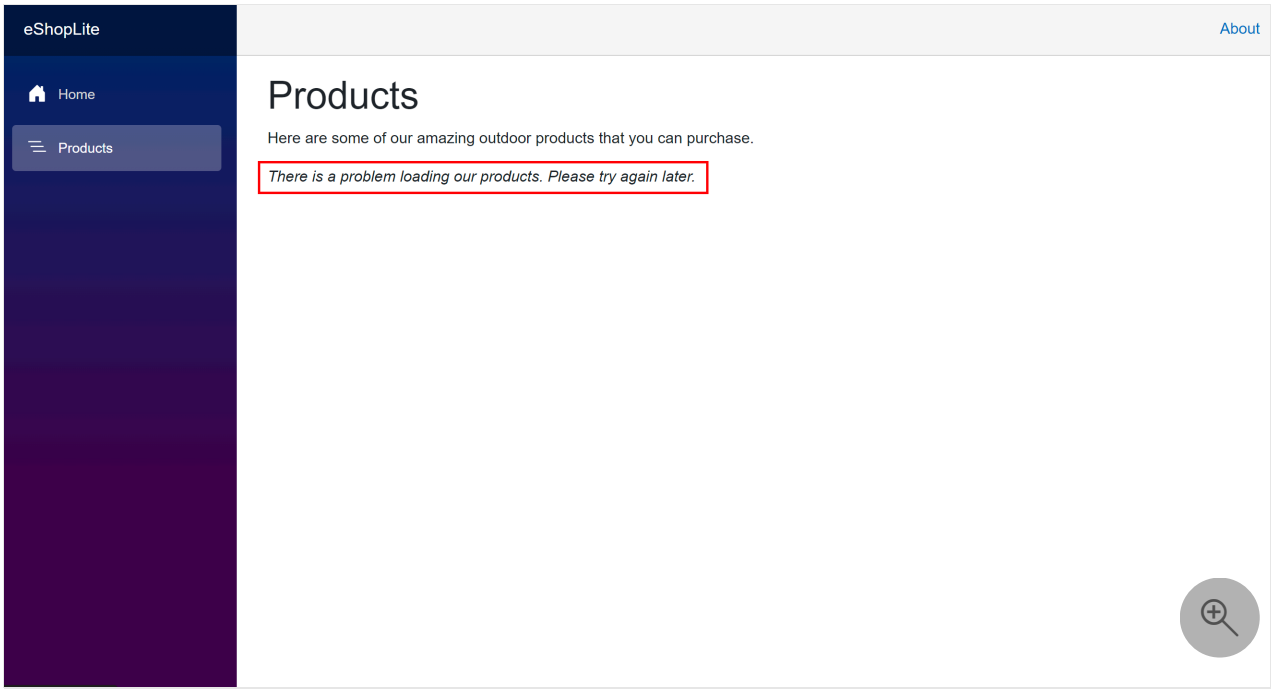
3. In the **Action** column, set **Products** and **Store** to *Start*.

4. Select **OK**.

5. Run the app by pressing F5, or by selecting **Start** in the toolbar.

   The app opens two instances of your default web browser. One shows the JSON output of the Web API, and the other shows the Blazor Web App.

6. In the web app, select **Products** from the menu. You should see this error.



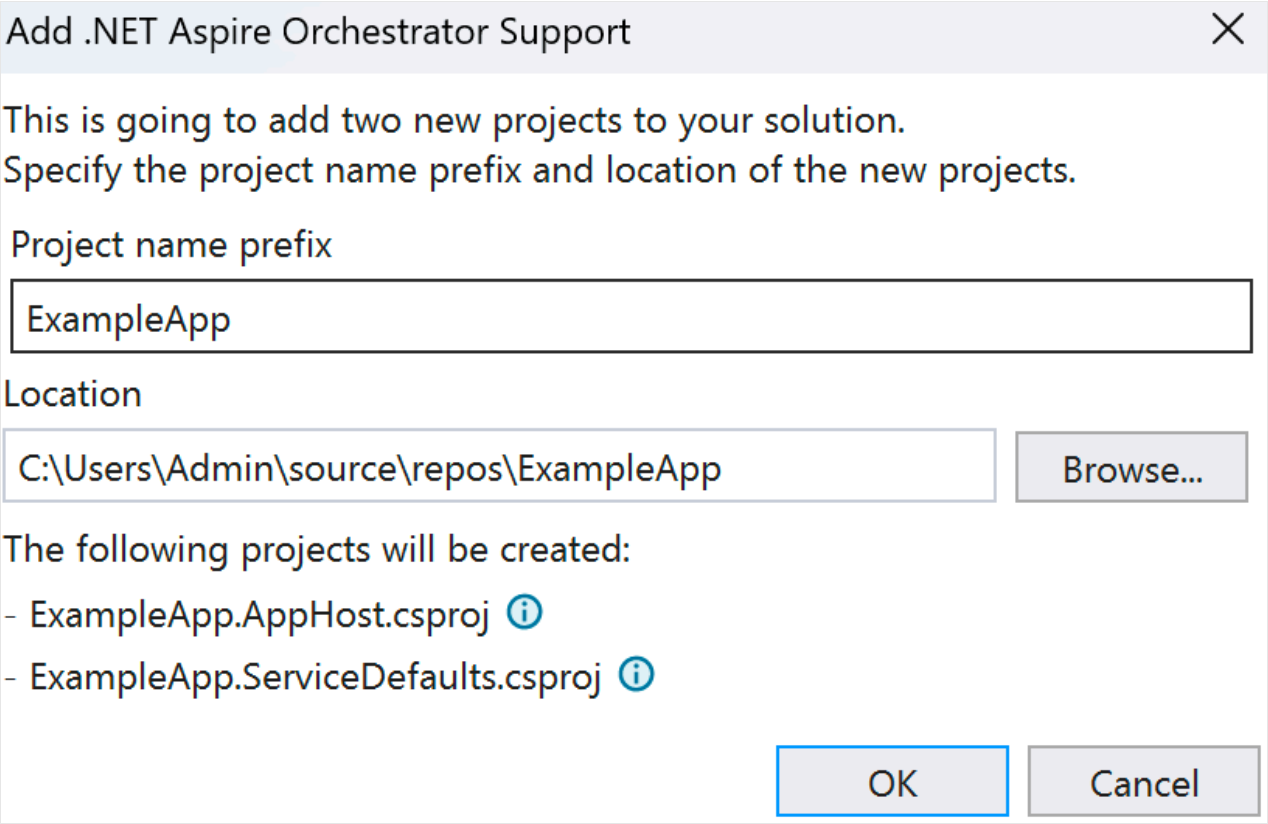
7. Stop the app by pressing `Shift`+`F5`, or select **Stop Debugging** in the toolbar.

This app is new to you. You're not sure how the endpoints and services are configured. Let's add .NET Aspire orchestration and see if it can help diagnose the problem.

Enlist the existing app in .NET Aspire orchestration

In Visual Studio:

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

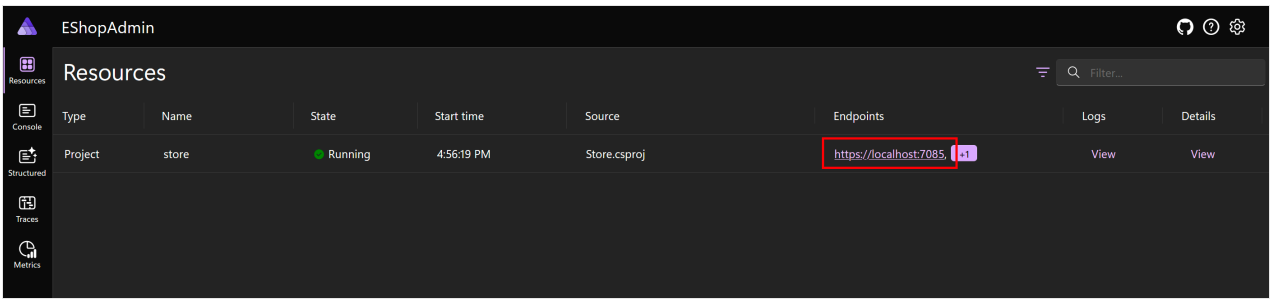


2. In the **Add .NET Aspire Orchestrator Support** dialog, select **OK**.

   Now you can see the **AppHost** and **ServiceDefaults** projects are added to the solution. The **AppHost** project is also set as the startup project.

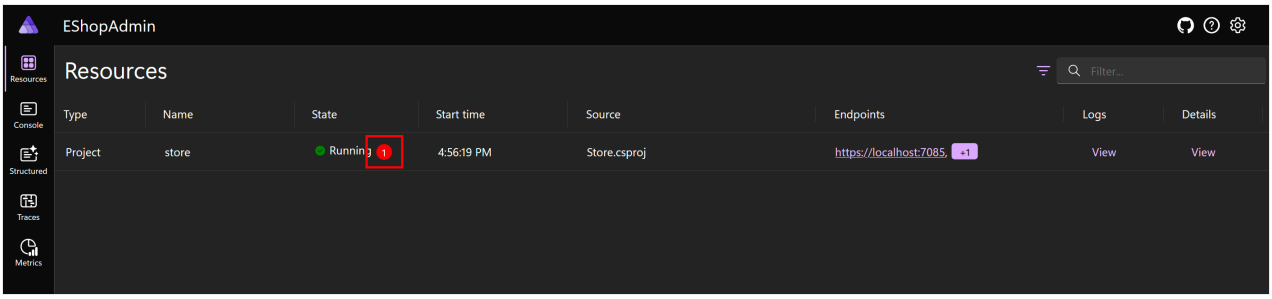
3. Run the app by pressing F5, or by selecting **Start** in the toolbar.

   This time, the solution opens a single browser window showing the .NET Aspire dashboard.

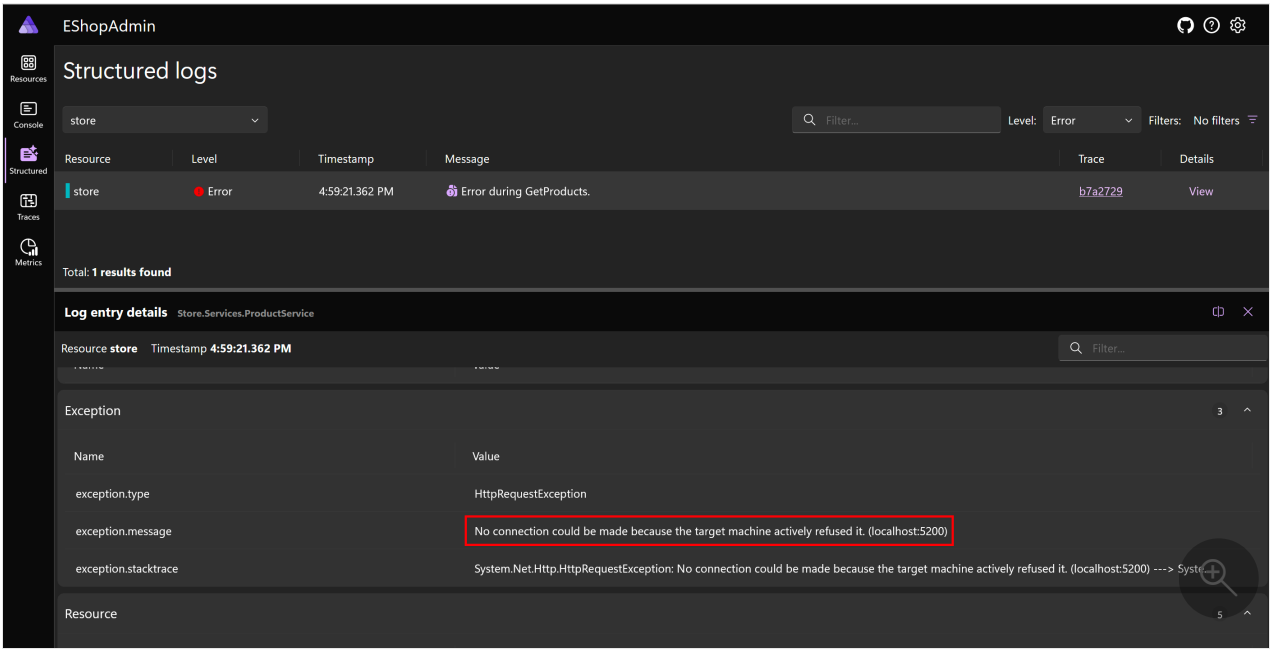


4. Select the endpoint for the **store** project, and then select **Products**. You should see the same error as before.

5. Return to the dashboard.



6. Select the red error notification next to **Running**, and then in the **Details** column, select **View**.



7. Scroll through the error details until you can see the **exception.message**. The Web app is struggling to connect to **localhost:5200**. Port 5200 is the port the front end thinks the products API is running on.

8. Stop the app by pressing `Shift`+`F5`, or select **Stop Debugging** in the toolbar.

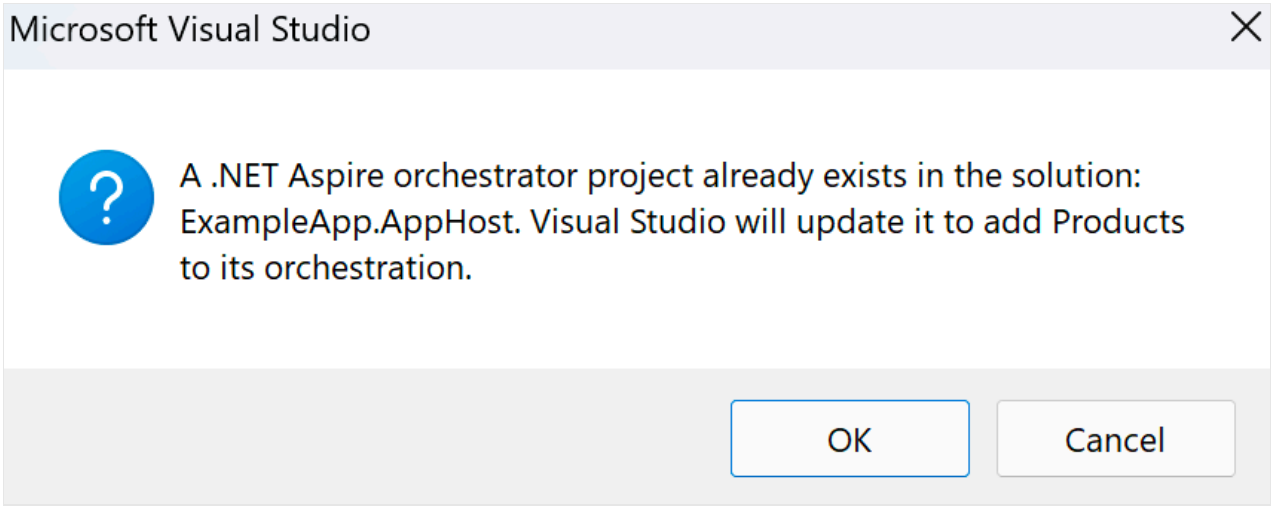
Update the orchestration to include the products API

There are many ways to resolve this issue. You could explore the code, work out how the front end is configured, edit the code, or change the **launchSettings.json** or **appsettings.json** files.

With .NET Aspire, you can change the orchestration so that the products API responds on port 5200.

In Visual Studio:

1. To add the *Products* project to the orchestration, in **Solution Explorer**, right-click the *Products* project, then select **Add** > **.NET Aspire Orchestrator Support...**.



2. In the dialog, select **OK**.

3. In **Solution Explorer**, open the *AppHost* project, then open the *Program.cs* file.

   Explore the code and see how the **Products** project is added to the orchestration:

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

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

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

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

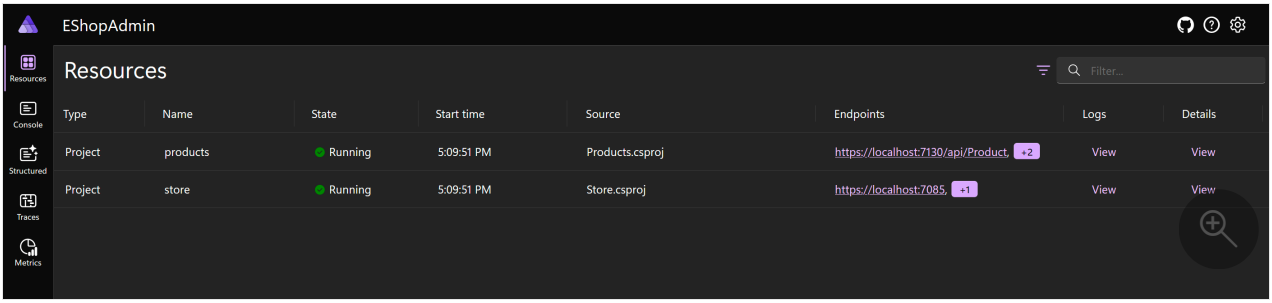
   Next configure .NET Aspire to add a new end point for the products API.

4. Replace the `builder.AddProject<Projects.Products>("products");` line with this code:

```C#
builder.AddProject<Projects.Products>("products")
       .WithHttpEndpoint(port: 5200, name: "products");
```

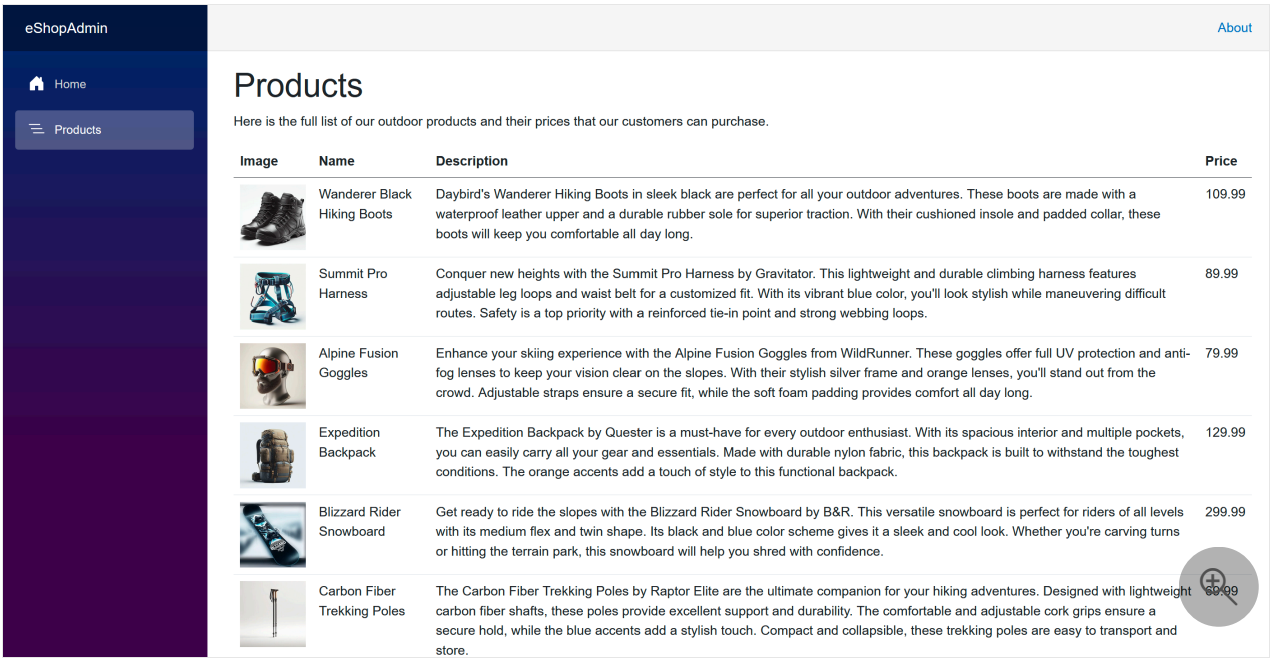
5. Run the app by pressing `F5`, or by selecting **Start** in the toolbar.

   The **products** project is now listed on the dashboard with two endpoints.



6. Select the endpoint for the **Store** project, and then select **Products**.

   The products are now displayed in the web app.



7. Stop the app by pressing `Shift`+`F5`, or select **Stop Debugging** in the toolbar.

This approach resolves the configuration issue, but it isn't the best way to fix the problem. If you decide to use .NET Aspire, you should also use .NET Aspire service discovery.

Update the web app to use .NET Aspire service discovery

In Visual Studio:

1. In **Solution Explorer**, open the **AppHost** project, then open the *Program.cs* file.

2. Replace the code in the file with this code:

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

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

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

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

   The above code reorders the projects. The products API is now passed as a project reference to the front end Store web app.

3. In **Solution Explorer**, open the **Store** project, then open the *appsettings.json* file.

4. Delete the endpoint configuration lines:

```JSON
"ProductEndpoint": "http://localhost:5200",
"ProductEndpointHttps": "https://localhost:5200"
```

   The settings are now:

```JSON
{
  "DetailedErrors": true,
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

5. In **Solution Explorer**, under the **Store** project, open the *Program.cs* file.

6. Replace this line:

```C#
var url = builder.Configuration["ProductEndpoint"]
     ?? throw new InvalidOperationException("ProductEndpoint is not
set");
```

   with this line:

```C#
var url = "http://products";
```

7. Run the app by pressing F5, or by selecting **Start** in the toolbar.

8. Select the endpoint for the **Store** project, then select **Products**.

The app is still working as expected, but the front end is now using .NET Aspire service discovery to get information about the products API endpoint.

Next unit: Knowledge check

Continue >

200 XP

Knowledge check

2 minutes

1. Which of these prerequisites must be present on your computer, before you install .NET Aspire? *

- Docker Desktop or Podman
- Docker Compose
- Kubernetes

2. You enlist your application in the .NET Aspire orchestration. Which of the following changes would you expect to see in your solution? *

- A new Redis cache container for outbound HTTP requests.
- The web app project is set to the new start-up project.
- A new ServiceDefaults project is added to the solution.

Check your answers

100 XP

Summary

1 minute

Congratulations! In this module, you learned how to create a brand-new .NET Aspire solution and how to add .NET Aspire to an existing cloud-native application.

You also saw how .NET Aspire takes care of many tasks that can be challenging for cloud-native apps, such as resolving a service discovery problem, with only a few lines of code. .NET Aspire also simplifies configuration tasks, and adding common services like databases, messaging, and caching.

Having completed this module, you can:

- Create new cloud-native apps by using the .NET Aspire templates in Visual Studio.
- Add the .NET Aspire stack to an existing .NET app for orchestration and simple cloud-native components.

Next steps

Check out more .NET Aspire resources!

- .NET Aspire documentation