.NET Aspire overview

Article • 01/26/2025

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

Why .NET Aspire?

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

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

Dev-time orchestration

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

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

.NET Aspire orchestration assists with the following concerns:

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

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

C#

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

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

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

For more information, see .NET Aspire orchestration overview.

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

.NET Aspire integrations

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

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

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

```
C#

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

The AddAzureServiceBusClient method handles the following concerns:

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

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

Project templates and tooling

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

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

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

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

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

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

Next steps

Quickstart: Build your first .NET Aspire project

Quickstart: Build your first .NET Aspire solution

Article • 11/07/2024

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

In this quickstart, you explore the following tasks:

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

Prerequisites

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

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

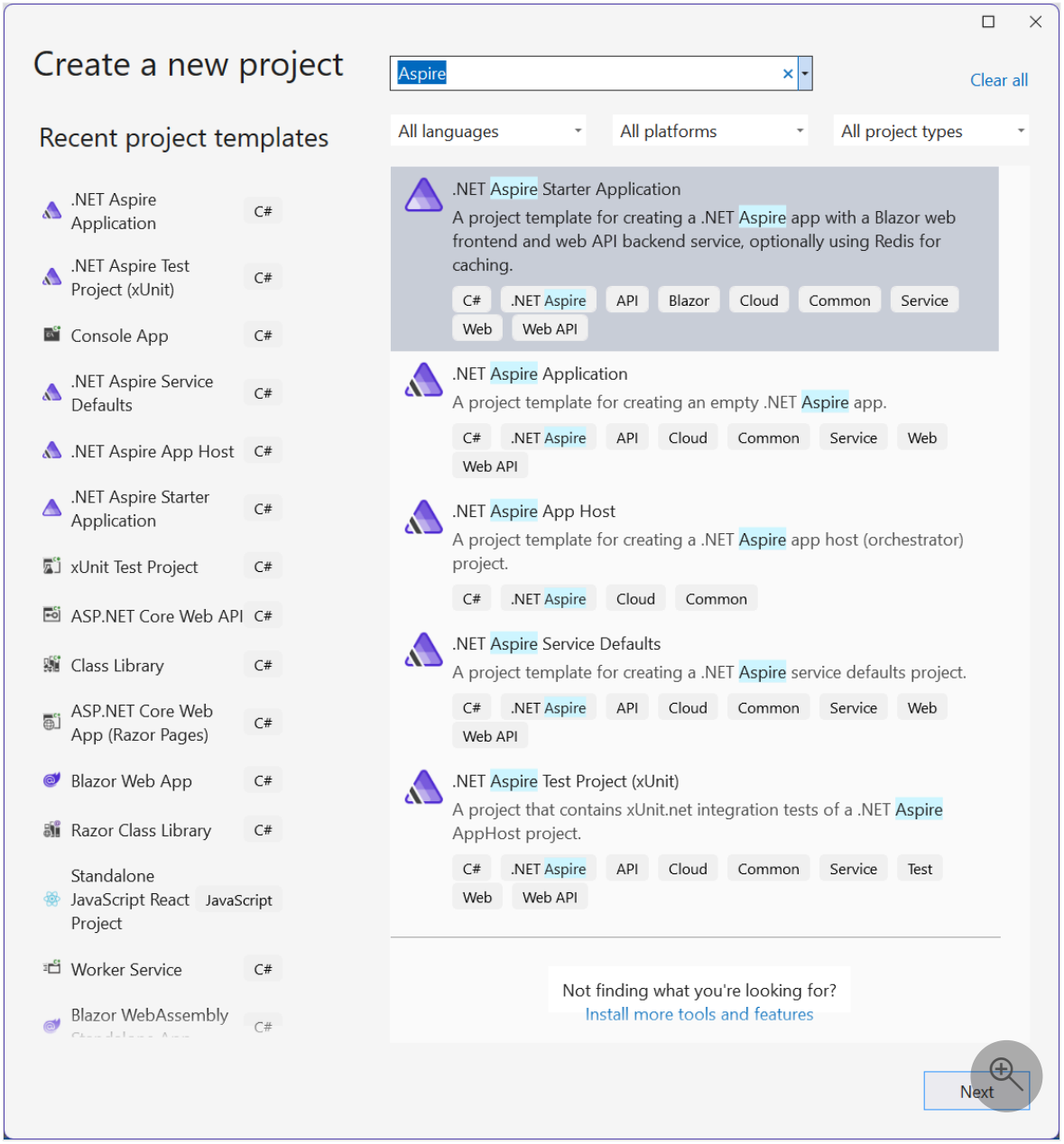
For more information, see .NET Aspire setup and tooling, and .NET Aspire SDK.

Create the .NET Aspire template

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

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

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



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

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

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

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

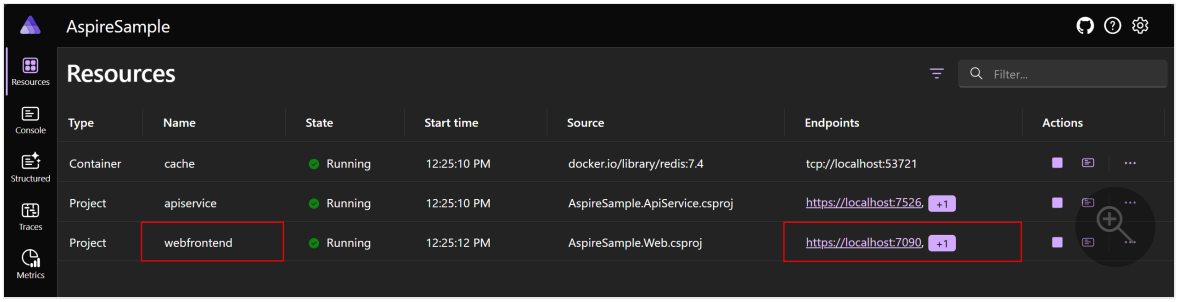
Test the app locally

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

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

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

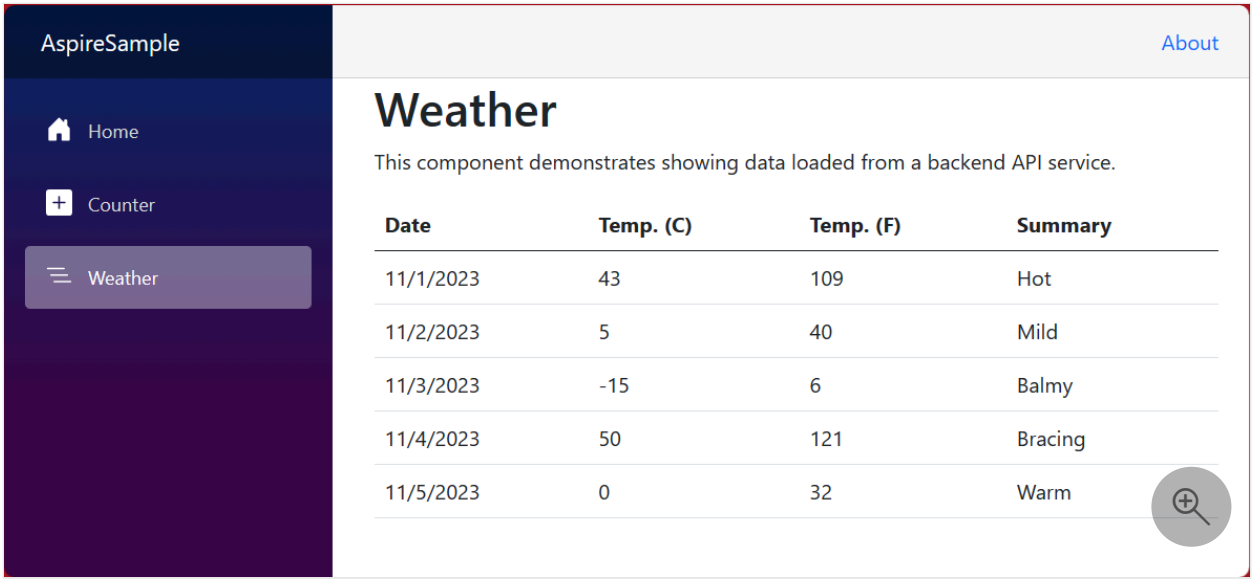
1. The app displays the .NET Aspire dashboard in the browser. You look at the dashboard in more detail later. For now, find the **webfrontend** project in the list of resources and select the project's **localhost** endpoint.



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

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

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



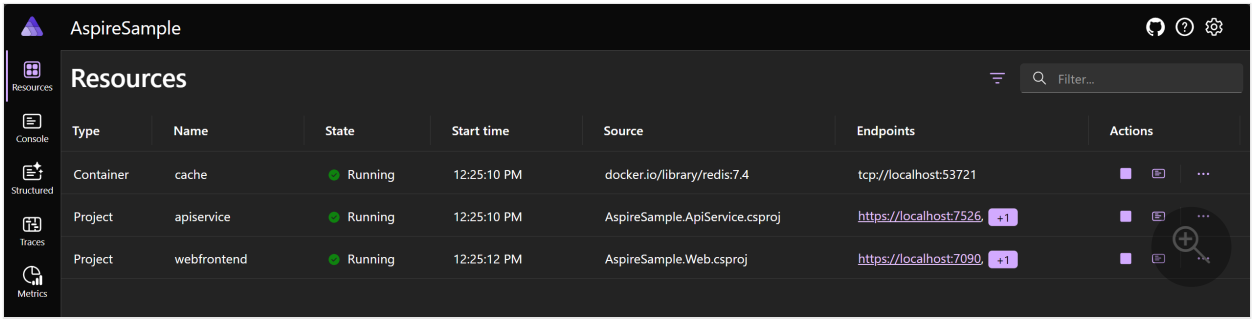
🤓 Congratulations! You created and ran your first .NET Aspire solution! To stop the app, close the browser window.

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

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

Explore the .NET Aspire dashboard

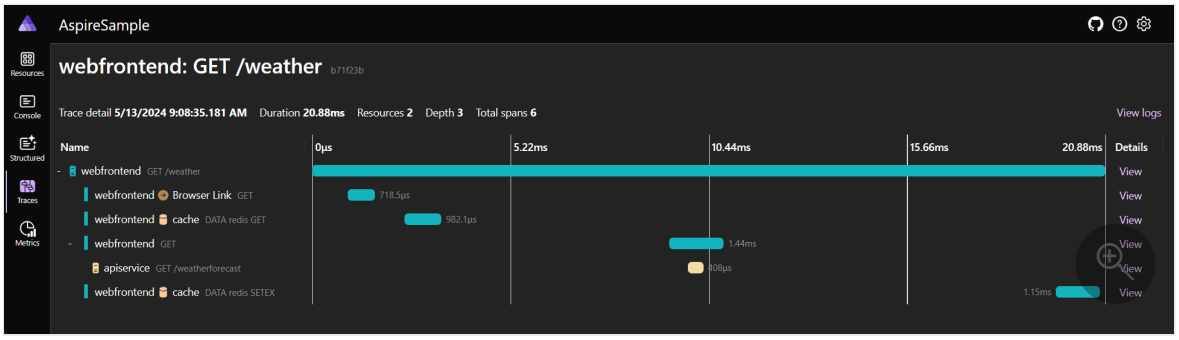
When you run a .NET Aspire project, a dashboard launches that you use to monitor various parts of your app. The dashboard resembles the following screenshot:



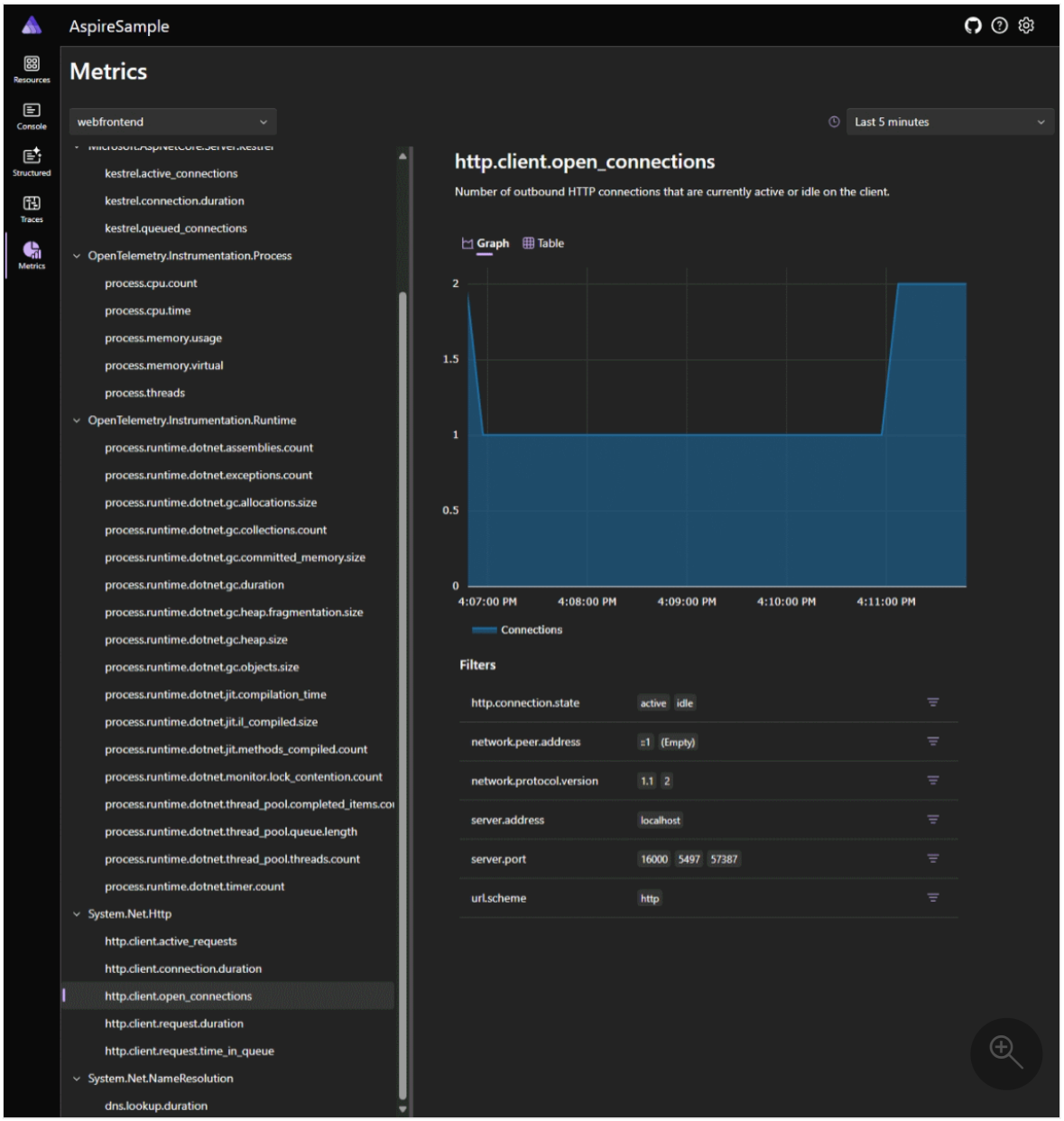
Visit each page using the left navigation to view different information about the .NET Aspire resources:

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

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



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



For more information, see .NET Aspire dashboard overview.

Understand the .NET Aspire solution structure

The solution consists of the following projects:

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

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

Your *AspireSample* directory should resemble the following structure:

Directory

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

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

Explore the starter projects

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

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

.NET Aspire host project

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

XML

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

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

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

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

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

```
</Project>
```

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

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

C#

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

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

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

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

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

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

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

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

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

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

.NET Aspire service defaults project

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

XML

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

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

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

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

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

</Project>
```

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

Orchestrate service communication

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

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

C#

```
namespace AspireSample.Web;

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

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

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

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

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

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

C#

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

var builder = WebApplication.CreateBuilder(args);

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

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

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

var app = builder.Build();

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

```
app.UseHttpsRedirection();

app.UseAntiforgery();

app.UseOutputCache();

app.MapStaticAssets();

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

app.MapDefaultEndpoints();

app.Run();
```

The preceding code:

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

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

See also

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

Next steps

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

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

Article • 04/21/2025

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

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

Prerequisites

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

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

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

Get started

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

1. Open a command prompt and change directories to where you want to store the code.
2. To clone to .NET 9 example solution, use the following `git clone` command:

```
Bash

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

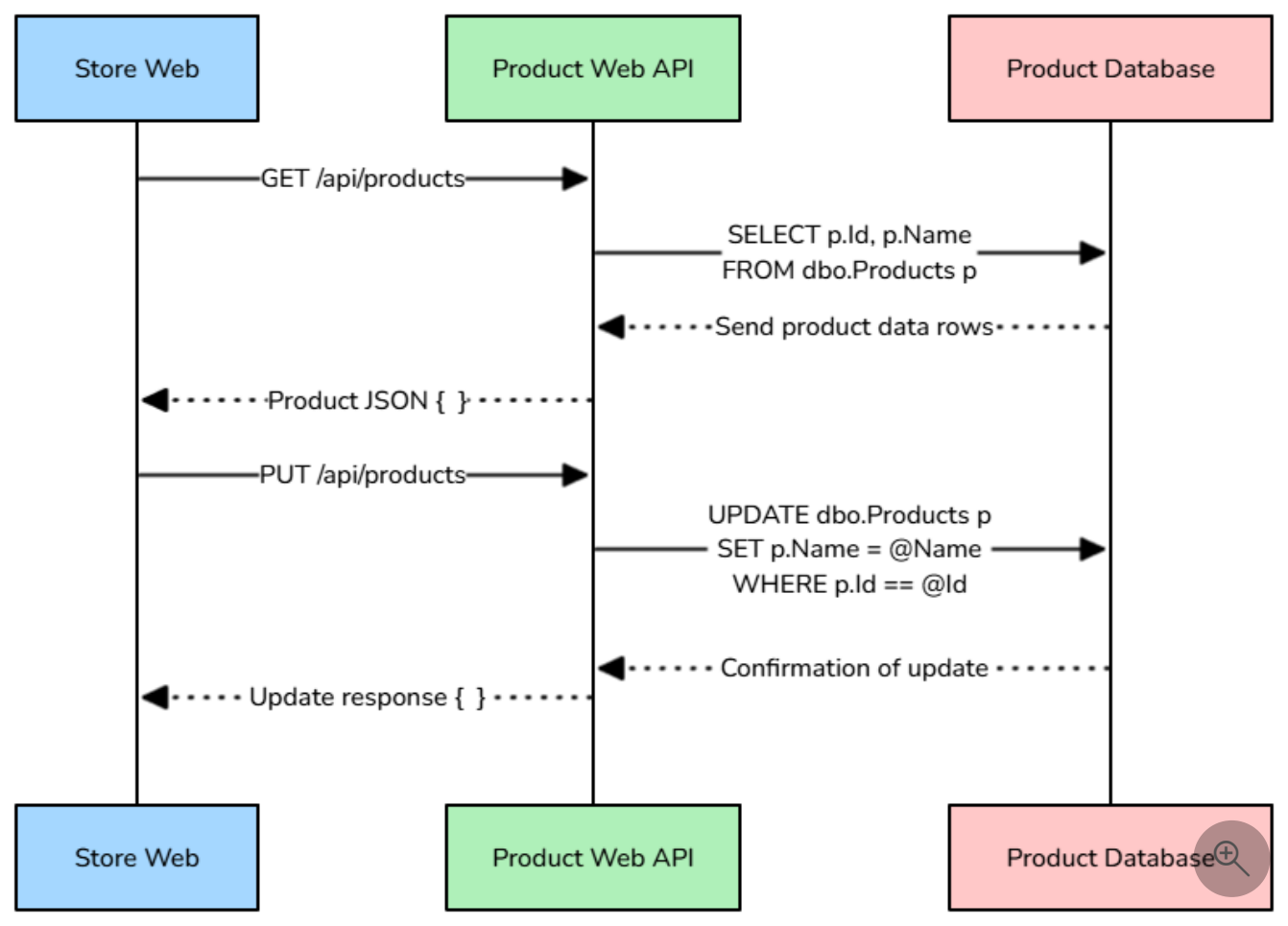
Explore the sample app

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

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

Sample app architecture

To better understand the structure of the sample app, consider the following diagram, which illustrates its simple three-tier architecture:



This layered design ensures a clear separation of concerns, making the app easier to maintain and scale.

Run the sample app

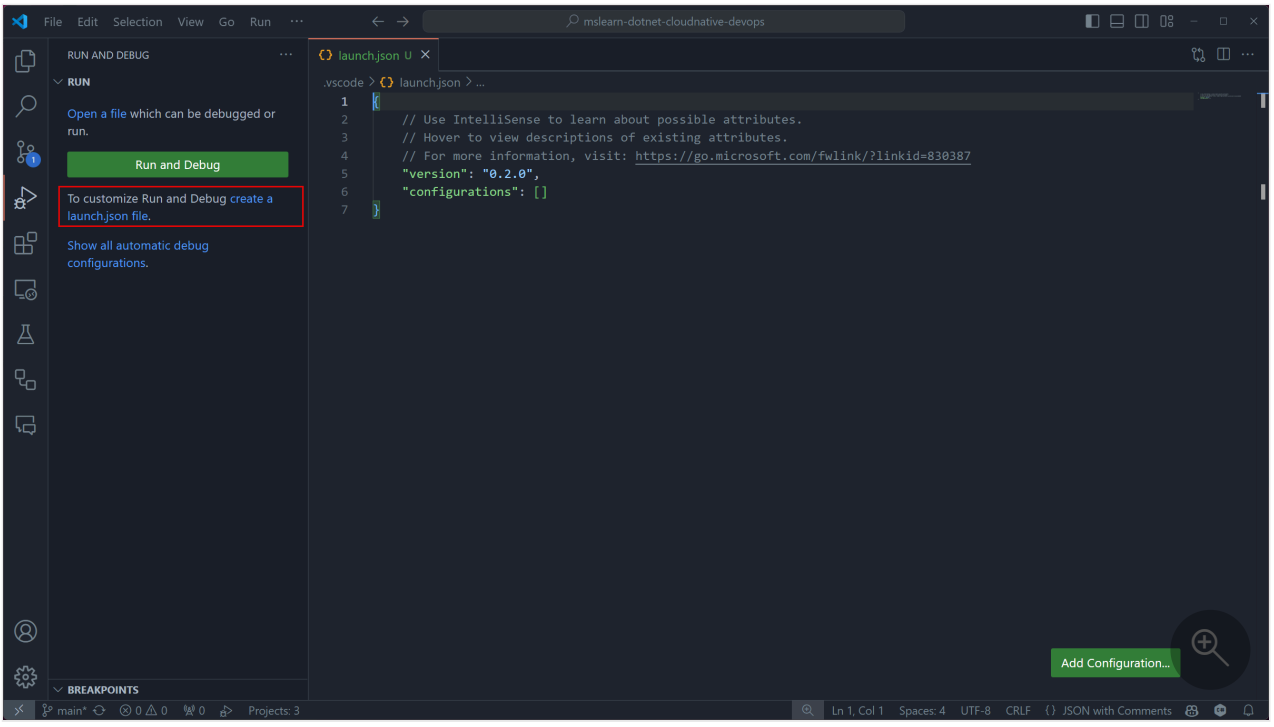
Open and start debugging the project to examine its default behavior:

1. Start Visual Studio Code and open the folder that you cloned. From the terminal where you cloned the repo, run the following command:

```Bash
code .
```

2. Select the **Run and Debug** menu item, or press `Ctrl` + `Shift` + `D`.

3. Select the **create a launch.json file** link.



4. Copy and paste the following JSON into this file and **Save**:

Windows

```JSON
{
    "version": "0.2.0",
    "compounds": [
        {
            "name": "Run all",
            "configurations": [
                "Run products",
                "Run store",
            ]
        }
    ],
    "configurations": [
        {
            "name": "Run products",
```

```
            "type": "dotnet",
            "request": "launch",
            "projectPath":
"${workspaceFolder}\\Products\\Products.csproj"
        },
        {
            "name": "Run store",
            "type": "dotnet",
            "request": "launch",
            "projectPath": "${workspaceFolder}\\Store\\Store.csproj"
        }
    ]
}
```

5. To start debugging the solution, press `F5` or select **Start**.

6. Two pages open in the browser:

    - A page displays products in JSON format from a call to the Products Web API.
    - A page displays the homepage of the website. In the menu on the left, select **Products** to see the catalog obtained from the Web API.

7. To stop debugging, close the browser, and then select the **Stop** button twice (once for each running debug instance).

No matter which tool you use—starting multiple projects manually or configuring connections between them is tedious. Additionally, the **Store** project requires explicit endpoint configuration for the **Products** API, which is both cumbersome and prone to errors. This is where .NET Aspire simplifies and streamlines the process!

Add .NET Aspire to the Store web app

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

Create an app host project

In order to orchestrate the existing projects, you need to create a new *app host* project. To create a new *app host* project from the available .NET Aspire templates, use the following .NET CLI command:

.NET CLI

```
dotnet new aspire-apphost -o eShopLite.AppHost
```

Add the *app host* project to existing solution:

Windows

.NET CLI

```
dotnet sln .\eShopLite.sln add .\eShopLite.AppHost\eShopLite.AppHost.csproj
```

Add the **Store** project as a project reference to the *app host* project using the following .NET CLI command:

Windows

.NET CLI

```
dotnet add .\eShopLite.AppHost\eShopLite.AppHost.csproj reference
.\Store\Store.csproj
```

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

Create a service defaults project

After the app host project is created, you need to create a new *service defaults* project. To create a new *service defaults* project from the available .NET Aspire templates, use the following .NET CLI command:

.NET CLI

```
dotnet new aspire-servicedefaults -o eShopLite.ServiceDefaults
```

To add the project to the solution, use the following .NET CLI command:

Windows

.NET CLI

```
dotnet sln .\eShopLite.sln add
.\eShopLite.ServiceDefaults\eShopLite.ServiceDefaults.csproj
```

Update the *app host* project to add a project reference to the **Products** project:

Windows

.NET CLI

```
dotnet add .\eShopLite.AppHost\eShopLite.AppHost.csproj reference
.\Products\Products.csproj
```

Both the **Store** and **Products** projects need to reference the *service defaults* project so that they can easily include service discovery. To add a reference to the *service defaults* project in the **Store** project, use the following .NET CLI command:

Windows

.NET CLI

```
dotnet add .\Store\Store.csproj reference
.\eShopLite.ServiceDefaults\eShopLite.ServiceDefaults.csproj
```

The same command with slightly different paths should be used to add a reference to the *service defaults* project in the **Products** project:

Windows

.NET CLI

```
dotnet add .\Products\Products.csproj reference
.\eShopLite.ServiceDefaults\eShopLite.ServiceDefaults.csproj
```

In both the **Store** and **Products** projects, update their *Program.cs* files, adding the following line immediately after their `var builder = WebApplication.CreateBuilder(args);` line:

C#

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

Update the app host project

Open the *Program.cs* file of the *app host* project, and replace its contents with the following C# code:

C#

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

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

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

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

The preceding code:

- Creates a new DistributedApplicationBuilder instance.
- Adds the **Store** project to the orchestrator.
- Adds the **Products** project to the orchestrator.
- Builds and runs the orchestrator.

Service Discovery

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

C#

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

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

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

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

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

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

JSON

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

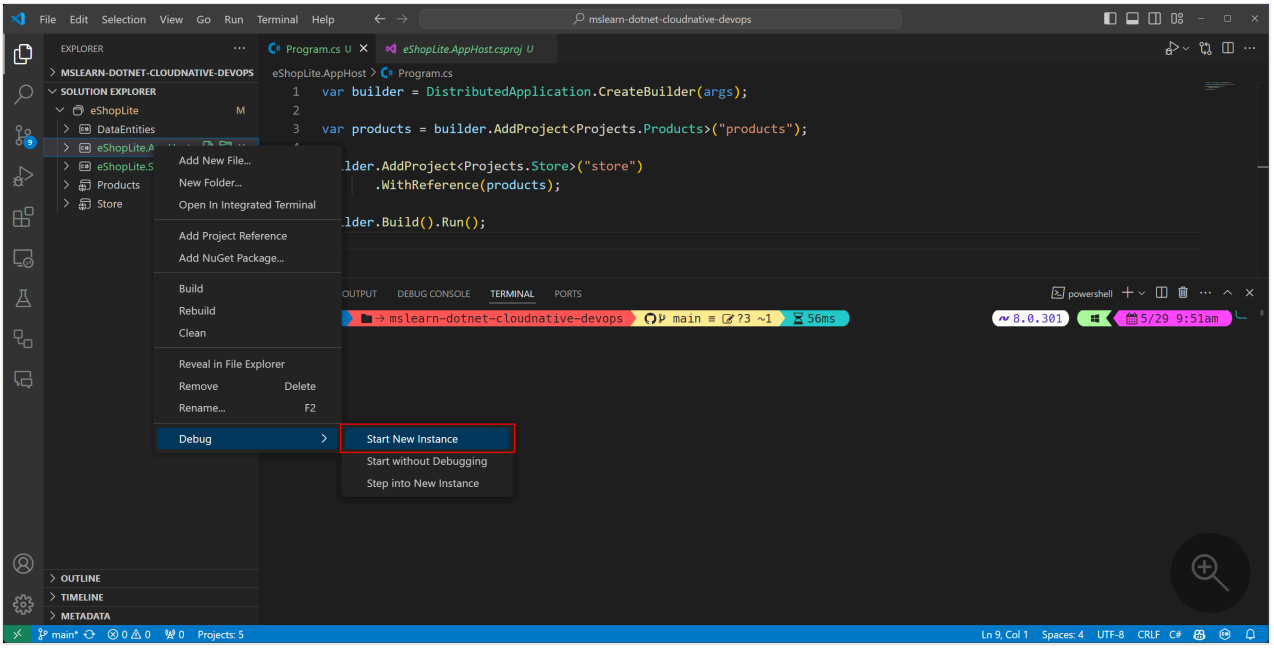
The addresses for both the endpoints now uses the "products" name that was added to the orchestrator in the *app host*. These names are used to discover the address of the *Products* project.

Explore the enrolled app

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

Delete the *launch.json* file that you created earlier, it no longer serves a purpose. Instead, start the *app host* project, which orchestrates the other projects:

1. Start the *app host* project by right-clicking the **eShopLite.AppHost** project in the **Solution Explorer** and selecting **Debug** > **Start New Instance**:



> **Note**

> If Docker Desktop (or Podman) isn't running, you experience an error. Start the container runtime and try again.

2. In the dashboard, select the endpoint for the **products** project. A new browser tab appears and displays the product catalog in JSON format.
3. In the dashboard, select the endpoint for the **store** project. A new browser tab appears and displays the home page for the web app.
4. In the menu on the left, select **Products**. The product catalog is displayed.
5. To stop debugging, close the browser.

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