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.

If you haven't already installed the .NET Aspire templates, run the following `dotnet new install` command:

.NET CLI

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

The preceding .NET CLI command ensures that you have the .NET Aspire templates available. To create the .NET Aspire Starter App from the template, run the following `dotnet new` command:

.NET CLI

```
dotnet new aspire-starter --use-redis-cache --output AspireSample
```

For more information, see dotnet new. The .NET CLI 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.

You need to trust the ASP.NET Core localhost certificate before running the app. Run the following command:

.NET CLI

```
dotnet dev-certs https --trust
```

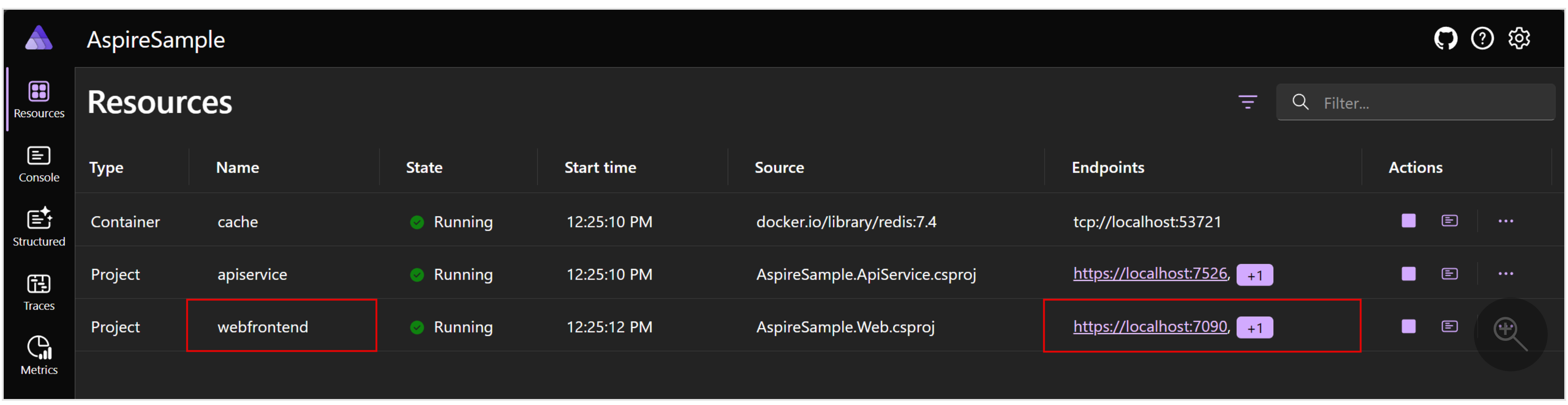
For more information, see Troubleshoot untrusted localhost certificate in .NET Aspire. For in-depth details about troubleshooting localhost certificates on Linux, see ASP.NET Core: GitHub repository issue #32842.

.NET CLI

```
dotnet run --project AspireSample/AspireSample.AppHost
```

For more information, see dotnet run.

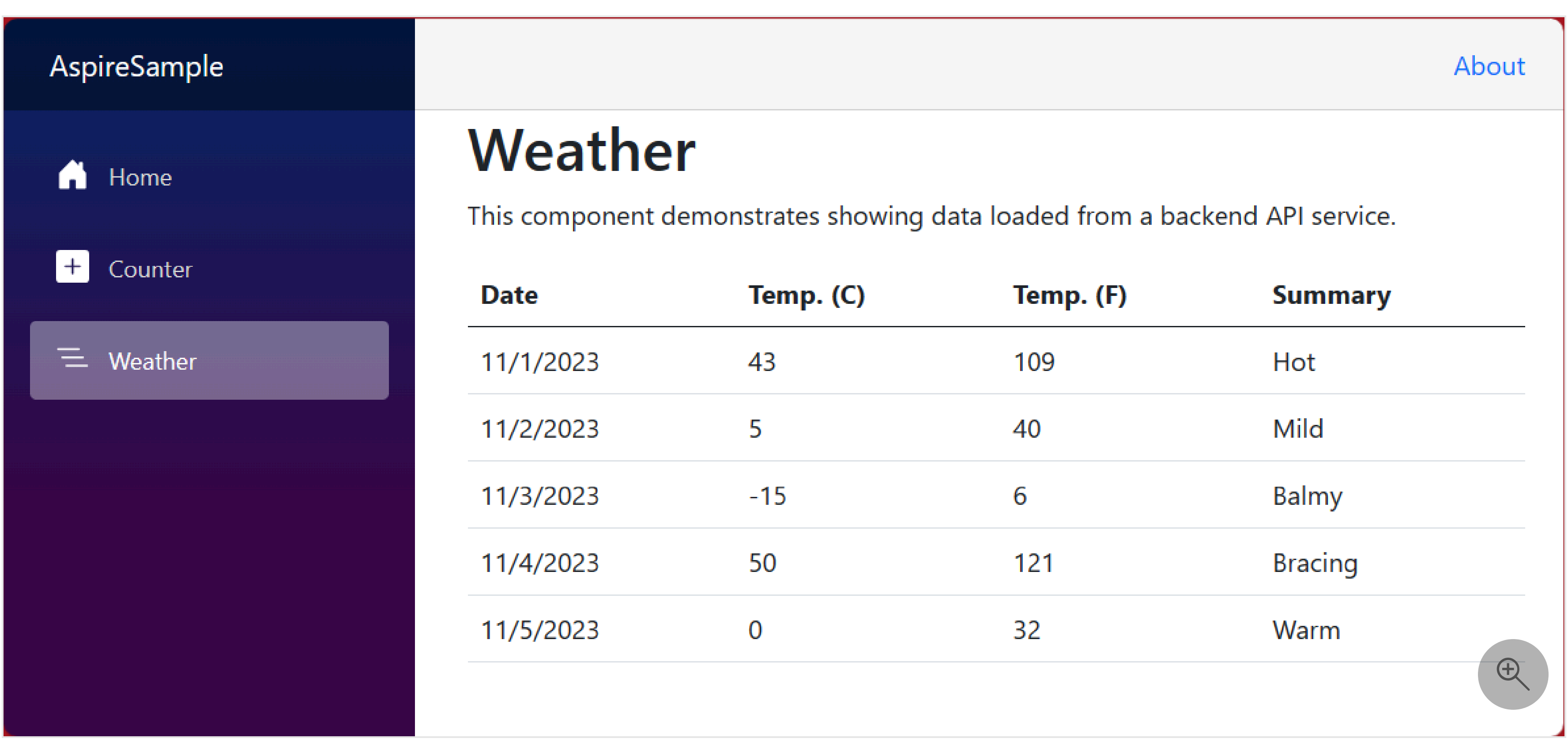
1. The app displays the .NET Aspire dashboard in the browser. You look at the dashboard in more detail later. For now, find the **webfrontend** project in the list of resources and select the project's **localhost** endpoint.

   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.



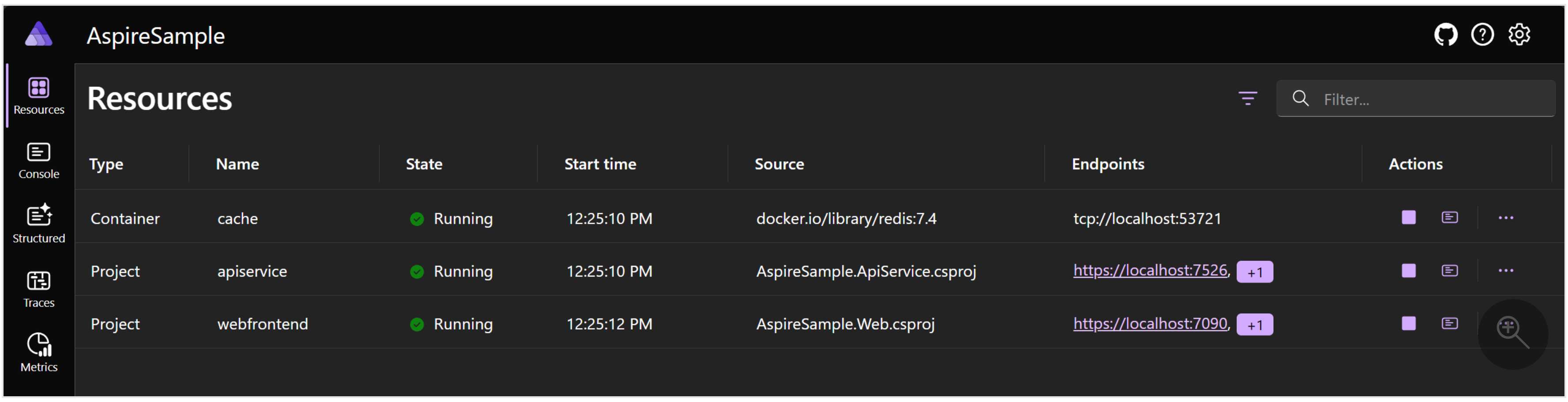
🤓 Congratulations! You created and ran your first .NET Aspire solution! To stop the app, close the browser window.

To stop the app, press `Ctrl` + `C` in the terminal window.

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

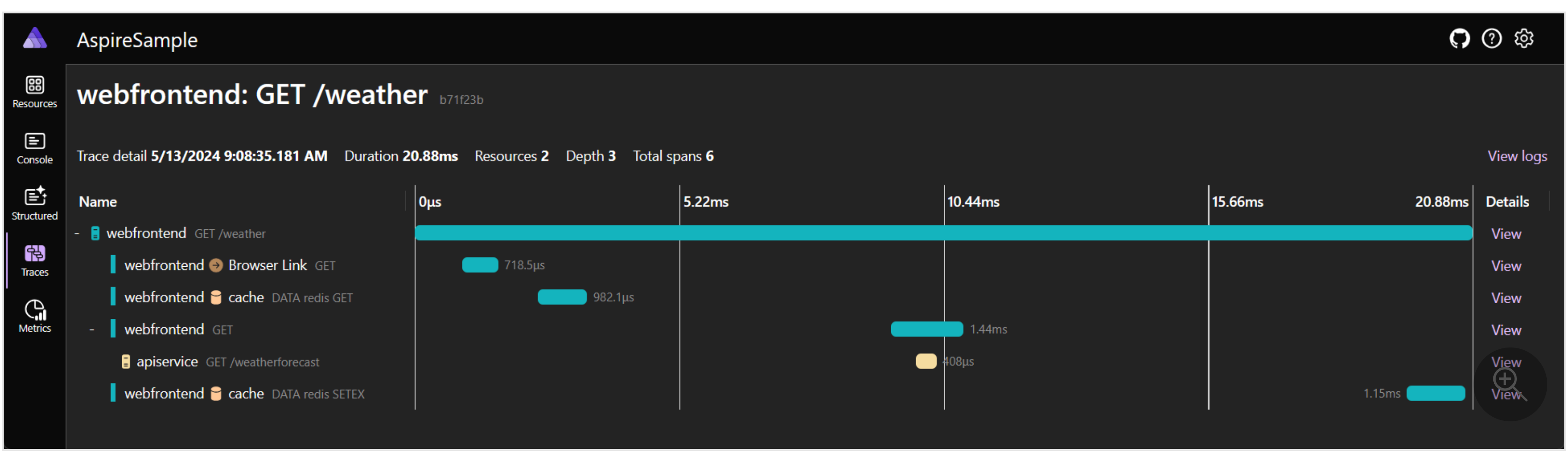
Explore the .NET Aspire dashboard

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:

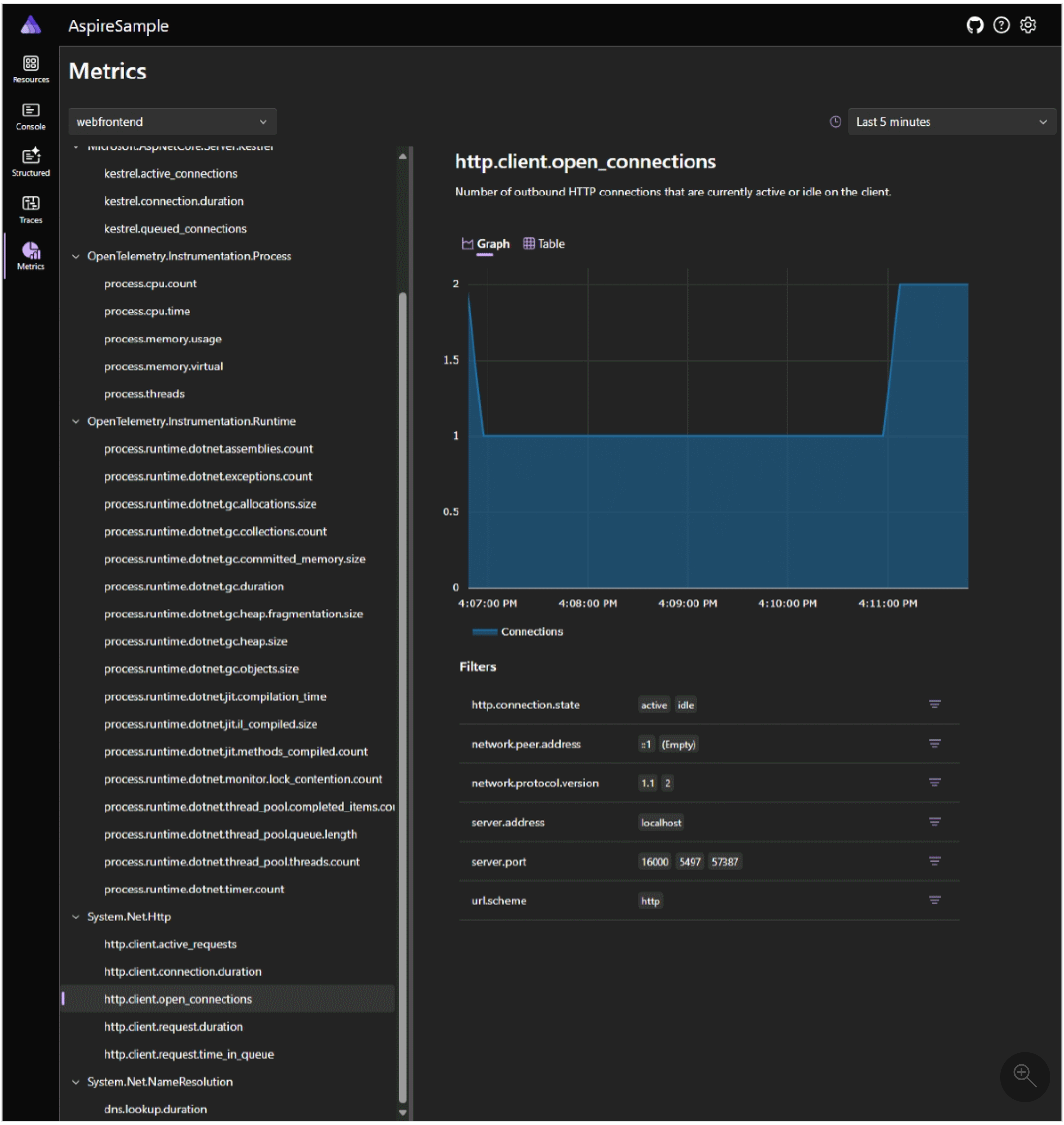


Visit each page using the left navigation to view different information about the .NET Aspire resources:

- **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="9.0.0" />

  <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="9.0.0" />
    <PackageReference Include="Aspire.Hosting.Redis" Version="9.0.0" />
  </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#

```csharp
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="9.2.0" />
    <PackageReference Include="Microsoft.Extensions.ServiceDiscovery" Version="9.0.0" />
    <PackageReference Include="OpenTelemetry.Exporter.OpenTelemetryProtocol" Version="1.11.1" />
    <PackageReference Include="OpenTelemetry.Extensions.Hosting" Version="1.11.1" />
    <PackageReference Include="OpenTelemetry.Instrumentation.AspNetCore" Version="1.11.0" />
    <PackageReference Include="OpenTelemetry.Instrumentation.Http" Version="1.11.0" />
    <PackageReference Include="OpenTelemetry.Instrumentation.Runtime" Version="1.11.0" />
  </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#

```csharp
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