

.NET Aspire documentation

Learn about .NET Aspire, an opinionated, cloud ready stack for building observable, production ready, distributed applications. Browse API reference, sample code, tutorials, quickstarts, conceptual articles and more.



OVERVIEW .NET Aspire overview



QUICKSTART Build your first .NET Aspire project



CONCEPT Distributed app host



CONCEPT .NET Aspire integrations



CONCEPT Service discovery



DEPLOY Deploy a .NET Aspire project



REFERENCE .NET Aspire FAQ



WHAT'S NEW What's new in .NET Aspire

Explore various aspects of .NET Aspire, including getting started, storage, database, messaging, caching, frameworks, deployment, and troubleshooting.

Get Started

- [.NET Aspire overview](#)
- [Build your first .NET Aspire project](#)
- [Add a Node.js app to a .NET Aspire project](#)

Storage integrations

- [Connect to storage with .NET Aspire](#)
- [Azure Blob Storage](#)
- [Azure Storage Queues](#)
- [Azure Table Storage](#)

Database integrations

- [Connect to SQL Server with EF Core](#)
- [PostgreSQL database](#)
- [PostgreSQL with EF Core](#)
- [Azure Cosmos DB](#)

- [.NET Aspire setup and tooling](#)
- [.NET Aspire service discovery](#)
- [.NET Aspire and launch profiles](#)
- [.NET Aspire service defaults](#)
- [.NET Aspire dashboard](#)
- [.NET Aspire local networking](#)

Messaging integrations

- [Implement Messaging with .NET Aspire](#)
- [Azure Event Hubs](#)
- [Azure Service Bus](#)
- [Azure Web PubSub](#)
- [RabbitMQ client](#)
- [Apache Kafka](#)
- [NATs](#)

- [Persist volume mount sample](#)

- [Azure Cosmos DB with EF Core](#)
- [SQL Database](#)
- [SQL Database with EF Core](#)
- [Entity Framework Core migrations](#)
- [MySQLConnector Database](#)
- [MongoDB Database](#)

Caching integrations

- [Improve app caching with .NET Aspire](#)
- [Stack Exchange Redis caching overview](#)
- [Redis caching](#)
- [Redis output caching](#)
- [Redis distributed caching](#)

Framework integrations

- [Use Orleans with .NET Aspire](#)
- [Orleans voting sample](#)
- [Use Dapr with .NET Aspire](#)
- [Dapr integration sample](#)

Deployment

- [Overview](#)
- [Deploy to Azure Container Apps](#)
- [Deploy using the Azure Developer CLI](#)
- [Integrate with Application Insights](#)
- [.NET Aspire deployment manifest format](#)

Troubleshooting

- [Allow unsecure transport](#)
- [Unable to install workload](#)
- [Untrusted localhost certificate](#)
- [The specified name is already in use](#)
- [Container runtime appears to be unhealthy](#)
- [The connection string is missing](#)
- [Ask questions on Discord](#)
- [Stack Overflow — .NET Aspire](#)

Training

- [Introduction to .NET Aspire](#)
- [Create a .NET Aspire project](#)
- [Use telemetry in a .NET Aspire project](#)
- [Use databases in a .NET Aspire project](#)
- [Improve performance with a cache in a .NET Aspire project](#)
- [Send messages with RabbitMQ in a .NET Aspire project](#)

.NET extensions

Learn about .NET extensions, including logging, dependency injection, configuration, and more. All of which are fundamental in .NET Aspire.

Fundamentals

- [Logging](#)
- [Dependency injection](#)
- [Configuration](#)
- [Make HTTP requests](#)

Telemetry

- [.NET observability with OpenTelemetry](#)
- [Networking telemetry](#)
- [.NET SDK telemetry](#)

Resiliency

- [Introduction to resilient app dev](#)
- [Build resilient HTTP apps](#)
- [Implement resiliency in a cloud-native ASP.NET Core microservice](#)

Observability

- [.NET app health checks in C#](#)
- [App health checks in ASP.NET Core](#)
- [Diagnostic tools in .NET](#)
- [Diagnostic resource monitoring in .NET](#)

.NET community resources

Find community resources for .NET, including webcasts, shows, open-source projects, and more.

.NET

- [.NET documentation](#)
- [.NET Aspire samples browser](#)
- [ASP.NET documentation](#)
- [Azure documentation](#)
- [C# documentation](#)
- [.NET Discord ↗](#)
- [Official .NET Aspire Collection](#)

Webcasts and shows

- [Azure Friday ↗](#)
- [The Cloud Native Show](#)
- [On .NET](#)
- [On .NET Live ↗](#)
- [.NET Community Standup ↗](#)
- [Welcome to .NET Aspire ↗](#)

Open source

- [.NET Aspire ↗](#)
- [.NET Aspire samples ↗](#)

Community

- [Follow @dotnet on X ↗](#)
- [Follow @dotnet on Mastodon ↗](#)

[.NET samples ↗](#)

[.NET Platform ↗](#)

[.NET Runtime ↗](#)

[ASP.NET Core ↗](#)

[.NET Foundation ↗](#)

[We Are .NET ↗](#)

[.NET Aspire YouTube playlist ↗](#)

[.NET Aspire on YouTube ↗](#)

API and language reference

Search the .NET API and language reference documentation.

[.NET Aspire API reference](#)

API reference documentation for .NET Aspire

[.NET Platform Extensions API reference](#)

API reference documentation for .NET Platform Extensions

[.NET API reference](#)

API reference documentation for .NET

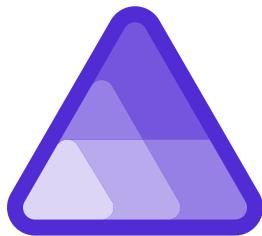
[ASP.NET Core API reference](#)

API reference documentation for ASP.NET Core

Are you interested in contributing to the .NET Aspire docs? For more information, see our [contributor guide](#).

.NET Aspire overview

Article • 09/28/2024



.NET Aspire is an opinionated, cloud ready stack for building observable, production ready, distributed applications. .NET Aspire is delivered through a collection of NuGet packages that handle specific cloud-native concerns. Cloud-native apps often consist of small, interconnected pieces or microservices rather than a single, monolithic code base. Cloud-native apps generally consume a large number of services, such as databases, messaging, and caching.

A *distributed application* is one that uses computational *resources* across multiple nodes, such as containers running on different hosts. Such nodes must communicate over network boundaries to deliver responses to users. A cloud-native app is a specific type of distributed app that takes full advantage of the scalability, resilience, and manageability of cloud infrastructures.

Why .NET Aspire?

.NET Aspire is designed to improve the experience of building .NET cloud-native apps. It provides a consistent, opinionated set of tools and patterns that help you build and run distributed apps. .NET Aspire is designed to help you with:

- **Orchestration:** .NET Aspire provides features for running and connecting multi-project applications and their 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.

Orchestration

In .NET Aspire, orchestration primarily focuses on enhancing the *local development* experience by simplifying the management of your cloud-native 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 provides 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. These abstractions ensure a consistent setup pattern across apps with numerous integrations and services, making it easier to manage complex applications 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 manages to inject the right connection strings or 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 and configures the appropriate connection string in the "frontend" project with only two 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);
```

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 many cloud-native concerns for you through standardized configuration patterns, such as adding health checks and telemetry. Integrations are two-fold, in that one side represents the service you're connecting to, and the other side represents the client or consumer of that service. In other words, for each hosting package there's a corresponding client package that handles the service connection.

Each integration is designed to work with .NET Aspire orchestration, 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. The templates are opinionated and come with a set of defaults that help you get started quickly. They include boilerplate code and configurations that are common to cloud-native apps, such as telemetry, health checks, and service discovery. For more information, see [.NET Aspire project templates](#).

.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 • 09/28/2024

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](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or the [.NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

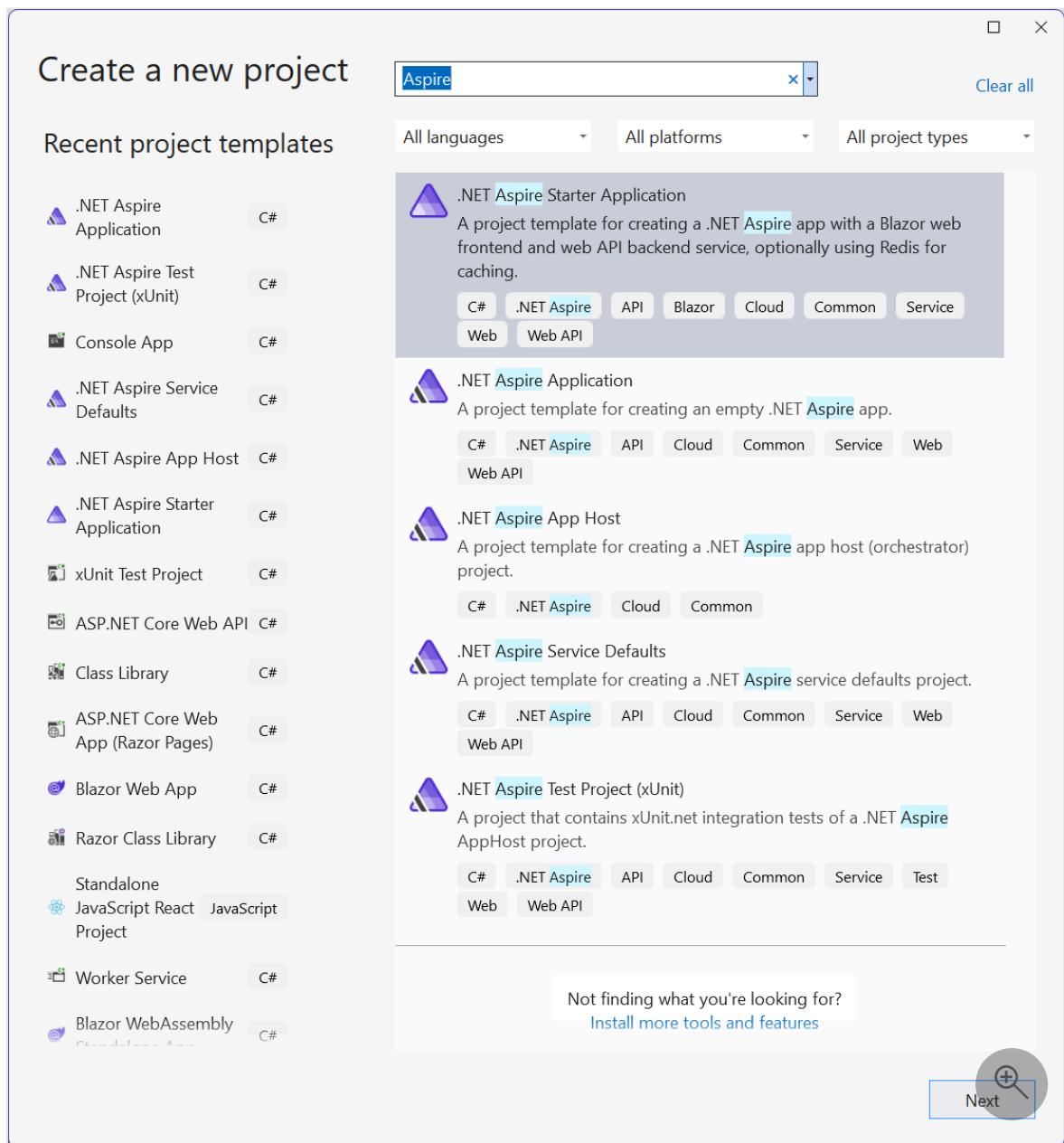
For more information, see [.NET Aspire setup and tooling](#).

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 project 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 Application**. 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 8.0 (Long 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 [Testing .NET Aspire projects](#).

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

Test the app locally

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**.

Then, press **F5** 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.

Type	Name	State	Start time	Source	Endpoints	Logs	Details
Container	cache	Running	9:07:09 AM	docker.io/library/redis:7.2	tcp://localhost:60665	View	View
Project	apiservice	Running	9:07:09 AM	AspireSample.ApiService.csproj	https://localhost:7352/weatherforecast	View	View
Project	webfrontend	Running	9:07:09 AM	AspireSample.Web.csproj	https://localhost:7163	View	View

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.

The screenshot shows a mobile-style interface with a dark purple header bar containing the title "AspireSample". Below the header is a navigation bar with three items: "Home" (with a house icon), "Counter" (with a plus icon), and "Weather" (with a three-line menu icon). The main content area has a light gray background and features a large title "Weather". Below the title is a subtitle: "This component demonstrates showing data loaded from a backend API service." To the right of the subtitle is a table with five rows of weather data. The table has four columns: "Date", "Temp. (C)", "Temp. (F)", and "Summary". The data is as follows:

Date	Temp. (C)	Temp. (F)	Summary
11/1/2023	43	109	Hot
11/2/2023	5	40	Mild
11/3/2023	-15	6	Balmy
11/4/2023	50	121	Bracing
11/5/2023	0	32	Warm

On the far right of the table is a circular button with a magnifying glass icon.

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:

The screenshot shows the .NET Aspire dashboard with the "Resources" tab selected. On the left is a vertical navigation bar with icons for "Console", "Structured", "Traces", and "Metrics". The main content area has a table titled "Resources". The table has columns: "Type", "Name", "State", "Start time", "Source", "Endpoints", "Logs", and "Details". There are three entries in the table:

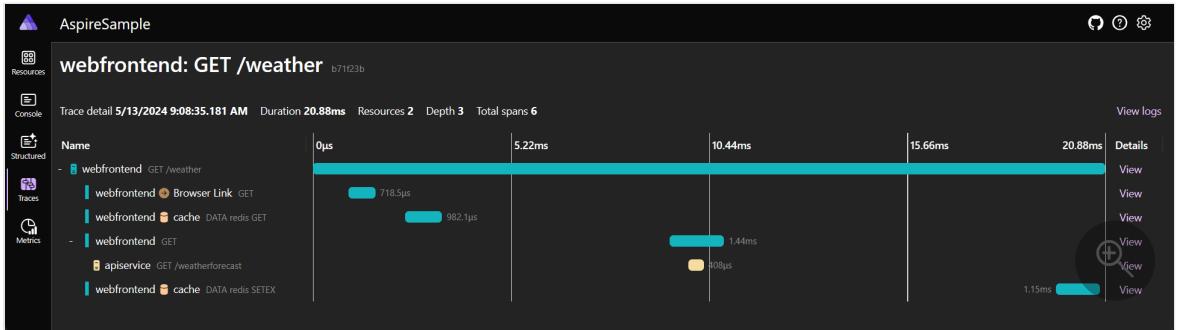
Type	Name	State	Start time	Source	Endpoints	Logs	Details
Container	cache	Running	9:07:09 AM	docker.io/library/redis:7.2	tcp://localhost:60665	View	View
Project	apiservice	Running	9:07:09 AM	AspireSample.ApiService.csproj	https://localhost:7352/weatherforecast	View	View
Project	webfrontend	Running	9:07:09 AM	AspireSample.Web.csproj	https://localhost:7163	View	View

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

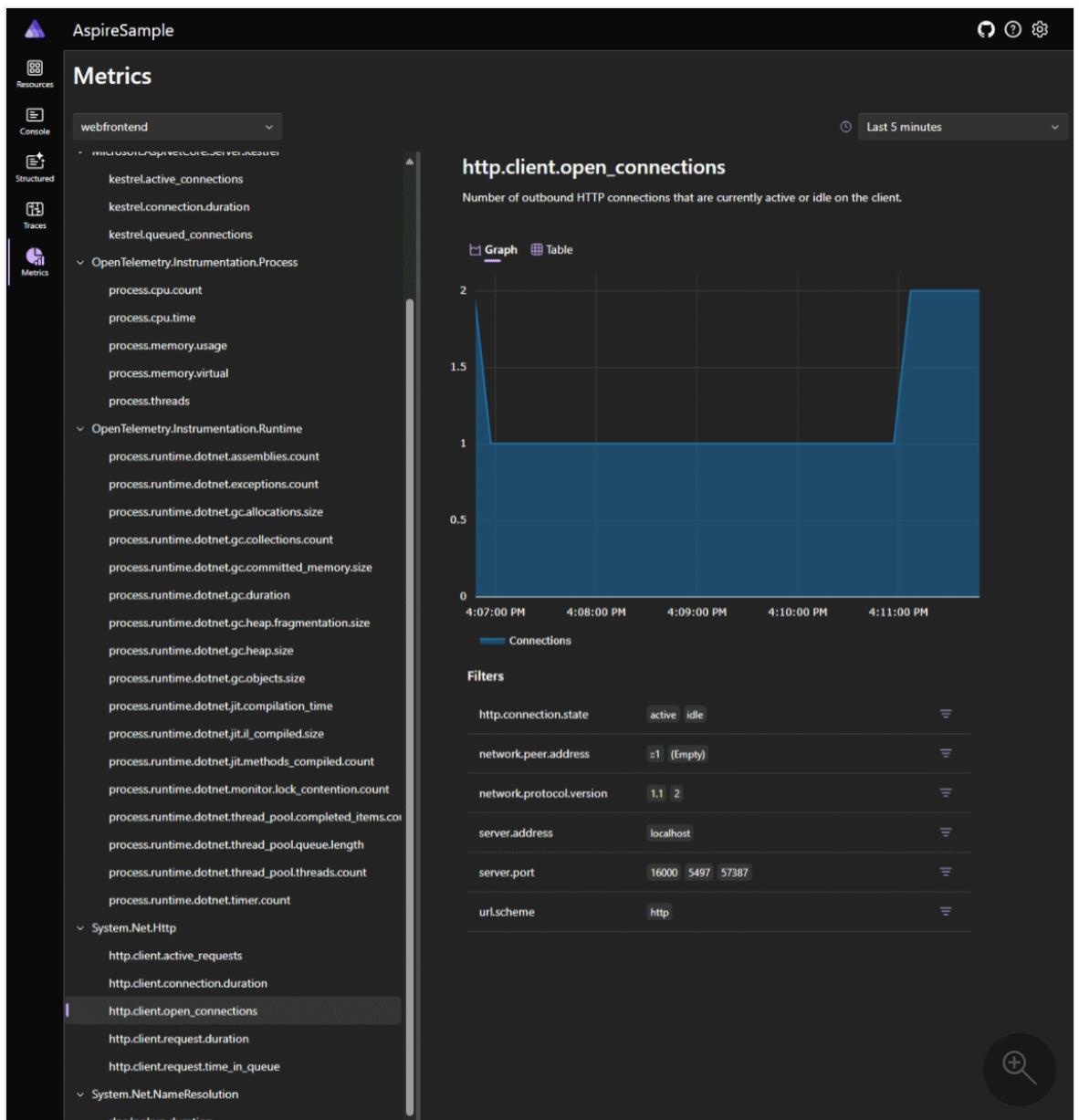
- **Resources:** Lists basic information for all of the individual .NET projects in your .NET Aspire project, such as the app state, endpoint addresses, and the environment variables that were loaded in.
- **Console:** Displays the console output from each of the projects in your app.
- **Structured:** 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](#).

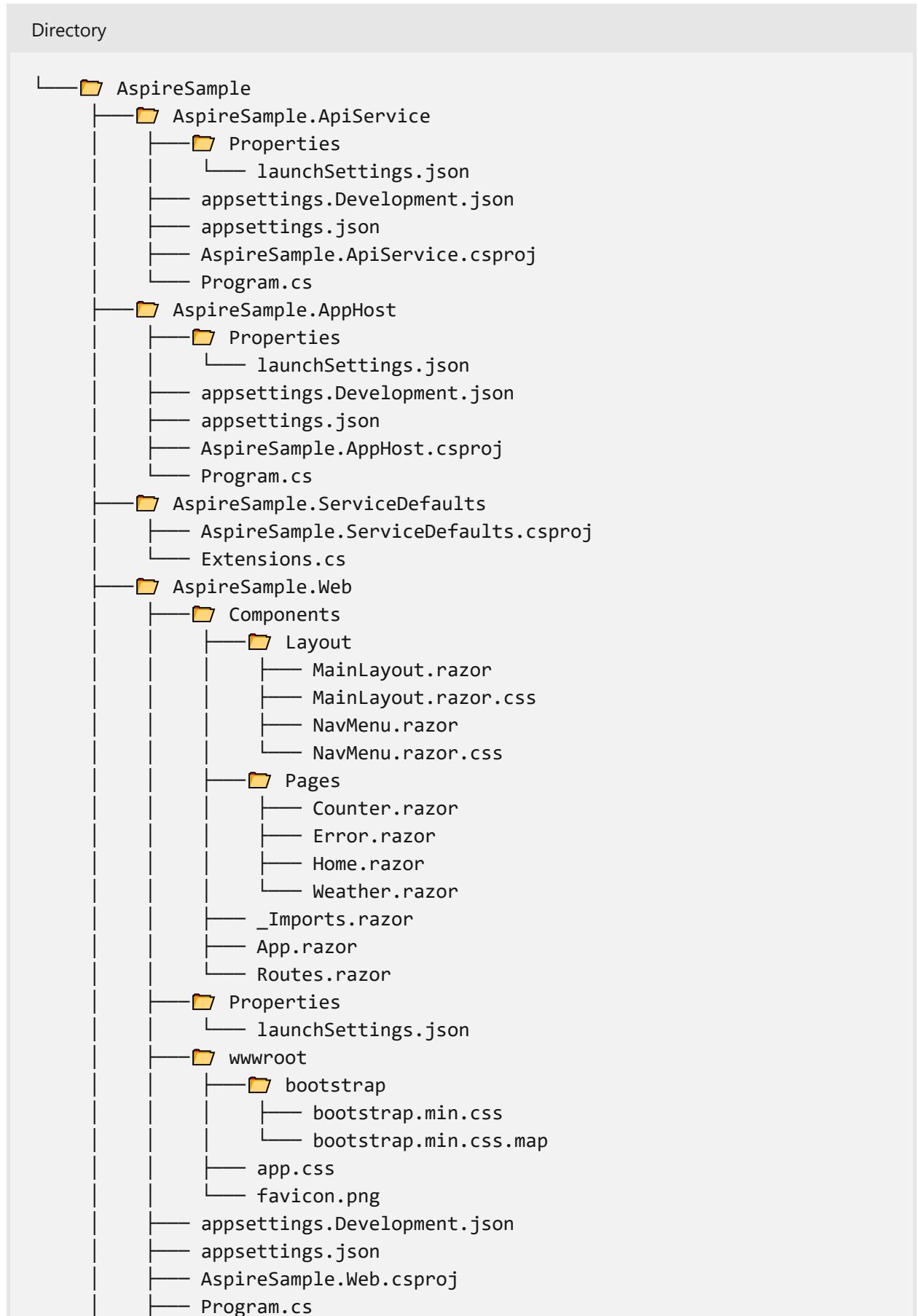
Understand the .NET Aspire solution structure

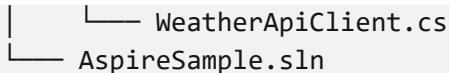
The solution consists of the following projects:

- **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:





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 [New Blazor Web App template](#). 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

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <IsAspireHost>true</IsAspireHost>
    <UserSecretsId>1b1a6997-c2dd-47d0-b3fa-46811f182483</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="8.2.0" />
    <PackageReference Include="Aspire.Hosting.Redis" Version="8.2.0" />
  </ItemGroup>

</Project>
```

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)
    .WithReference(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.

Finally, the app is built and run. The `DistributedApplication.Run()` method is provided by the .NET Aspire SDK, and 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

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

<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
  <ImplicitUsings>enable</ImplicitUsings>
  <Nullable>enable</Nullable>
  <IsAspireSharedProject>true</IsAspireSharedProject>
</PropertyGroup>

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

    <PackageReference Include="Microsoft.Extensions.Http.Resilience"
Version="8.9.1" />
    <PackageReference Include="Microsoft.Extensions.ServiceDiscovery"
Version="8.2.0" />
    <PackageReference Include="OpenTelemetry.Exporter.OpenTelemetryProtocol"
Version="1.9.0" />
    <PackageReference Include="OpenTelemetry.Extensions.Hosting"
Version="1.9.0" />
    <PackageReference Include="OpenTelemetry.Instrumentation.AspNetCore"
Version="1.9.0" />
    <PackageReference Include="OpenTelemetry.Instrumentation.Http"
Version="1.9.0" />
    <PackageReference Include="OpenTelemetry.Instrumentation.Runtime"
Version="1.9.0" />
  </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#

```
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 components.  
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.UseStaticFiles();  
app.UseAntiforgery();  
  
app.UseOutputCache();  
  
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 • 06/03/2024

If you have already created a microservices .NET web app, you can add .NET Aspire to it and get all the included features and benefits. In this article, you'll add .NET Aspire orchestration to a simple, pre-existing .NET 8 project. You'll 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](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or [the .NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

Get started

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

1. Open a command prompt and change to a directory where you want to store the code.
2. To clone to .NET 8 example solution, use this command:

Bash

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

Explore the sample app

This article uses a .NET 8 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.

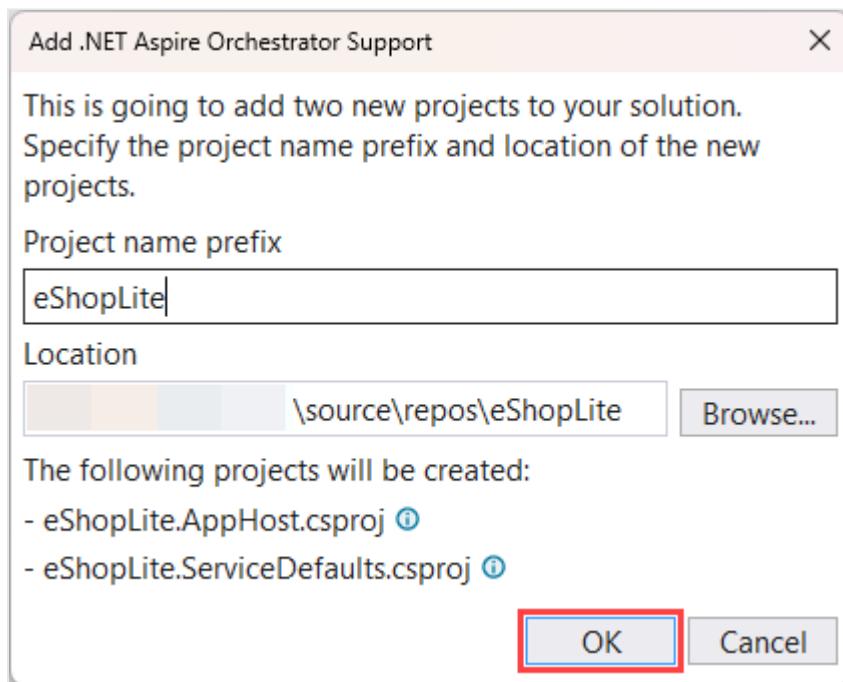
Open and start debugging the project to examine its default behavior:

1. Start Visual Studio and then select **File > Open > Project/Solution**.
2. Navigate to the top level folder of the solution you cloned, select `eShopLite.sln`, and then select **Open**.
3. In the **Solution Explorer**, right-click the `eShopLite` solution, and then select **Configure Startup Projects**.
4. Select **Multiple startup projects**.
5. In the **Action** column, select **Start** for both the **Products** and **Store** projects.
6. Select **OK**.
7. To start debugging the solution, press `F5` or select **Start**.
8. 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.
9. To stop debugging, close the browser.

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:

1. In Visual Studio, in the **Solution Explorer**, right-click the **Store** project, select **Add**, and then select **.NET Aspire Orchestrator Support**.
2. In the **Add .NET Aspire Orchestrator Support** dialog, select **OK**.



You should now have two new projects, both added to the solution:

- **eShopLite.AppHost:** An orchestrator project designed to connect and configure the different projects and services of your app. The orchestrator is set as the *Startup project*, and it depends on the **eShopLite.Store** project.
- **eShopLite.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](#).

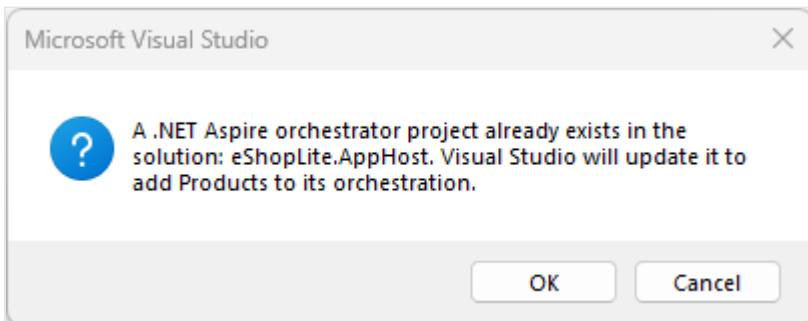
In the **eShopLite.AppHost** project, open the *Program.cs* file. Notice this line of code, which registers the **Store** project in the .NET Aspire orchestration:

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

For more information, see [AddProject](#).

To add the **Products** project to .NET Aspire:

1. In Visual Studio, in the **Solution Explorer**, right-click the **Products** project, select **Add**, and then select **.NET Aspire Orchestrator Support**.
2. A dialog indicating that .NET Aspire Orchestrator project already exists, select **OK**.



In the **eShopLite.AppHost** project, open the *Program.cs* file. Notice this line of code, which registers the **Products** project in the .NET Aspire orchestration:

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

Also notice that the **eShopLite.AppHost** project, now depends on both the **Store** and **Products** projects.

Service Discovery

At this point, both projects are part of .NET Aspire orchestration, but the *Store* needs to be able to discover the **Products** backend address through [.NET Aspire's service discovery](#). To enable service discovery, open the *Program.cs* file in **eShopLite.AppHost** and update the code that the *Store* 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);  
  
builder.Build().Run();
```

You've added a reference to the *Products* project in the *Store* project. This reference is used to discover the address of the *Products* project. 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.

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 has added.

ⓘ Note

Notice that the `eShopLite.AppHost` project is the new startup project.

1. In Visual Studio, to start debugging, press `F5` Visual Studio builds the projects.
2. If the **Start Docker Desktop** dialog appears, select **Yes**. Visual Studio starts the Docker engine and creates the necessary containers. When the deployment is complete, the .NET Aspire dashboard is displayed.
3. In the dashboard, select the endpoint for the `products` project. A new browser tab appears and displays the product catalog in JSON format.
4. In the dashboard, select the endpoint for the `store` project. A new browser tab appears and displays the home page for the web app.
5. In the menu on the left, select **Products**. The product catalog is displayed.
6. Close the browser to stop debugging.

Congratulations! You have added .NET Aspire orchestration to your pre-existing web app. You can now add .NET Aspire integrations and use the .NET Aspire tooling to streamline your cloud-native web app development.

.NET Aspire setup and tooling

Article • 09/24/2024

.NET Aspire includes tooling to help you create and configure cloud-native apps. The tooling includes useful starter project templates and other features to streamline getting started with .NET Aspire for Visual Studio, Visual Studio Code, and CLI workflows. In the sections ahead, you'll learn how to work with .NET Aspire tooling and explore the following tasks:

- ✓ Install .NET Aspire and its dependencies
- ✓ Create starter project templates using Visual Studio, Visual Studio Code, or the .NET CLI
- ✓ Install .NET Aspire integrations
- ✓ Work with the .NET Aspire dashboard

Install .NET Aspire

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

- [.NET 8.0](#).
- .NET Aspire workload (installed either Visual Studio or the .NET CLI).
- 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 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

The .NET Aspire workload installs internal dependencies and makes available other tooling, such as project templates and Visual Studio features. There are two ways to install the .NET Aspire workload. If you prefer to use Visual Studio Code, follow the .NET CLI instructions:

Visual Studio 2022 17.10 or higher includes the latest .NET Aspire workload by default. To verify that you have the .NET Aspire workload installed, run the following command:

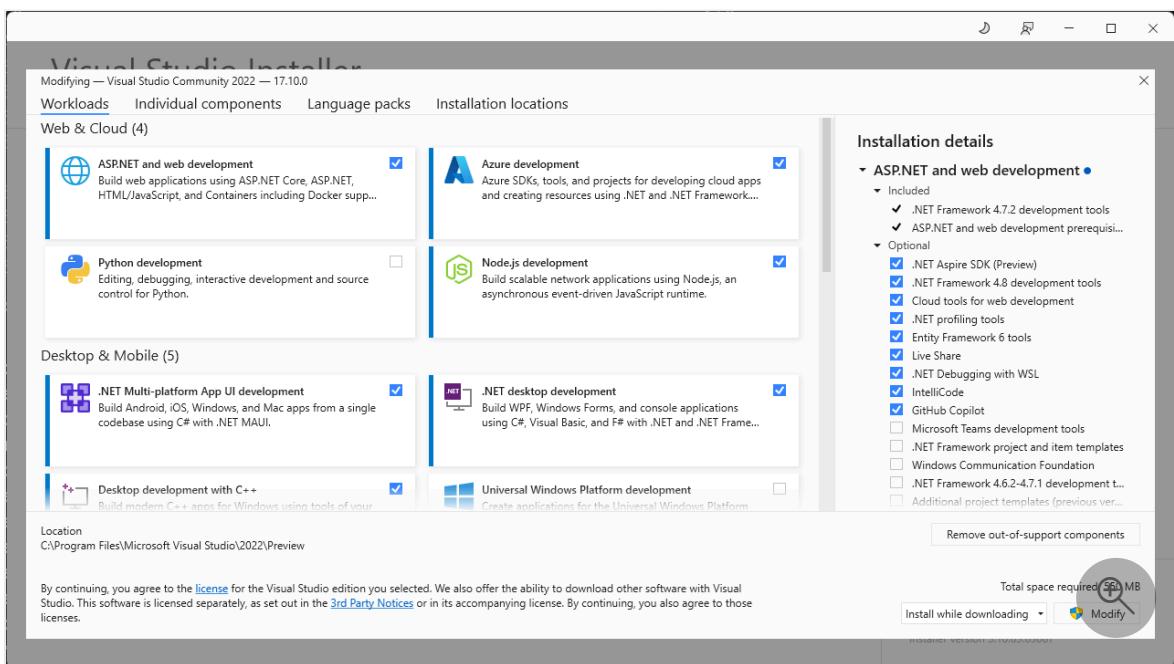
.NET CLI

```
dotnet workload list
```

If you have an earlier version of Visual Studio 2022, you can either upgrade to Visual Studio 2022 17.10 or you can install the .NET Aspire workload using the following steps:

To install the .NET Aspire workload in Visual Studio 2022, use the Visual Studio installer.

1. Open the Visual Studio Installer.
2. Select **Modify** next to Visual Studio 2022.
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 integration.



6. To ensure that you install the latest version of the .NET Aspire workload, run the following `dotnet workload update` command before you install .NET Aspire:

```
.NET CLI  
  
dotnet workload update
```

7. To install the .NET Aspire workload from the .NET CLI, use the `dotnet workload install` command:

```
.NET CLI  
  
dotnet workload install aspire
```

8. To check your version of .NET Aspire, run this command:

```
dotnet workload list
```

Container runtime

.NET Aspire projects are designed to run in containers. You can use either Docker Desktop or Podman as your container runtime. [Docker Desktop](#) is the most common container runtime. [Podman](#) is an open-source daemonless alternative to Docker, that can build and run Open Container Initiative (OCI) containers. If your host environment has both Docker and Podman installed, .NET Aspire defaults to using Docker. You can instruct .NET Aspire to use Podman instead, by setting the

`DOTNET_ASPIRE_CONTAINER_RUNTIME` environment variable to `podman`:

Windows

PowerShell

```
$env:DOTNET_ASPIRE_CONTAINER_RUNTIME = "podman"
```

For more information, see [Install Podman on Windows](#).

.NET Aspire project templates

The .NET Aspire workload makes available .NET Aspire project templates. These project templates allow you to create new apps pre-configured with the .NET Aspire project structure and default settings. These projects also provide a unified debugging experience across the different resources of your app.

There are currently four project templates available:

- **.NET Aspire Empty App:** A minimal .NET Aspire project that includes the following:
 - **AspireSample.AppHost:** An orchestrator project designed to connect and configure the different projects and services of your app.
 - **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](#).

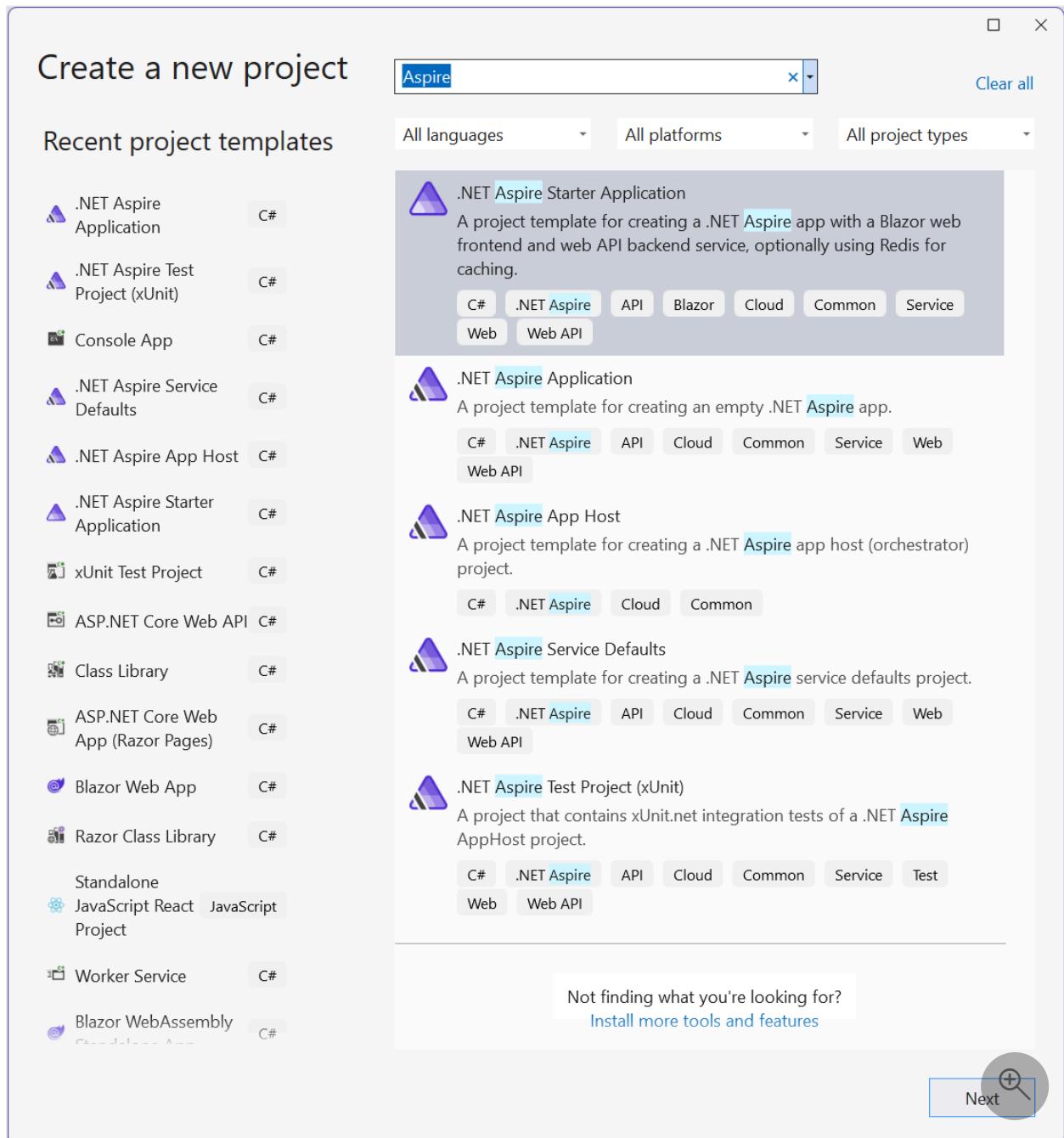
- **.NET Aspire Starter Application:** In addition to the `.AppHost` and `.ServiceDefaults` projects, the .NET Aspire Starter Application also includes the following—assuming the solution is named `AspireSample`:
 - **AspireSample.ApiService:** An ASP.NET Core Minimal API project is used to provide data to the frontend. This project depends on the shared `AspireSample.ServiceDefaults` project.
 - **AspireSample.Web:** An ASP.NET Core Blazor App project with default .NET Aspire service configurations, this project depends on the `AspireSample.ServiceDefaults` project.
 - **AspireSample.Test:** Either an MSTest, NUnit, or xUnit test project with project references to the `AspireSample.AppHost` and an example `WebTests.cs` file demonstrating an integration test.
- **.NET Aspire App Host:** A standalone `.AppHost` project that can be used to orchestrate and manage the different projects and services of your app.
- **.NET Aspire Test projects:** These project templates are used to create test projects for your .NET Aspire app, and they're intended to represent functional and integration tests. The test projects include the following templates:
 - **MSTest:** A project that contains MSTest integration of a .NET Aspire AppHost project.
 - **NUnit:** A project that contains NUnit integration of a .NET Aspire AppHost project.
 - **xUnit:** A project that contains xUnit.net integration of a .NET Aspire AppHost project.
- **.NET Aspire Service Defaults:** A standalone `.ServiceDefaults` project that can be used to manage configurations that are reused across the projects in your solution related to [resilience](#), [service discovery](#), and [telemetry](#).

Important

The service defaults project template takes a `FrameworkReference` dependency on `Microsoft.AspNetCore.App`. This may not be ideal for some project types. For more information, see [.NET Aspire service defaults](#).

[samples](#) repository.

To create a .NET Aspire project using Visual Studio, search for *Aspire* in the Visual Studio new project window and select your desired template.



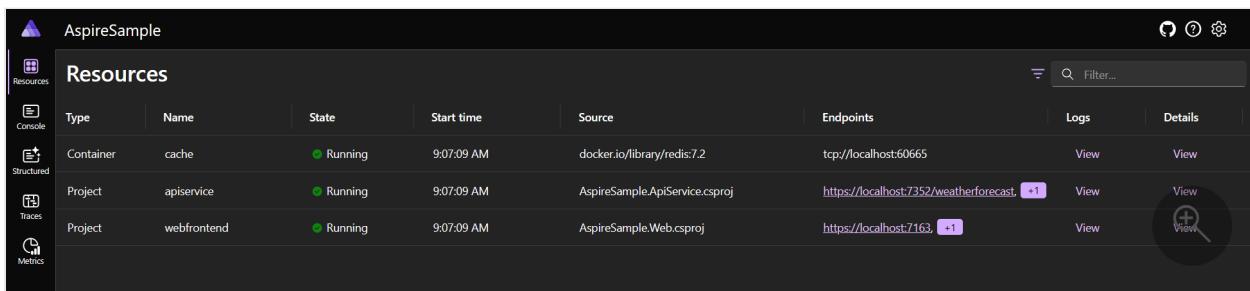
.NET Aspire dashboard

.NET Aspire templates that expose the app host project also include a useful [dashboard](#) that can be used to monitor and inspect various aspects of your app, such as logs, traces, and environment configurations. This dashboard is designed to improve the local development experience and provides an overview of the overall state and structure of your app.

The .NET Aspire dashboard is only visible while the app is running and starts automatically when you start the **.AppHost** project. Visual Studio launches both your app and the .NET Aspire dashboard for you automatically in your browser. If you start the app using the .NET CLI, copy and paste the dashboard URL from the output into your browser, or hold **ctrl** and select the link (if your terminal supports hyperlinks).

```
info: Aspire.Hosting.DistributedApplication[0]
Aspire version: 8.0.0+d215c528c07c7919c3ac30b35d92f4e51a60523b
info: Aspire.Hosting.DistributedApplication[0]
Distributed application starting.
info: Aspire.Hosting.DistributedApplication[0]
Application host directory is: D:\source\repos\docs-aspire\docs\get-started\snippets\quickstart\AspireSample\AspireSample.AppHost
info: Aspire.Hosting.DistributedApplication[0]
Now listening on: https://localhost:17187
info: Aspire.Hosting.DistributedApplication[0]
Login to the dashboard at https://localhost:17187/login?t=fd127643c01bd8d1afe61c7e1dbb340f
info: Aspire.Hosting.DistributedApplication[0]
Distributed application started. Press Ctrl+C to shut down.
```

The left navigation provides links to the different parts of the dashboard, each of which you'll explore in the following sections.



The screenshot shows the Aspire Sample dashboard with the 'Resources' tab selected. On the left, there's a sidebar with icons for 'Console', 'Structured', 'Traces', and 'Metrics'. The main area has a table with columns: Type, Name, State, Start time, Source, Endpoints, Logs, and Details. There are three entries:

Type	Name	State	Start time	Source	Endpoints	Logs	Details
Container	cache	Running	9:07:09 AM	docker.io/library/redis:7.2	tcp://localhost:60665	View	View
Project	apiservice	Running	9:07:09 AM	AspireSample.ApiService.csproj	https://localhost:7352/weatherforecast, +1	View	View
Project	webfrontend	Running	9:07:09 AM	AspireSample.Web.csproj	https://localhost:7163, +1	View	View

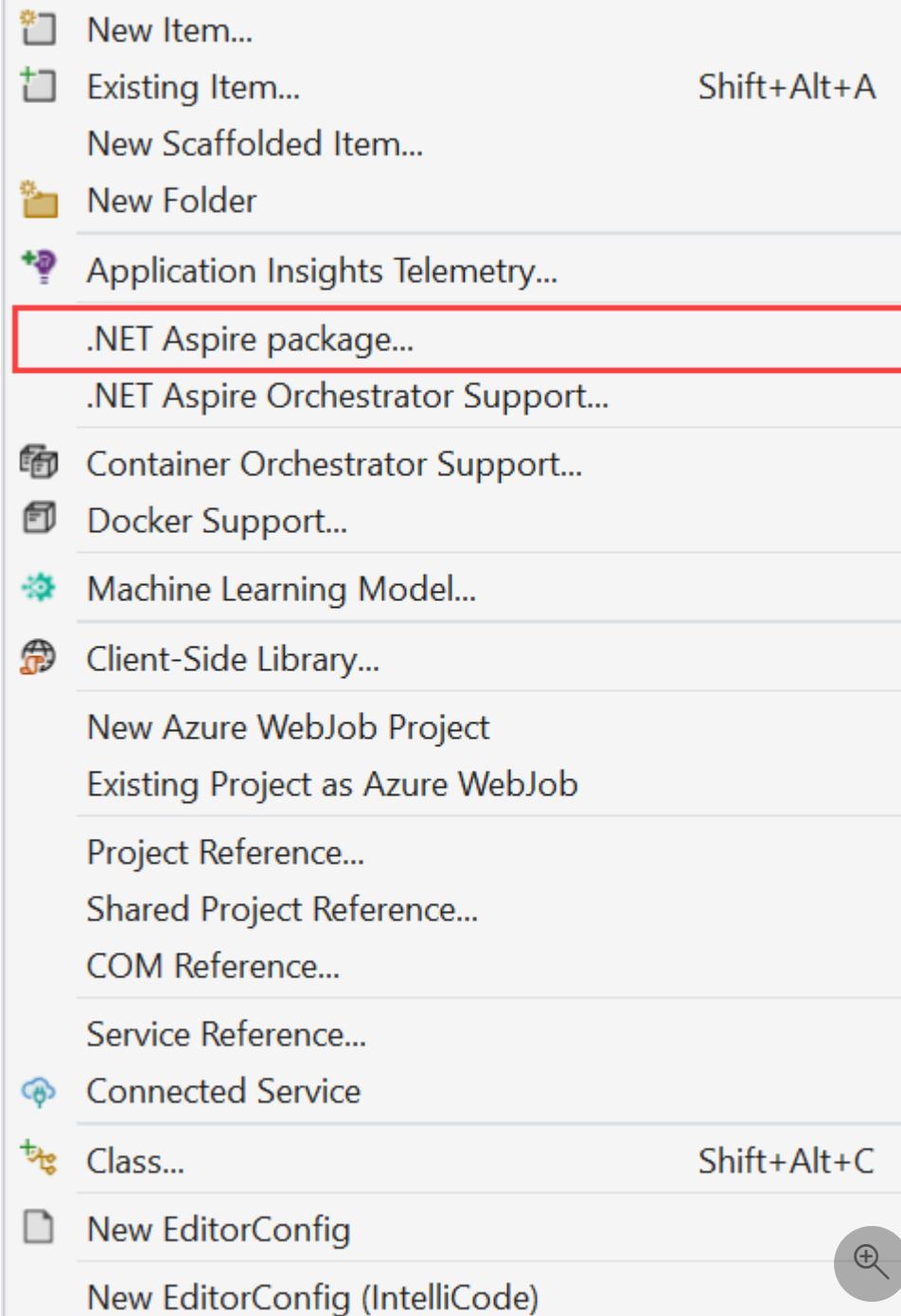
Visual Studio tooling

Visual Studio provides additional features for working with .NET Aspire integrations and the App Host orchestrator project. Not all of these features are currently available in Visual Studio Code or through the CLI.

Add a integration package

You add .NET Aspire integrations to your app like any other NuGet package using Visual Studio. However, Visual Studio also provides UI options to add .NET Aspire integrations directly.

1. In Visual Studio, right click on the project you want to add an .NET Aspire integration to and select **Add > .NET Aspire package....**



2. The package manager will open with search results pre-configured (populating filter criteria) for .NET Aspire integrations, allowing you to easily browse and select the desired integration.

NuGet Package Manager: AspireSample.Web

owner:Aspire tags:component Include prerelease

Package source: All

 Aspire.StackExchange.Redis by Microsoft, 65.3K downloads	8.0.0
A generic Redis® client that integrates with Aspire, including health checks, logging, and telemetry.	
 Aspire.RabbitMQ.Client by Microsoft, 34K downloads	8.0.0
A RabbitMQ client that integrates with Aspire, including health checks, logging, and telemetry.	
 Aspire.Npgsql.EntityFrameworkCore.PostgreSQL by Microsoft, 56.1K downloads	8.0.0
A PostgreSQL® provider for Entity Framework Core that integrates with Aspire, including connectio...	
 Aspire.Npgsql by Microsoft, 36.1K downloads	8.0.0
A PostgreSQL® client that integrates with Aspire, including health checks, metrics, logging, and tele...	
 Aspire.StackExchange.Redis.OutputCaching by Microsoft, 26.2K downloads	8.0.0
A Redis® implementation for ASP.NET Core Output Caching that integrates with Aspire, including h...	
 Aspire.Azure.Storage.Blobs by Microsoft, 16.5K downloads	8.0.0
A client for Azure Blob Storage that integrates with Aspire, including health checks, logging and tele...	
 Aspire.Azure.AI.OpenAI by Microsoft, 11.6K downloads	8.0.0
A client for Azure OpenAI that integrates with Aspire, including logging and tele...	
 Aspire.StackExchange.Redis.DistributedCaching by Microsoft, 16.1K downloads	8.0.0
A Redis® implementation for IDistributedCache that integrates with Aspire, including health checks....	
 Aspire.Microsoft.EntityFrameworkCore.SqlServer by Microsoft, 22.6K downloads	8.0.0
A Microsoft SQL Server provider for Entity Framework Core that integrates with Aspire, including co...	
 Aspire.Azure.Security.KeyVault by Microsoft, 7.5K downloads	8.0.0
Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.	
<input type="checkbox"/> Don't show this again	



Aspire.StackExchange.Redis

Version: Latest stable 8.0.0

Package source mapping is off. [Configure](#)

Options

Description
A generic Redis® client that integrates with Aspire, including health checks, logging, and telemetry.

Version: 8.0.0
Author(s): Microsoft
License: MIT
Date published: Wednesday, May 8, 2024 (5/8/2024)
Project URL: <https://github.com/dotnet/aspire>
Report Abuse: <https://www.nuget.org/packages/Aspire.StackExchange.Redis/8.0.0/ReportAbuse>

Tags: aspire, component, cloud, cache, caching, redis

Dependencies

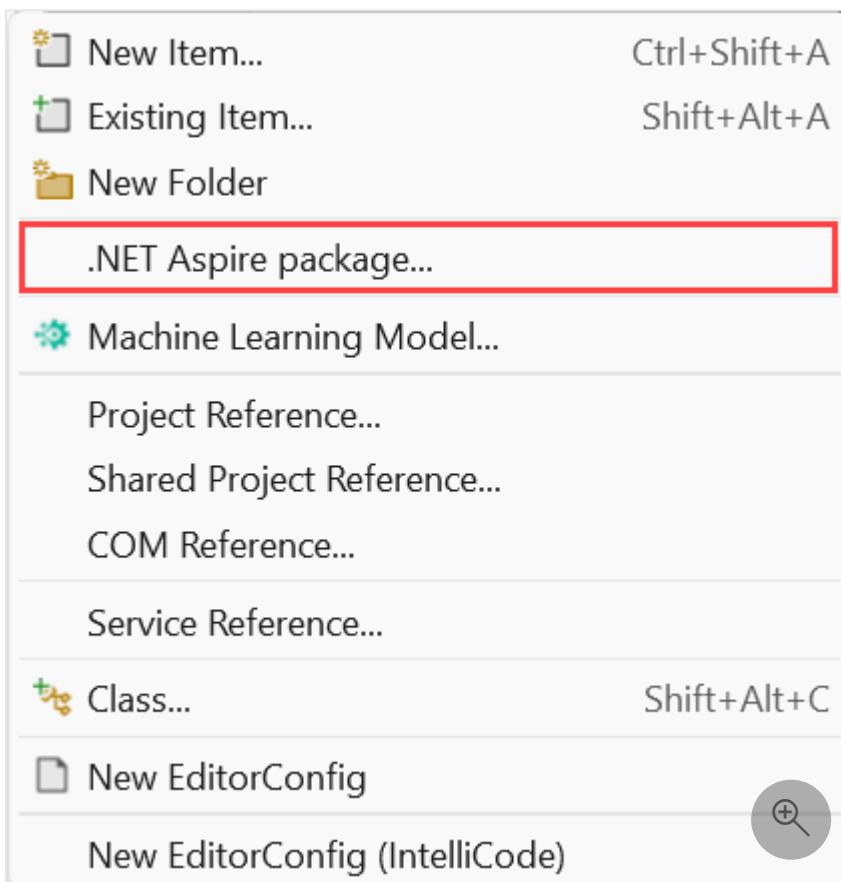
- net8.0
 - AspNetCore.HealthChecks.Redis (>= 8.0.1)
 - Microsoft.Extensions.Configuration.Binder (>= 8.0.1)
 - Microsoft.Extensions.Diagnostics.HealthChecks (>= 8.0.1)
 - Microsoft.Extensions.Hosting.Abstractions (>= 8.0.0)
 - OpenTelemetry.Extensions.Hosting (>= 1.8.1)
 - StackExchange.Redis (>= 2.7.33)

For more information on .NET Aspire integrations, see [.NET Aspire integrations overview](#).

Add hosting packages

.NET Aspire hosting packages are used to configure various resources and dependencies an app may depend on or consume. Hosting packages are differentiated from other integration packages in that they are added to the **.AppHost** project. To add a hosting package to your app, follow these steps:

1. In Visual Studio, right click on the **.AppHost** project and select **Add > .NET Aspire package....**



2. The package manager will open with search results pre-configured (populating filter criteria) for .NET Aspire hosting packages, allowing you to easily browse and select the desired package.

The screenshot shows the NuGet Package Manager interface with the following details:

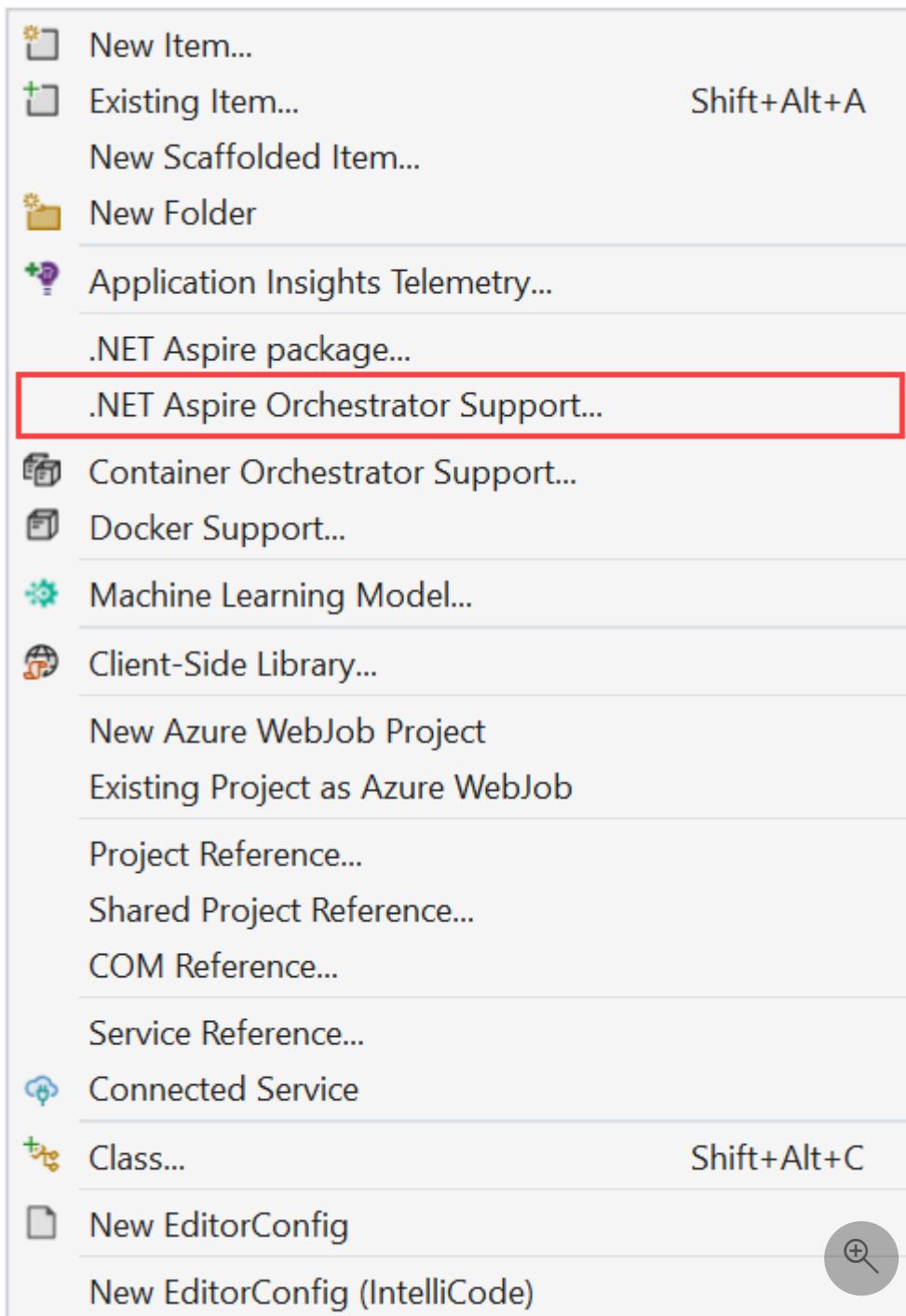
- Browse**, **Installed**, **Updates** tabs are visible at the top.
- Search bar: `owner:Aspire tags:hosting`
- Filter: Include prerelease
- Package source: All
- Results table:

Package	Description	Version
Aspire.Hosting by Microsoft, 163K downloads	Core abstractions for the .NET Aspire application model.	8.0.0
Aspire.Hosting.Sdk by Microsoft, 86.3K downloads	.NET Aspire Hosting SDK. Enabled via <IsAspireHost>true</IsAspireHost>.	8.0.0
Aspire.Hosting.AppHost by Microsoft, 35.5K downloads	Core library and MSBuild logic for .NET Aspire AppHost projects.	8.0.0
Aspire.Hosting.Azure by Microsoft, 31.1K downloads	Azure resource types for .NET Aspire.	8.0.0
Aspire.Hosting.Redis by Microsoft, 14K downloads	Redis® support for .NET Aspire.	8.0.0
Aspire.Hosting.Dapr by Microsoft, 11.1K downloads	Dapr support for .NET Aspire.	8.0.0
Aspire.Hosting.RabbitMQ by Microsoft, 8.48K downloads	RabbitMQ support for .NET Aspire.	8.0.0
Aspire.Hosting.NodeJs by Microsoft, 7.09K downloads	Node.js support for .NET Aspire.	8.0.0
Aspire.Hosting.PostgreSQL by Microsoft, 10.8K downloads	PostgreSQL® support for .NET Aspire.	8.0.0
Aspire.Hosting.SqlServer by Microsoft, 6.15K downloads	SQL Server support for .NET Aspire.	8.0.0
- Right panel for **Aspire.Hosting** version 8.0.0:
 - Version:** 8.0.0
 - Author(s):** Microsoft
 - License:** MIT
 - Date published:** Wednesday, May 8, 2024 (5/8/2024)
 - Project URL:** <https://github.com/dotnet/aspire>
 - Report Abuse:** <https://www.nuget.org/packages/Aspire.Hosting/8.0.0/ReportAbuse>
 - Tags:** aspire, hosting
 - Dependencies:**
 - net8.0
 - Grpc.AspNetCore (>= 2.60.0)
 - KubernetesClient (>= 13.0.11)
 - Microsoft.Extensions.Hosting (>= 8.0.0)
 - Polly.Core (>= 8.3.1)

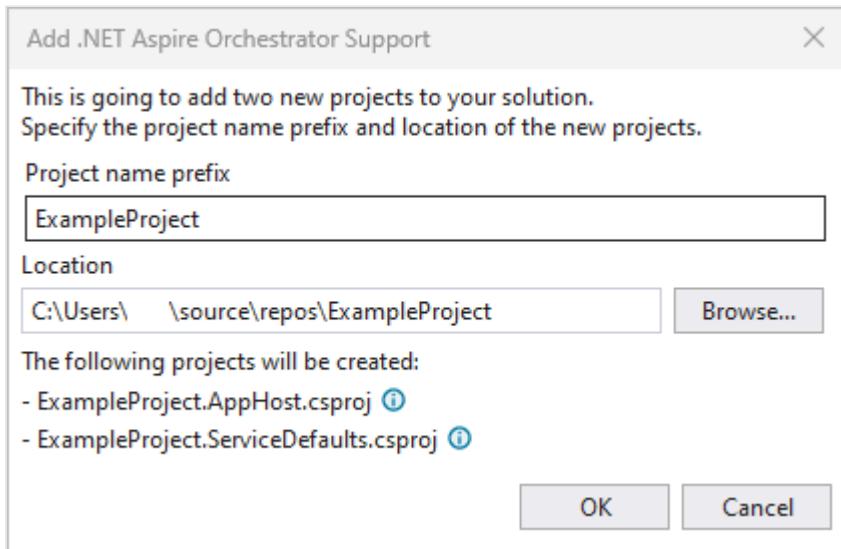
Add orchestration projects

You can add .NET Aspire orchestration projects to an existing app using the following steps:

1. In Visual Studio, right click on an existing project and select **Add > .NET Aspire Orchestrator Support...**



2. A dialog window will open with a summary of the `.AppHost` and `.ServiceDefaults` projects that will be added to your solution.



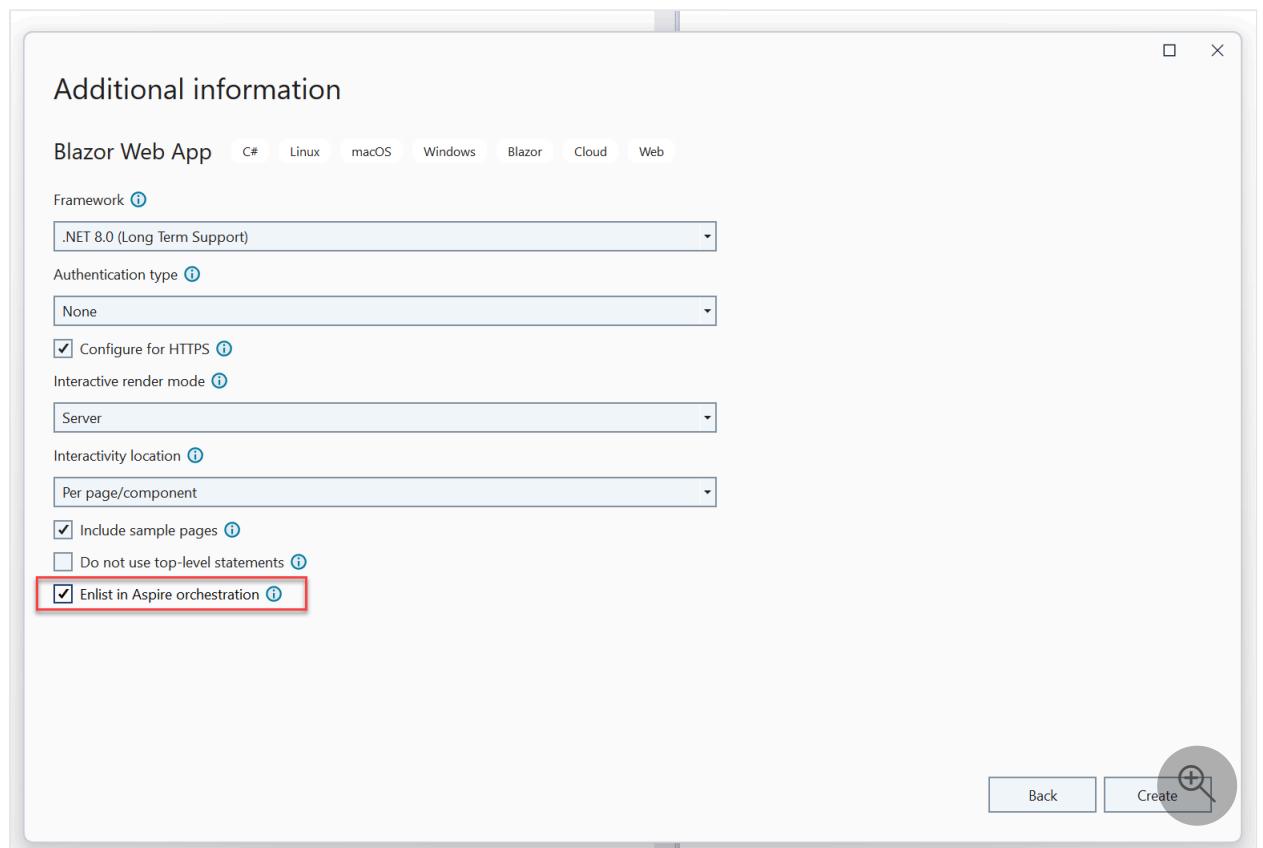
3. Select **OK** and the following changes will be applied:

- The **.AppHost** and **.ServiceDefault** orchestration projects will be added to your solution.
- A call to `builder.AddServiceDefaults` will be added to the *Program.cs* file of your original project.
- A reference to your original project will be added to the *Program.cs* file of the **.AppHost** project.

For more information on .NET Aspire orchestration, see [.NET Aspire orchestration overview](#).

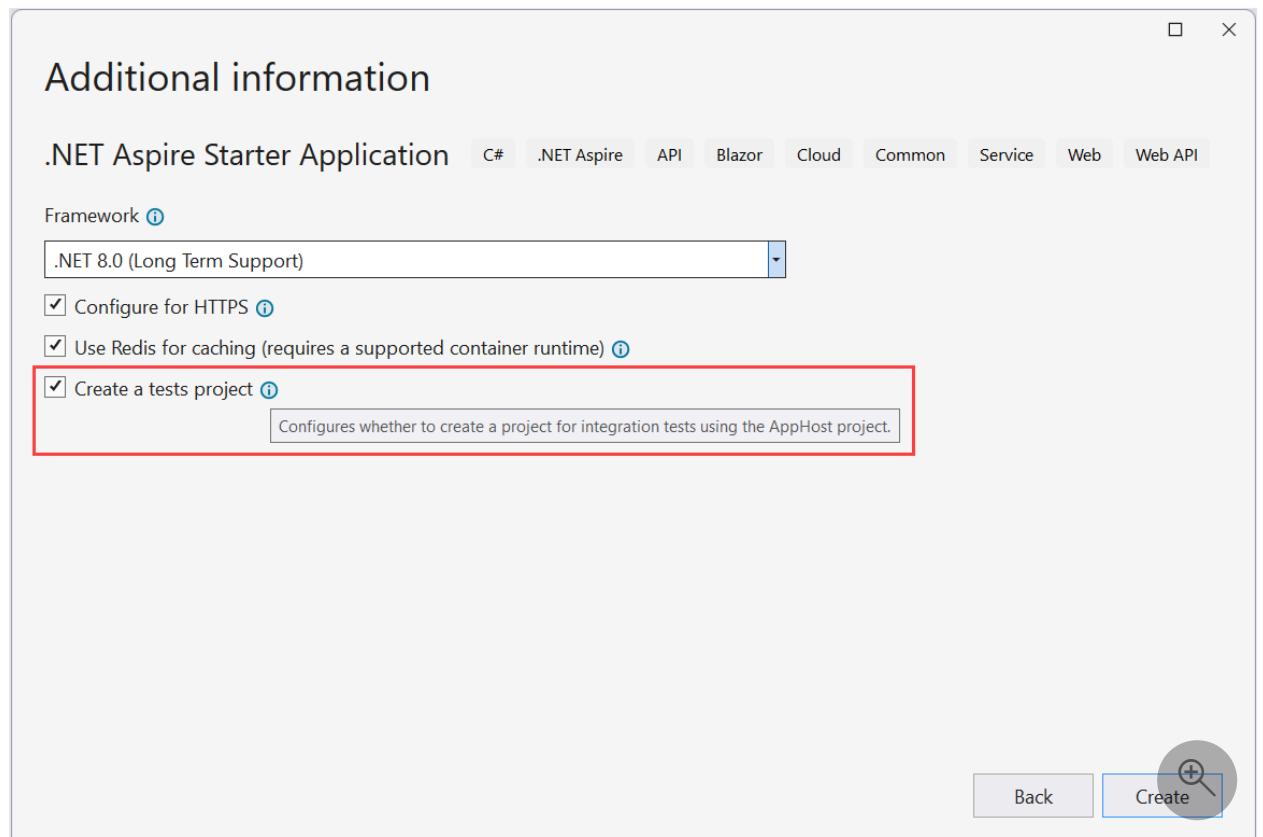
Enlist in orchestration

Visual Studio provides the option to **Enlist in Aspire orchestration** during the new project workflow. Select this option to have Visual Studio create **.AppHost** and **.ServiceDefault** projects alongside your selected project template.



Create test project

When you're using Visual Studio, and you select the **.NET Aspire Start Application** template, you have the option to include a test project. This test project is an xUnit project that includes a sample test that you can use as a starting point for your tests.



For more information, see [Testing .NET Aspire projects](#).

See also

- [Unable to install .NET Aspire workload](#)
- [Use Dev Proxy with .NET Aspire project](#)

.NET Aspire — what's new?

Welcome to what's new in .NET Aspire docs. Use this page to quickly find the latest changes.

.NET Aspire release documentation

WHAT'S NEW

[Announcing .NET Aspire 8.2 ↗](#)

[What's new in .NET Aspire 8.1 ↗](#)

[.NET Aspire 8.0 \(General Availability\) ↗](#)

.NET Aspire GitHub release notes

WHAT'S NEW

[.NET Aspire 8.2.0 ↗](#)

[.NET Aspire 8.1.0 ↗](#)

[.NET Aspire 8.0.2 ↗](#)

[.NET Aspire 8.0.0 ↗](#)

Latest documentation updates

WHAT'S NEW

[August 2024](#)

[July 2024](#)

[June 2024](#)

Find language updates

WHAT'S NEW

[What's new in C#](#)

[What's new in F#](#)

[What's new for Visual Basic](#)

Get involved - contribute

OVERVIEW

[.NET Aspire docs repository ↗](#)

[Project structure and labels for issues and pull requests](#)

[.NET Foundation ↗](#)

CONCEPT

[Microsoft docs contributor guide](#)

[.NET docs contributor guide](#)

.NET developers

DOWNLOAD

[Download the .NET SDK ↗](#)

SAMPLE

[.NET Aspire samples browser](#)

GET STARTED

[.NET on Q&A](#)

[.NET tech community forums ↗](#)

WHAT'S NEW

[Community ↗](#)

Related what's new pages

WHAT'S NEW

[.NET docs updates](#)

[ASP.NET Core docs updates](#)

[Xamarin docs updates](#)

[.NET release notes ↗](#)

[ASP.NET Core release notes](#)

[C# compiler \(Roslyn\) release notes ↗](#)

[Visual Studio release notes](#)

[Visual Studio Code release notes ↗](#)

.NET Aspire orchestration overview

Article • 09/28/2024

.NET Aspire provides APIs for expressing resources and dependencies within your distributed application. In addition to these APIs, [there's tooling](#) that enables several compelling scenarios. The orchestrator is intended for *local development* purposes and isn't supported in production environments.

Before continuing, consider some common terminology used in .NET Aspire:

- **App model:** A collection of resources that make up your distributed application ([DistributedApplication](#)). For a more formal definition, see [Define the app model](#).
- **App host/Orchestrator project:** The .NET project that orchestrates the *app model*, named with the `*.AppHost` suffix (by convention).
- **Resource:** A [resource](#) is a dependent part of an application, such as a .NET project, container, executable, database, cache, or cloud service. It represents any part of the application that can be managed or referenced.
- **Integration:** An integration is a NuGet package for either the *app host* that models a *resource* or a package that configures a client for use in a consuming app. For more information, see [.NET Aspire integrations overview](#).
- **Reference:** A reference defines a connection between resources, expressed as a dependency using the [WithReference](#) API. For more information, see [Reference resources](#).

(!) Note

.NET Aspire's orchestration is designed to enhance your *local development* experience by simplifying the management of your cloud-native app's configuration and interconnections. While it's an invaluable tool for development, it's not intended to replace production environment systems like [Kubernetes](#), which are specifically designed to excel in that context.

Define the app model

.NET Aspire empowers you to seamlessly build, provision, deploy, configure, test, run, and observe your distributed applications. All of these capabilities are achieved through the utilization of an *app model* that outlines the resources in your .NET Aspire solution and their relationships. These resources encompass projects, executables, containers, and external services and cloud resources that your app depends on. Within every .NET

Aspire solution, there's a designated [App host project](#), where the app model is precisely defined using methods available on the [IDistributedApplicationBuilder](#). This builder is obtained by invoking [DistributedApplication.CreateBuilder](#).

C#

```
// Create a new app model builder
var builder = DistributedApplication.CreateBuilder(args);

// TODO:
//   Add resources to the app model
//   Express dependencies between resources

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

App host project

The app host project handles running all of the projects that are part of the .NET Aspire project. In other words, it's responsible for orchestrating all apps within the app model. The following code describes an application with two projects and a Redis cache:

C#

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

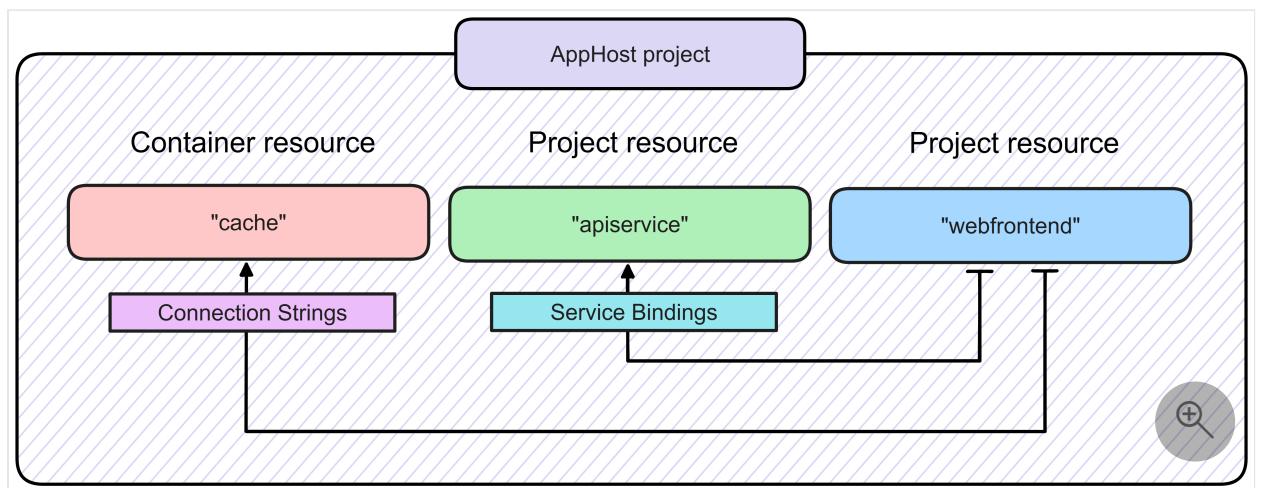
var cache = builder.AddRedis("cache");

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

builder.AddProject<Projects.AspireApp_Web>("webfrontend")
    .WithReference(cache)
    .WithReference(apiservice);

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

To help visualize the relationship between the app host project and the resources it describes, consider the following diagram:



Each resource must be uniquely named. This diagram shows each resource and the relationships between them. The container resource is named "cache" and the project resources are named "apiservice" and "webfrontend". The web frontend project references the cache and API service projects. When you're expressing references in this way, the web frontend project is saying that it depends on these two resources, the "cache" and "apiservice" respectively.

Built-in resource types

.NET Aspire projects are made up of a set of resources. The primary base resource types in the [Aspire.Hosting.AppHost](#) NuGet package are described in the following table:

[] Expand table

Method	Resource type	Description
AddProject	ProjectResource	A .NET project, for example ASP.NET Core web apps.
AddContainer	ContainerResource	A container image, such as a Docker image.
AddExecutable	ExecutableResource	An executable file, such as a Node.js app .
AddParameter	ParameterResource	A parameter resource that can be used to express external parameters .

Project resources represent .NET projects that are part of the app model. When you add a project reference to the app host project, the .NET Aspire SDK generates a type in the `Projects` namespace for each referenced project.

To add a project to the app model, use the `AddProject` method:

C#

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

// Adds the project "apiservice" of type "Projects.AspireApp_ApiService".
var apiservice = builder.AddProject<Projects.AspireApp_ApiService>
("apiservice");
```

Reference resources

A reference represents a dependency between resources. Consider the following example app host `Program` C# code:

C#

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

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

builder.AddProject<Projects.AspireApp_Web>("webfrontend")
    .WithReference(cache);
```

The "webfrontend" project resource uses [WithReference](#) to add a dependency on the "cache" container resource. These dependencies can represent connection strings or [service discovery](#) information. In the preceding example, an environment variable is *injected* into the "webfrontend" resource with the name `ConnectionStrings__cache`. This environment variable contains a connection string that the `webfrontend` uses to connect to Redis via the [.NET Aspire Redis integration](#), for example,

`ConnectionStrings__cache="localhost:62354"`.

APIs for adding and expressing resources

.NET Aspire [hosting integrations](#) and [client integrations](#) are both delivered as NuGet packages, but they serve different purposes. While *client integrations* provide client library configuration for consuming apps outside the scope of the app host, *hosting integrations* provide APIs for expressing resources and dependencies within the app host. For more information, see [Integration responsibilities](#).

Express container resources

To express a container resource, use the [AddContainer](#) method:

C#

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

var ollama = builder.AddContainer("ollama", "ollama/ollama")
    .WithBindMount("ollama", "/root/.ollama")
    .WithBindMount("./ollamaconfig", "/usr/config")
    .WithHttpEndpoint(port: 11434, targetPort: 11434, name: "ollama")
    .WithEntrypoint("/usr/config/entrypoint.sh")
    .WithContainerRuntimeArgs("--gpus=all");
```

For more information, see [GPU support in Docker Desktop](#).

The preceding code adds a container resource named "ollama" with the image "ollama/ollama". The container resource is configured with multiple bind mounts, a named HTTP endpoint, an entrypoint that resolves to Unix shell script, and container run arguments with the [WithContainerRuntimeArgs](#) method.

Beyond the base resource types, [ProjectResource](#), [ContainerResource](#), and [ExecutableResource](#), .NET Aspire provides extension methods to add common resources to your app model. For more information, see [Hosting integrations](#).

Connection string and endpoint references

It's common to express dependencies between project resources. Consider the following example code:

C#

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

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

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

builder.AddProject<Projects.AspireApp_Web>("webfrontend")
    .WithReference(cache)
    .WithReference(apiservice);
```

Project-to-project references are handled differently than resources that have well-defined connection strings. Instead of connection string being injected into the "webfrontend" resource, environment variables to support service discovery are injected.

Method	Environment variable
WithReference(cache)	ConnectionStrings__cache="localhost:62354"
WithReference(apiservice)	services__apiservice__http__0="http://localhost:5455" services__apiservice__https__0="https://localhost:7356"

Adding a reference to the "apiservice" project results in service discovery environment variables being added to the frontend. This is because typically, project-to-project communication occurs over HTTP/gRPC. For more information, see [.NET Aspire service discovery](#).

It's possible to get specific endpoints from a container or executable using the [WithEndpoint](#) and calling the [GetEndpoint](#):

C#

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

var customContainer = builder.AddContainer("myapp", "mycustomcontainer")
    .WithHttpEndpoint(port: 9043, name:
"endpoint");

var endpoint = customContainer.GetEndpoint("endpoint");

var apiservice = builder.AddProject<Projects.AspireApp_ApiService>
("apiservice")
    .WithReference(endpoint);
```

Method	Environment variable
WithReference(endpoint)	services__myapp__endpoint__0=https://localhost:9043

The `port` parameter is the port that the container is listening on. For more information on container ports, see [Container ports](#). For more information on service discovery, see [.NET Aspire service discovery](#).

Service endpoint environment variable format

In the preceding section, the `WithReference` method is used to express dependencies between resources. When service endpoints result in environment variables being

injected into the dependent resource, the format might not be obvious. This section provides details on this format.

When one resource depends on another resource, the app host injects environment variables into the dependent resource. These environment variables configure the dependent resource to connect to the resource it depends on. The format of the environment variables is specific to .NET Aspire and expresses service endpoints in a way that is compatible with [Service Discovery](#).

Service endpoint environment variable names are prefixed with `services_` (double underscore), then the service name, the endpoint name, and finally the index. The index supports multiple endpoints for a single service, starting with `0` for the first endpoint and incrementing for each endpoint.

Consider the following environment variable examples:

Environment

```
services_apiservice_http_0
```

The preceding environment variable expresses the first HTTP endpoint for the `apiservice` service. The value of the environment variable is the URL of the service endpoint. A named endpoint might be expressed as follows:

Environment

```
services_apiservice_myendpoint_0
```

In the preceding example, the `apiservice` service has a named endpoint called `myendpoint`. The value of the environment variable is the URL of the service endpoint.

Reference existing resources

Some situations warrant that you reference an existing resource, perhaps one that is deployed to a cloud provider. For example, you might want to reference an Azure database. In this case, you'd rely on the [Execution context](#) to dynamically determine whether the app host is running in "run" mode or "publish" mode. If you're running locally and want to rely on a cloud resource, you can use the `IsRunMode` property to conditionally add the reference. You might choose to instead create the resource in publish mode. Some [hosting integrations](#) support providing a connection string directly, which can be used to reference an existing resource.

Likewise, there might be use cases where you want to integrate .NET Aspire into an existing solution. One common approach is to add the .NET Aspire app host project to an existing solution. Within your app host, you express dependencies by adding project references to the app host and [building out the app model](#). For example, one project might depend on another. These dependencies are expressed using the [WithReference](#) method. For more information, see [Add .NET Aspire to an existing .NET app](#).

Execution context

The [IDistributedApplicationBuilder](#) exposes an execution context ([DistributedApplicationExecutionContext](#)), which provides information about the current execution of the app host. This context can be used to evaluate whether or not the app host is executing as "run" mode, or as part of a publish operation. Consider the following properties:

- [IsRunMode](#): Returns `true` if the current operation is running.
- [IsPublishMode](#): Returns `true` if the current operation is publishing.

This information can be useful when you want to conditionally execute code based on the current operation. Consider the following example that demonstrates using the `IsRunMode` property. In this case, an extension method is used to generate a stable node name for RabbitMQ for local development runs.

C#

```
private static IResourceBuilder<RabbitMQServerResource>
RunWithStableNodeName(
    this IResourceBuilder<RabbitMQServerResource> builder)
{
    if (builder.ApplicationBuilder.ExecutionContext.IsRunMode)
    {
        builder.WithEnvironment(context =>
        {
            // Set a stable node name so queue storage is consistent between
            sessions
            var nodeName = $"{builder.Resource.Name}@localhost";
            context.EnvironmentVariables["RABBITMQ_NODENAME"] = nodeName;
        });
    }

    return builder;
}
```

The execution context is often used to conditionally add resources or connection strings that point to existing resources. Consider the following example that demonstrates

conditionally adding Redis or a connection string based on the execution context:

C#

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

var redis = builder.ExecutionContext.IsRunMode
    ? builder.AddRedis("redis")
    : builder.AddConnectionString("redis");

builder.AddProject<Projects.WebApplication>("api")
    .WithReference(redis);

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

In the preceding code:

- If the app host is running in "run" mode, a Redis container resource is added.
- If the app host is running in "publish" mode, a connection string is added.

This logic can easily be inverted to connect to an existing Redis resource when you're running locally, and create a new Redis resource when you're publishing.

See also

- [.NET Aspire integrations overview](#)
- [Service discovery in .NET Aspire](#)
- [.NET Aspire service defaults](#)
- [Expressing external parameters](#)
- [.NET Aspire inner-loop networking overview](#)

Orchestrate Node.js apps in .NET Aspire

Article • 09/12/2024

In this article, you learn how to use Node.js and Node Package Manager (`npm`) apps in a .NET Aspire project. The sample app in this article demonstrates [Angular](#), [React](#), and [Vue](#) client experiences. The following .NET Aspire APIs exist to support these scenarios—and they're part of the [Aspire.Hosting.NodeJS](#) NuGet package:

- [Node.js](#): `AddNodeApp`.
- [npm apps](#): `AddNpmApp`.

The difference between these two APIs is that the former is used to host Node.js apps, while the latter is used to host apps that execute from a `package.json` file's `scripts` section—and the corresponding `npm run <script-name>` command.

💡 Tip

The sample source code for this article is available on [GitHub](#), and there are details available on the [Code Samples: .NET Aspire with Angular, React and Vue](#) page.

ⓘ Important

While this article is focused on Single-Page App (SPA) frontend bits, there's an additional Node.js sample available on the [Code Samples: .NET Aspire Node.js sample](#) page, that demonstrates how to use Node.js as a server app with [express](#).

Prerequisites

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

- [.NET 8.0](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or the [.NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)

- [Visual Studio Code](#) (Optional)
- [C# Dev Kit: Extension](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

Additionally, you need to install [Node.js](#) on your machine. The sample app in this article was built with Node.js version 20.12.2 and npm version 10.5.1. To verify your Node.js and npm versions, run the following commands:

```
Node.js
```

```
node --version
```

```
Node.js
```

```
npm --version
```

To download Node.js (including npm), see the [Node.js download page](#).

Clone sample source code

To clone the sample source code from [GitHub](#), run the following command:

```
Bash
```

```
git clone https://github.com/dotnet/aspire-samples.git
```

After cloning the repository, navigate to the *samples/AspireWithJavaScript* folder:

```
Bash
```

```
cd samples/AspireWithJavaScript
```

From this directory, there are six child directories described in the following list:

- **AspireJavaScript.Angular**: An Angular app that consumes the weather forecast API and displays the data in a table.
- **AspireJavaScript.AppHost**: A .NET Aspire project that orchestrates the other apps in this sample. For more information, see [.NET Aspire orchestration overview](#).
- **AspireJavaScript.MinimalApi**: An HTTP API that returns randomly generated weather forecast data.

- **AspireJavaScript.React**: A React app that consumes the weather forecast API and displays the data in a table.
- **AspireJavaScript.ServiceDefaults**: The default shared project for .NET Aspire projects. For more information, see [.NET Aspire service defaults](#).
- **AspireJavaScript.Vue**: A Vue app that consumes the weather forecast API and displays the data in a table.

Install client dependencies

The sample app demonstrates how to use JavaScript client apps that are built on top of Node.js. Each client app was written either using a `npm create` template command or manually. The following table lists the template commands used to create each client app, along with the default port:

 [Expand table](#)

App type	Create template command	Default port
Angular	<code>npm create @angular@latest</code>	4200
React	Didn't use a template.	PORT env var
Vue	<code>npm create vue@latest</code>	5173

💡 Tip

You don't need to run any of these commands, since the sample app already includes the clients. Instead, this is a point of reference from which the clients were created. For more information, see [npm-init](#).

To run the app, you first need to install the dependencies for each client. To do so, navigate to each client folder and run `npm install` (or the `install` alias `npm i`) [commands](#).

Install Angular dependencies

Node.js

```
npm i ./AspireJavaScript.Angular/
```

For more information on the Angular app, see [explore the Angular client](#).

Install React dependencies

```
Node.js
```

```
npm i ./AspireJavaScript.React/
```

For more information on the React app, see [explore the React client](#).

Install Vue dependencies

```
Node.js
```

```
npm i ./AspireJavaScript.Vue/
```

For more information on the Vue app, see [Vue client](#).

Run the sample app

To run the sample app, call the `dotnet run` command given the orchestrator app host `AspireJavaScript.AppHost.csproj` as the `--project` switch:

```
.NET CLI
```

```
dotnet run --project  
./AspireJavaScript.AppHost/AspireJavaScript.AppHost.csproj
```

The [.NET Aspire dashboard](#) launches in your default browser, and each client app endpoint displays under the **Endpoints** column of the **Resources** page. The following image depicts the dashboard for this sample app:

Type	Name	Status	Start time	Source	Endpoints	Logs	Details
Executable	angular	Running	8:02:54 AM	npm run start	http://localhost:62163	View	View
Executable	react	Running	8:02:54 AM	npm run start	http://localhost:62164	View	View
Executable	vue	Running	8:02:54 AM	npm run start	http://localhost:62165	View	View
Project	weatherapi	Running	8:02:54 AM	AspireJavaScript.MinimalApi.csproj	https://localhost:7167/swagger +1	View	View

The `weatherapi` service endpoint resolves to a Swagger UI page that documents the HTTP API. Each client app consumes this service to display the weather forecast data. You can view each client app by navigating to the corresponding endpoint in the .NET

Aspire dashboard. Their screenshots and the modifications made from the template starting point are detailed in the following sections.

In the same terminal session that you used to run the app, press **[ctrl] + [c]** to stop the app.

Explore the app host

To help understand how each client app resource is orchestrated, look to the app host project. The app host requires the [Aspire.Hosting.NodeJS](#) NuGet package to host Node.js apps:

XML

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

<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net8.0</TargetFramework>
  <ImplicitUsings>enable</ImplicitUsings>
  <Nullable>enable</Nullable>
  <IsAspireHost>true</IsAspireHost>
</PropertyGroup>

<ItemGroup>
  <PackageReference Include="Aspire.Hosting.AppHost" Version="8.2.0" />
  <PackageReference Include="Aspire.Hosting.NodeJs" Version="8.2.0" />
</ItemGroup>

<ItemGroup>
  <ProjectReference
Include="..\AspireJavaScript.MinimalApi\AspireJavaScript.MinimalApi.csproj"
/>
</ItemGroup>

<Target Name="RestoreNpm" BeforeTargets="Build" Condition="
'$(DesignTimeBuild)' != 'true' ">
  <ItemGroup>
    <PackageJsons Include="..\*\package.json" />
  </ItemGroup>

    <!-- Install npm packages if node_modules is missing -->
    <Message Importance="Normal" Text="Installing npm packages for %
(PackageJsons.RelativeDir)" Condition="!Exists('%(PackageJsons.RootDir)%
(PackageJsons.Directory)/node_modules') />
    <Exec Command="npm install" WorkingDirectory="%(PackageJsons.RootDir)%
(PackageJsons.Directory)" Condition="!Exists('%(PackageJsons.RootDir)%
(PackageJsons.Directory)/node_modules') />
  </Target>
```

```
</Project>
```

The project file also defines a build target that ensures that the npm dependencies are installed before the app host is built. The app host code (*Program.cs*) declares the client app resources using the `AddNpmApp` API.

C#

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

var weatherApi = builder.AddProject<Projects.AspireJavaScript_MinimalApi>
("weatherapi")
    .WithExternalHttpEndpoints();

builder.AddNpmApp("angular", "../AspireJavaScript.Angular")
    .WithReference(weatherApi)
    .WithHttpEndpoint(env: "PORT")
    .WithExternalHttpEndpoints()
    .PublishAsDockerFile();

builder.AddNpmApp("react", "../AspireJavaScript.React")
    .WithReference(weatherApi)
    .WithEnvironment("BROWSER", "none") // Disable opening browser on npm
start
    .WithHttpEndpoint(env: "PORT")
    .WithExternalHttpEndpoints()
    .PublishAsDockerFile();

builder.AddNpmApp("vue", "../AspireJavaScript.Vue")
    .WithReference(weatherApi)
    .WithHttpEndpoint(env: "PORT")
    .WithExternalHttpEndpoints()
    .PublishAsDockerFile();

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

The preceding code:

- Creates a `DistributedApplicationBuilder`.
- Adds the "weatherapi" service as a project to the app host.
 - Marks the HTTP endpoints as external.
- With a reference to the "weatherapi" service, adds the "angular", "react", and "vue" client apps as npm apps.
 - Each client app is configured to run on a different container port, and uses the `PORT` environment variable to determine the port.
 - All client apps also rely on a *Dockerfile* to build their container image and are configured to express themselves in the publishing manifest as a container from

the `PublishAsDockerfile`.

For more information on inner-loop networking, see [.NET Aspire inner-loop networking overview](#). For more information on deploying apps, see [.NET Aspire manifest format for deployment tool builders](#).

When the app host orchestrates the launch of each client app, it uses the `npm run start` command. This command is defined in the `scripts` section of the `package.json` file for each client app. The `start` script is used to start the client app on the specified port. Each client app relies on a proxy to request the "weatherapi" service.

The proxy is configured in:

- The `proxy.conf.js` file for the Angular client.
- The `webpack.config.js` file for the React client.
- The `vite.config.ts` file for the Vue client.

Explore the Angular client

There are several key modifications from the original Angular template. The first is the addition of a `proxy.conf.js` file. This file is used to proxy requests from the Angular client to the "weatherapi" service.

JavaScript

```
module.exports = {
  "/api": {
    target: process.env["services_weatherapi_https_0"] ||
            process.env["services_weatherapi_http_0"],
    secure: process.env["NODE_ENV"] !== "development",
    pathRewrite: {
      "^/api": ""
    },
  },
};
```

The .NET Aspire app host sets the `services_weatherapi_http_0` environment variable, which is used to resolve the "weatherapi" service endpoint. The preceding configuration proxies HTTP requests that start with `/api` to the target URL specified in the environment variable.

The second update is to the `package.json` file. This file is used to configure the Angular client to run on a different port than the default port. This is achieved by using the `PORT`

environment variable, and the `run-script-os` npm package to set the port.

JSON

```
{  
  "name": "angular-weather",  
  "version": "0.0.0",  
  "engines": {  
    "node": ">=20.12"  
  },  
  "scripts": {  
    "ng": "ng",  
    "start": "run-script-os",  
    "start:win32": "ng serve --port %PORT%",  
    "start:default": "ng serve --port $PORT",  
    "build": "ng build",  
    "watch": "ng build --watch --configuration development",  
    "test": "ng test"  
  },  
  "private": true,  
  "dependencies": {  
    "@angular/animations": "^18.1.1",  
    "@angular/common": "^18.1.1",  
    "@angular/compiler": "^18.1.1",  
    "@angular/core": "^18.1.1",  
    "@angular/forms": "^18.1.1",  
    "@angular/platform-browser": "^18.1.1",  
    "@angular/platform-browser-dynamic": "^18.1.1",  
    "@angular/router": "^18.1.1",  
    "rxjs": "~7.8.0",  
    "tslib": "^2.6.3",  
    "zone.js": "~0.14.8"  
  },  
  "devDependencies": {  
    "@angular-devkit/build-angular": "^18.1.1",  
    "@angular/cli": "^18.1.1",  
    "@angular/compiler-cli": "^18.1.1",  
    "@types/jasmine": "~5.1.0",  
    "jasmine-core": "~5.2.0",  
    "karma": "~6.4.3",  
    "karma-chrome-launcher": "~3.2.0",  
    "karma-coverage": "~2.2.0",  
    "karma-jasmine": "~5.1.0",  
    "karma-jasmine-html-reporter": "~2.1.0",  
    "typescript": "~5.5.3",  
    "run-script-os": "^1.1.6"  
  }  
}
```

The `scripts` section of the `package.json` file is used to define the `start` script. This script is used by the `npm start` command to start the Angular client app. The `start` script is configured to use the `run-script-os` package to set the port, which delegates to the `ng`

`serve` command passing the appropriate `--port` switch based on the OS-appropriate syntax.

In order to make HTTP calls to the "weatherapi" service, the Angular client app needs to be configured to provide the Angular `HttpClient` for dependency injection. This is achieved by using the `provideHttpClient` helper function while configuring the application in the `app.config.ts` file.

TypeScript

```
import { ApplicationConfig } from '@angular/core';
import { provideHttpClient } from '@angular/common/http';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(routes),
    provideHttpClient()
  ]
};
```

Finally, the Angular client app needs to call the `/api/WeatherForecast` endpoint to retrieve the weather forecast data. There are several HTML, CSS, and TypeScript updates, all of which are made to the following files:

- `app.integration.css`: Update the CSS to style the table. ↗
- `app.integration.html`: Update the HTML to display the weather forecast data in a table. ↗
- `app.integration.ts`: Update the TypeScript to call the `/api/WeatherForecast` endpoint and display the data in the table. ↗

TypeScript

```
import { Component, Injectable } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';
import { HttpClient } from '@angular/common/http';
import { WeatherForecasts } from '../types/weatherForecast';

@Injectable()
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
```

```

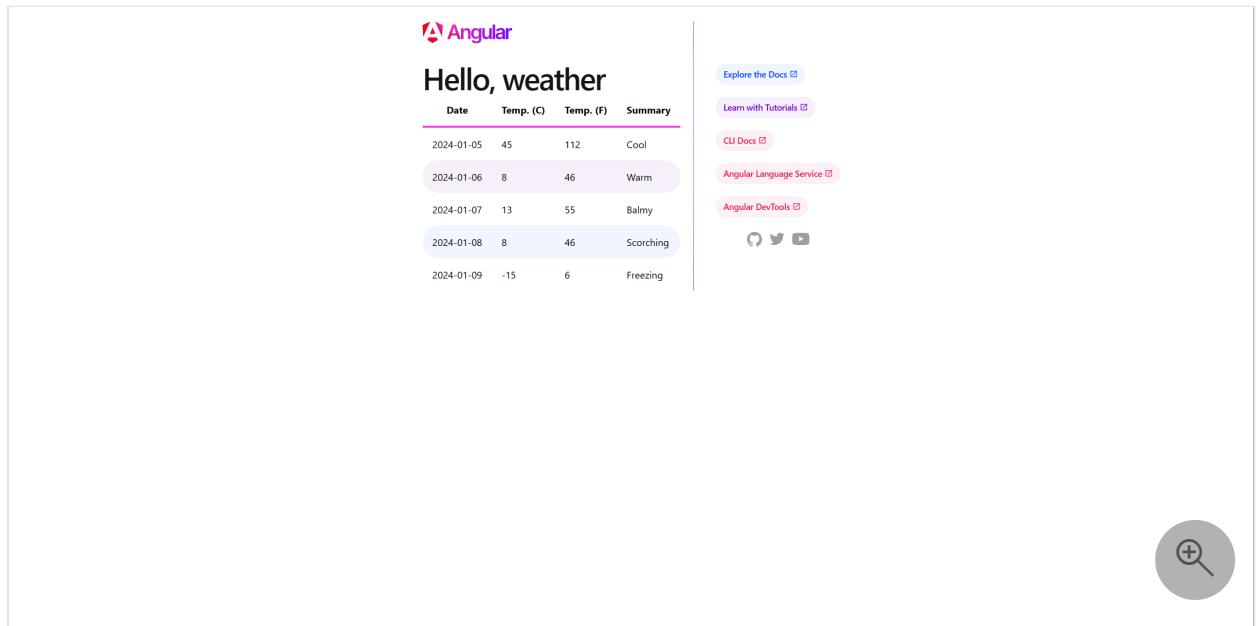
})
export class AppComponent {
  title = 'weather';
  forecasts: WeatherForecasts = [];

  constructor(private http: HttpClient) {
    http.get<WeatherForecasts>('api/weatherforecast').subscribe({
      next: result => this.forecasts = result,
      error: console.error
    });
  }
}

```

Angular app running

To visualize the Angular client app, navigate to the "angular" endpoint in the .NET Aspire dashboard. The following image depicts the Angular client app:



Explore the React client

The React app wasn't written using a template, and instead was written manually. The complete source code can be found in the [dotnet/aspire-samples repository](#). Some of the key points of interest are found in the *src/App.js* file:

JavaScript

```

import { useEffect, useState } from "react";
import "./App.css";

function App() {
  const [forecasts, setForecasts] = useState([]);

```

```
const requestWeather = async () => {
  const weather = await fetch("api/weatherforecast");
  console.log(weather);

  const weatherJson = await weather.json();
  console.log(weatherJson);

  setForecasts(weatherJson);
};

useEffect(() => {
  requestWeather();
}, []);

return (
  <div className="App">
    <header className="App-header">
      <h1>React Weather</h1>
      <table>
        <thead>
          <tr>
            <th>Date</th>
            <th>Temp. (C)</th>
            <th>Temp. (F)</th>
            <th>Summary</th>
          </tr>
        </thead>
        <tbody>
          {(
            forecasts ?? [
              {
                date: "N/A",
                temperatureC: "",
                temperatureF: "",
                summary: "No forecasts",
              },
            ],
          ).map((w) => {
            return (
              <tr key={w.date}>
                <td>{w.date}</td>
                <td>{w.temperatureC}</td>
                <td>{w.temperatureF}</td>
                <td>{w.summary}</td>
              </tr>
            );
          })}
        </tbody>
      </table>
    </header>
  </div>
);
}
```

```
export default App;
```

The `App` function is the entry point for the React client app. It uses the `useState` and `useEffect` hooks to manage the state of the weather forecast data. The `fetch` API is used to make an HTTP request to the `/api/WeatherForecast` endpoint. The response is then converted to JSON and set as the state of the weather forecast data.

JavaScript

```
const HTMLWebpackPlugin = require("html-webpack-plugin");

module.exports = (env) => {
  return {
    entry: "./src/index.js",
    devServer: {
      port: env.PORT || 4001,
      allowedHosts: "all",
      proxy: [
        {
          context: ["/api"],
          target: process.env.services_weatherapi_https_0 ||
            process.env.services_weatherapi_http_0,
          pathRewrite: { "/api": "" },
          secure: false,
        },
      ],
    },
    output: {
      path: `${__dirname}/dist`,
      filename: "bundle.js",
    },
    plugins: [
      new HTMLWebpackPlugin({
        template: "./src/index.html",
        favicon: "./src/favicon.ico",
      }),
    ],
    module: {
      rules: [
        {
          test: /\.js$/,
          exclude: /node_modules/,
          use: {
            loader: "babel-loader",
            options: {
              presets: [
                "@babel/preset-env",
                ["@babel/preset-react", { runtime: "automatic" }],
              ],
            },
          },
        },
      ],
    },
  };
};
```

```
        },
      },
      {
        test: /\.css$/,
        exclude: /node_modules/,
        use: ["style-loader", "css-loader"],
      },
    ],
  },
};
```

The preceding code defines the `module.exports` as follows:

- The `entry` property is set to the `src/index.js` file.
 - The `devServer` relies on a proxy to forward requests to the "weatherapi" service, sets the port to the `PORT` environment variable, and allows all hosts.
 - The `output` results in a `dist` folder with a `bundle.js` file.
 - The `plugins` set the `src/index.html` file as the template, and expose the `favicon.ico` file.

The final updates are to the following files:

- *App.css*: Update the CSS to style the table. ↵
 - *App.js*: Update the JavaScript to call the /api/WeatherForecast endpoint and display the data in the table. ↵

React app running

To visualize the React client app, navigate to the "react" endpoint in the .NET Aspire dashboard. The following image depicts the React client app:

The screenshot shows a dark-themed React application titled "React Weather". At the top, there's a navigation bar with a logo and the title. Below it is a table with five columns: Date, Temp. (C), Temp. (F), and Summary. The table contains five rows of weather forecast data. At the bottom right is a circular search icon with a magnifying glass and a plus sign.

Date	Temp. (C)	Temp. (F)	Summary
2024-04-25	-12	11	Freezing
2024-04-26	3	37	Sweltering
2024-04-27	6	42	Balmy
2024-04-28	8	46	Bracing
2024-04-29	21	69	Chilly

Explore the Vue client

There are several key modifications from the original Vue template. The primary updates were the addition of the `fetch` call in the `TheWelcome.vue` file to retrieve the weather forecast data from the `/api/WeatherForecast` endpoint. The following code snippet demonstrates the `fetch` call:

HTML

```
<script lang="ts">
interface WeatherForecast {
  date: string
  temperatureC: number
  temperatureF: number
  summary: string
};

type Forecasts = WeatherForecast[];

export default {
  name: 'TheWelcome',
  data() {
    return {
      forecasts: [],
      loading: true,
      error: null
    }
  },
  mounted() {
    fetch('api/weatherforecast')
      .then(response => response.json())
      .then(data => {
        this.forecasts = data
      })
  }
},
```

```
)  
    .catch(error => {  
      this.error = error  
    })  
    .finally(() => (this.loading = false))  
  }  
}  
</script>  
  
<template>  
  <table>  
    <thead>  
      <tr>  
        <th>Date</th>  
        <th>Temp. (C)</th>  
        <th>Temp. (F)</th>  
        <th>Summary</th>  
      </tr>  
    </thead>  
    <tbody>  
      <tr v-for="forecast in (forecasts as Forecasts)">  
        <td>{{ forecast.date }}</td>  
        <td>{{ forecast.temperatureC }}</td>  
        <td>{{ forecast.temperatureF }}</td>  
        <td>{{ forecast.summary }}</td>  
      </tr>  
    </tbody>  
  </table>  
</template>  
  
<style>  
table {  
  border: none;  
  border-collapse: collapse;  
}  
  
th {  
  font-size: x-large;  
  font-weight: bold;  
  border-bottom: solid .2rem hsla(160, 100%, 37%, 1);  
}  
  
th,  
td {  
  padding: 1rem;  
}  
  
td {  
  text-align: center;  
  font-size: large;  
}  
  
tr:nth-child(even) {  
  background-color: var(--vt-c-black-soft);
```

```
}
```

```
</style>
```

As the `TheWelcome` integration is `mounted`, it calls the `/api/weatherforecast` endpoint to retrieve the weather forecast data. The response is then set as the `forecasts` data property. To set the server port, the Vue client app uses the `PORT` environment variable. This is achieved by updating the `vite.config.ts` file:

TypeScript

```
import { fileURLToPath, URL } from 'node:url'

import { defineConfig } from 'vite'
import vue from '@vitejs/plugin-vue'

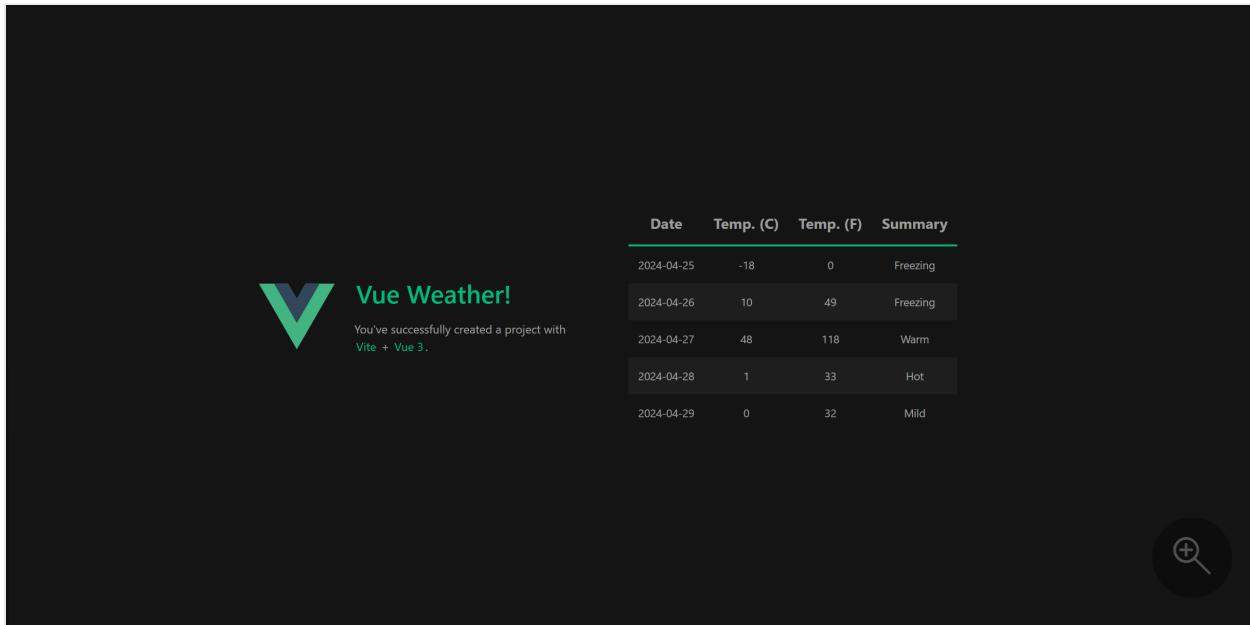
// https://vitejs.dev/config/
export default defineConfig({
  plugins: [
    vue(),
  ],
  resolve: {
    alias: {
      '@': fileURLToPath(new URL('./src', import.meta.url))
    }
  },
  server: {
    host: true,
    port: parseInt(process.env.PORT ?? "5173"),
    proxy: {
      '/api': {
        target: process.env.services_weatherapi_https_0 ||
process.env.services_weatherapi_http_0,
        changeOrigin: true,
        rewrite: path => path.replace(/^\/api/, ''),
        secure: false
      }
    }
  }
})
```

Additionally, the Vite config specifies the `server.proxy` property to forward requests to the "weatherapi" service. This is achieved by using the `services_weatherapi_http_0` environment variable, which is set by the .NET Aspire app host.

The final update from the template is made to the `TheWelcome.vue` file. This file calls the `/api/WeatherForecast` endpoint to retrieve the weather forecast data, and displays the data in a table. It includes [CSS, HTML, and TypeScript updates](#).

Vue app running

To visualize the Vue client app, navigate to the "vue" endpoint in the .NET Aspire dashboard. The following image depicts the Vue client app:



Deployment considerations

The sample source code for this article is designed to run locally. Each client app deploys as a container image. The *Dockerfile* for each client app is used to build the container image. Each *Dockerfile* is identical, using a multistage build to create a production-ready container image.

Dockerfile

```
FROM node:20 as build

WORKDIR /app

COPY package.json package.json
COPY package-lock.json package-lock.json

RUN npm install

COPY . .

RUN npm run build

FROM nginx:alpine

COPY --from=build /app/default.conf.template
/etc/nginx/templates/default.conf.template
COPY --from=build /app/dist/weather/browser /usr/share/nginx/html
```

```
# Expose the default nginx port
EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

The client apps are currently configured to run as true SPA apps, and aren't configured to run in a server-side rendered (SSR) mode. They sit behind `nginx`, which is used to serve the static files. They use a `default.conf.template` file to configure `nginx` to proxy requests to the client app.

```
nginx

server {
    listen      ${PORT};
    listen  [::]:${PORT};
    server_name localhost;

    access_log  /var/log/nginx/server.access.log  main;

    location / {
        root /usr/share/nginx/html;
        try_files $uri $uri/ /index.html;
    }

    location /api/ {
        proxy_pass ${services__weatherapi__https__0};
        proxy_http_version 1.1;
        proxy_ssl_server_name on;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        rewrite ^/api(/.*)$ $1 break;
    }
}
```

Node.js server app considerations

While this article focuses on client apps, you might have scenarios where you need to host a Node.js server app. The same semantics are required to host a Node.js server app as a SPA client app. The .NET Aspire app host requires a package reference to the [Aspire.Hosting.NodeJS](#) NuGet package and the code needs to call either `AddNodeApp` or `AddNpmApp`. These APIs are useful for adding existing JavaScript apps to the .NET Aspire app host.

When configuring secrets and passing environment variables to JavaScript-based apps, whether they are client or server apps, use parameters. For more information, see [.NET Aspire: External parameters—secrets](#).

Use the OpenTelemetry JavaScript SDK

To export OpenTelemetry logs, traces, and metrics from a Node.js server app, you use the [OpenTelemetry JavaScript SDK](#).

For a complete example of a Node.js server app using the OpenTelemetry JavaScript SDK, you can refer to the [Code Samples: .NET Aspire Node.js sample](#) page. Consider the sample's *instrumentation.js* file, which demonstrates how to configure the OpenTelemetry JavaScript SDK to export logs, traces, and metrics:

JavaScript

```
import { env } from 'node:process';
import { NodeSDK } from '@opentelemetry/sdk-node';
import { OTLPTTraceExporter } from '@opentelemetry/exporter-trace-otlp-grpc';
import { OTLPMetricExporter } from '@opentelemetry/exporter-metrics-otlp-
grpc';
import { OTLPLogExporter } from '@opentelemetry/exporter-logs-otlp-grpc';
import { SimpleLogRecordProcessor } from '@opentelemetry/sdk-logs';
import { PeriodicExportingMetricReader } from '@opentelemetry/sdk-metrics';
import { HttpInstrumentation } from '@opentelemetry/instrumentation-http';
import { ExpressInstrumentation } from '@opentelemetry/instrumentation-
express';
import { RedisInstrumentation } from '@opentelemetry/instrumentation-redis-
4';
import { diag, DiagConsoleLogger, DiagLogLevel } from '@opentelemetry/api';
import { credentials } from '@grpc/grpc-js';

const environment = process.env.NODE_ENV || 'development';

// For troubleshooting, set the log level to DiagLogLevel.DEBUG
//diag.setLogger(new DiagConsoleLogger(), environment === 'development' ?
DiagLogLevel.INFO : DiagLogLevel.WARN);

const otlpServer = env.OTEL_EXPORTER_OTLP_ENDPOINT;

if (otlpServer) {
  console.log(`OTLP endpoint: ${otlpServer}`);

  const isHttps = otlpServer.startsWith('https://');
  const collectorOptions = {
    credentials: !isHttps
      ? credentials.createInsecure()
      : credentials.createSsl()
  };
}

const sdk = new NodeSDK({
  traceExporter: new OTLPTTraceExporter(collectorOptions),
  metricReader: new PeriodicExportingMetricReader({
    exportIntervalMillis: environment === 'development' ? 5000 :
10000,
    exporter: new OTLPMetricExporter(collectorOptions),
  })
});
```

```
        },
        logRecordProcessor: new SimpleLogRecordProcessor({
            exporter: new OTLPLogExporter(collectorOptions)
        }),
        instrumentations: [
            new HttpInstrumentation(),
            new ExpressInstrumentation(),
            new RedisInstrumentation()
        ],
    });
}

sdk.start();
}
```

💡 Tip

To configure the .NET Aspire dashboard OTEL CORS settings, see the [.NET Aspire dashboard OTEL CORS settings](#) page.

Summary

While there are several considerations that are beyond the scope of this article, you learned how to build .NET Aspire projects that use Node.js and Node Package Manager (`npm`). You also learned how to use the `AddNpmApp` APIs to host Node.js apps and apps that execute from a `package.json` file, respectively. Finally, you learned how to use the `npm` CLI to create Angular, React, and Vue client apps, and how to configure them to run on different ports.

See also

- [Code Samples: .NET Aspire with Angular, React, and Vue](#)
- [Code Samples: .NET Aspire Node.js App](#)

Orchestrate Python apps in .NET Aspire

Article • 07/23/2024

In this article, you learn how to use Python apps in a .NET Aspire project. The sample app in this article demonstrates launching a Python application. The Python extension for .NET Aspire requires the use of virtual environments.

Prerequisites

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

- [.NET 8.0](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or [the .NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

Additionally, you need to install [Python](#) on your machine. The sample app in this article was built with Python version 3.12.4 and pip version 24.1.2. To verify your Python and pip versions, run the following commands:

```
Python
```

```
python --version
```

```
Python
```

```
pip --version
```

To download Python (including `pip`), see the [Python download page](#).

Create a .NET Aspire project using the template

To get started launching a Python project in .NET Aspire first use the starter template to create a .NET Aspire application host:

```
.NET CLI
```

```
dotnet new aspire -o PythonSample
```

In the same terminal session, change directories into the newly created project:

```
.NET CLI
```

```
cd PythonSample
```

Once the template has been created launch the app host with the following command to ensure that the app host and the [.NET Aspire dashboard](#) launches successfully:

```
.NET CLI
```

```
dotnet run --project PythonSample.AppHost/PythonSample.AppHost.csproj
```

Once the app host starts it should be possible to click on the dashboard link in the console output. At this point the dashboard will not show any resources. Stop the app host by pressing **Ctrl** + **C** in the terminal.

Prepare a Python project

From your previous terminal session where you created the .NET Aspire solution, create a new directory to contain the Python source code.

```
Console
```

```
mkdir hello-python
```

Change directories into the newly created *hello-python* directory:

```
Console
```

```
cd hello-python
```

Initialize the Python virtual environment

To work with Python projects, they need to be within a virtual environment. To create a virtual environment, run the following command:

Python

```
python -m venv .venv
```

For more information on virtual environments, see the [Python: Install packages in a virtual environment using pip and venv ↗](#).

To activate the virtual environment, enabling installation and usage of packages, run the following command:

Unix/macOS

Bash

```
source .venv/bin/activate
```

Ensure that pip within the virtual environment is up-to-date by running the following command:

Python

```
python -m pip install --upgrade pip
```

Install Python packages

Install the Flask package by creating a *requirements.txt* file in the *hello-python* directory and adding the following line:

Python

```
Flask==3.0.3
```

Then, install the Flask package by running the following command:

Python

```
python -m pip install -r requirements.txt
```

After Flask is installed, create a new file named *main.py* in the *hello-python* directory and add the following code:

```
Python

import os
import flask

app = flask.Flask(__name__)

@app.route('/', methods=['GET'])
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    port = int(os.environ.get('PORT', 8111))
    app.run(host='0.0.0.0', port=port)
```

The preceding code creates a simple Flask app that listens on port 8111 and returns the message "Hello, World!" when the root endpoint is accessed.

Update the app host project

Install the Python hosting package by running the following command:

```
.NET CLI

dotnet add ..\PythonSample.AppHost\PythonSample.AppHost.csproj package
Aspire.Hosting.Python --version 8.1.0
```

After the package is installed, the project XML should have a new package reference similar to the following:

```
XML

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

    <PropertyGroup>
        <OutputType>Exe</OutputType>
        <TargetFramework>net8.0</TargetFramework>
        <ImplicitUsings>enable</ImplicitUsings>
        <Nullable>enable</Nullable>
        <IsAspireHost>true</IsAspireHost>
    </PropertyGroup>

    <ItemGroup>
        <PackageReference Include="Aspire.Hosting.AppHost" Version="8.1.0" />
```

```
<!-- Add this reference to PythonSample.AppHost.csproj -->
<PackageReference Include="Aspire.Hosting.Python" Version="8.1.0" />
</ItemGroup>

</Project>
```

Update the app host *Program.cs* file to include the Python project, by calling the `AddPythonProject` API and specifying the project name, project path, and the entry point file:

C#

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

builder.AddPythonProject("hello-python", "../hello-python", "main.py")
    .WithEndpoint(targetPort: 8111, scheme: "http", env: "PORT")
    //WithEnvironment("OTEL_PYTHON_OTLP_TRACES_SSL", "false")
    ;

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

Run the app

Now that you've added the Python hosting package, updated the app host *Program.cs* file, and created a Python project, you can run the app host:

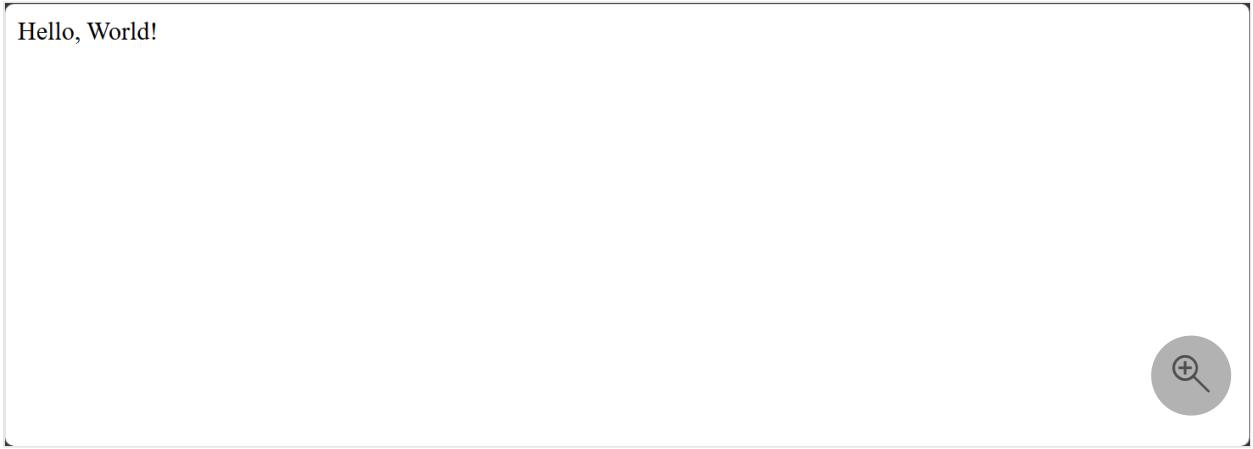
.NET CLI

```
dotnet run --project ../PythonSample.AppHost/PythonSample.AppHost.csproj
```

Launch the dashboard by clicking the link in the console output. The dashboard should display the Python project as a resource.

Type	Name	State	Start time	Source	Endpoints	Logs	Details
Executable	hello-python	Running	10:33:25 AM	python.exe main.py	http://localhost:56721	View	View

Select the **Endpoints** link to open the `hello-python` endpoint in a new browser tab. The browser should display the message "Hello, World!":



Hello, World!



Stop the app host by pressing `ctrl` + `c` in the terminal.

Add telemetry support.

To add a bit of observability, add telemetry to help monitor the dependant Python app. In the Python project, add the following OpenTelemetry package as a dependency in the `requirements.txt` file:



Python

```
Flask==3.0.3
opentelemetry-distro[otlp]
```

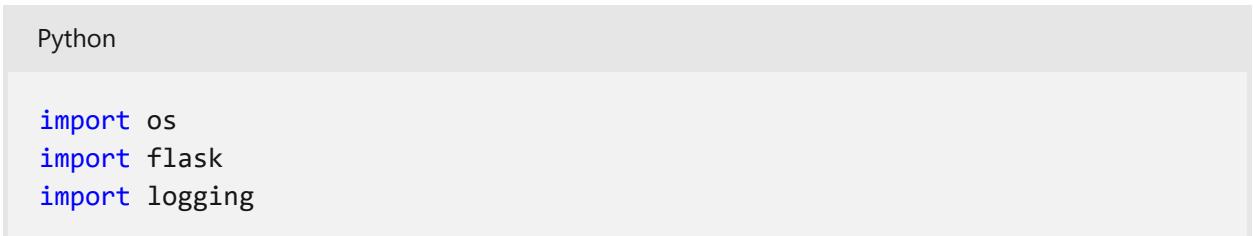
The preceding requirement update, adds the OpenTelemetry package and the OTLP exporter. Next, re-install the Python app requirements into the virtual environment by running the following command:



Python

```
python -m pip install -r requirements.txt
```

The preceding command installs the OpenTelemetry package and the OTLP exporter, in the virtual environment. Update the Python app to include the OpenTelemetry code, by replacing the existing `main.py` code with the following:



Python

```
import os
import flask
import logging
```

```

logging.basicConfig()
logging.getLogger().setLevel(logging.NOTSET)

app = flask.Flask(__name__)

@app.route('/', methods=['GET'])
def hello_world():
    logging.getLogger(__name__).info("request received!")
    return 'Hello, World!'

if __name__ == '__main__':
    port = int(os.environ.get('PORT', 8111))
    app.run(host='0.0.0.0', port=port)

```

Update the app host project's *launchSettings.json* file to include the `ASPIRE_ALLOW_UNSECURED_TRANSPORT` environment variable:

JSON

```
{
  "$schema": "https://json.schemastore.org/launchsettings.json",
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "http://localhost:15209",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "DOTNET_ENVIRONMENT": "Development",
        "DOTNET_DASHBOARD_OTLP_ENDPOINT_URL": "http://localhost:19171",
        "DOTNET_RESOURCE_SERVICE_ENDPOINT_URL": "http://localhost:20208",
        "ASPIRE_ALLOW_UNSECURED_TRANSPORT": "true"
      }
    }
  }
}
```

The `ASPIRE_ALLOW_UNSECURED_TRANSPORT` variable is required because when running locally the OpenTelemetry client in Python rejects the local development certificate. Launch the *app host* again:

.NET CLI

```
dotnet run --project ../PythonSample.AppHost/PythonSample.AppHost.csproj
```

Once the app host has launched navigate to the dashboard and note that in addition to console log output, structured logging is also being routed through to the dashboard.

The screenshot shows the .NET Aspire application insights interface for a project named "PythonSample". The left sidebar has tabs for "Resources", "Console", "Structured", "Traces", and "Metrics", with "Structured" currently selected. The main area is titled "Structured logs" and displays a table of log entries. The table has columns: Resource, Level, Timestamp, Message, Trace, and Data. There are 5 results found. The first few log entries are:

Resource	Level	Timestamp	Message	Trace	Data
hello-pyt...	Information	12:18:50.3...	Press CTRL+C to quit[0m	0000000	View
hello-pyt...	Information	12:18:50.3...	[31m[1mWARNING: This is a development server. Do not use it in a pr...	0000000	View
hello-pyt...	Information	12:18:59.1...	127.0.0.1 - - [23/Jul/2024 12:18:59] "GET / HTTP/1.1" 200 -	0000000	View
hello-pyt...	Information	12:18:59.1...	request received!	0000000	View
hello-pyt...	Information	12:18:59.2...	127.0.0.1 - - [23/Jul/2024 12:18:59] "[33mGET /favicon.ico HTTP/1.1[0..."	0000000	View

Total: 5 results found

Summary

While there are several considerations that are beyond the scope of this article, you learned how to build .NET Aspire solution that integrates with Python. You also learned how to use the `AddPythonProject` API to host Python apps.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Add Dockerfiles to your .NET app model

Article • 07/23/2024

With .NET Aspire it's possible to specify a *Dockerfile* to build when the [app host](#) is started using either the [AddDockerfile](#) or [WithDockerfile](#) extension methods.

Add a Dockerfile to the app model

In the following example the [AddDockerfile](#) extension method is used to specify a container by referencing the context path for the container build.

C#

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

var container = builder.AddDockerfile(
    "mycontainer", "relative/context/path");
```

Unless the context path argument is a rooted path the context path is interpreted as being relative to the app host projects directory (where the AppHost `*.csproj` folder is located).

By default the name of the *Dockerfile* which is used is `Dockerfile` and is expected to be within the context path directory. It's possible to explicitly specify the *Dockerfile* name either as an absolute path or a relative path to the context path.

This is useful if you wish to modify the specific *Dockerfile* being used when running locally or when the app host is deploying.

C#

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

var container = builder.ExecutionContext.IsRunMode
    ? builder.AddDockerfile(
        "mycontainer", "relative/context/path", "Dockerfile.debug")
    : builder.AddDockerfile(
        "mycontainer", "relative/context/path", "Dockerfile.release");
```

Customize existing container resources

When using [AddDockerfile](#) the return value is an `IResourceBuilder<ContainerResource>`. .NET Aspire includes many custom resource types that are derived from [ContainerResource](#).

Using the [WithDockerfile](#) extension method it's possible to continue using these strongly typed resource types and customize the underlying container that is used.

C#

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

var pgsql = builder.AddPostgres("pgsql")
    .WithDockerfile("path/to/context")
    .WithPgAdmin();
```

Pass build arguments

The [WithBuildArg](#) method can be used to pass arguments into the container image build.

C#

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

var container = builder.AddDockerfile("mygoapp", "relative/context/path")
    .WithBuildArg("GO_VERSION", "1.22");
```

The value parameter on the [WithBuildArg](#) method can be a literal value (`boolean`, `string`, `int`) or it can be a resource builder for a [parameter resource](#). The following code replaces the `GO_VERSION` with a parameter value that can be specified at deployment time.

C#

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

var goVersion = builder.AddParameter("goversion");

var container = builder.AddDockerfile("mygoapp", "relative/context/path")
    .WithBuildArg("GO_VERSION", goVersion);
```

Build arguments correspond to the [ARG command](#) in *Dockerfiles*. Expanding the preceding example, this is a multi-stage *Dockerfile* which specifies specific container image version to use as a parameter.

Dockerfile

```
# Stage 1: Build the Go program
ARG GO_VERSION=1.22
FROM golang:${GO_VERSION} AS builder
WORKDIR /build
COPY .
RUN go build mygoapp.go

# Stage 2: Run the Go program
FROM mcr.microsoft.com/cbl-mariner/base/core:2.0
WORKDIR /app
COPY --from=builder /build/mygoapp .
CMD ["../mygoapp"]
```

! Note

Instead of hardcoding values into the container image, it's recommended to use environment variables for values that frequently change. This avoids the need to rebuild the container image whenever a change is required.

Pass build secrets

In addition to build arguments it's possible to specify build secrets using [WithBuildSecret](#) which are made selectively available to individual commands in the *Dockerfile* using the `--mount=type=secret` syntax on `RUN` commands.

C#

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

var accessToken = builder.AddParameter("accesstoken", secret: true);

var container = builder.AddDockerfile("myapp", "relative/context/path")
    .WithBuildSecret("ACCESS_TOKEN", accessToken);
```

For example, consider the `RUN` command in a *Dockerfile* which exposes the specified secret to the specific command:

Dockerfile

```
# The helloworld command can read the secret from /run/secrets/ACCESS_TOKEN
RUN --mount=type=secret,id=ACCESS_TOKEN helloworld
```

Caution

Caution should be exercised when passing secrets in build environments. This is often done when using a token to retrieve dependencies from private repositories or feeds before a build. It is important to ensure that the injected secrets are not copied into the final or intermediate images.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

.NET Aspire inner-loop networking overview

Article • 08/20/2024

One of the advantages of developing with .NET Aspire is that it enables you to develop, test, and debug cloud-native apps locally. Inner-loop networking is a key aspect of .NET Aspire that allows your apps to communicate with each other in your development environment. In this article, you learn how .NET Aspire handles various networking scenarios with proxies, service bindings, endpoint configurations, and launch profiles.

Networking in the inner loop

The inner loop is the process of developing and testing your app locally before deploying it to a target environment. .NET Aspire provides several tools and features to simplify and enhance the networking experience in the inner loop, such as:

- **Launch profiles:** Launch profiles are configuration files that specify how to run your app locally. You can use launch profiles (such as the `launchSettings.json` file) to define the service bindings, environment variables, and launch settings for your app.
- **Kestrel configuration:** Kestrel configuration allows you to specify the endpoints that the Kestrel web server listens on. You can configure Kestrel endpoints in your app settings, and .NET Aspire automatically uses these settings to create service bindings.
- **Service bindings/Endpoint configurations:** Service bindings are the connections between your app and the services it depends on, such as databases, message queues, or APIs. Service bindings provide information such as the service name, host port, scheme, and environment variable. You can add service bindings to your app either implicitly (via launch profiles) or explicitly by calling [WithEndpoint](#).
- **Proxies:** .NET Aspire automatically launches a proxy for each service binding you add to your app, and assigns a port for the proxy to listen on. The proxy then forwards the requests to the port that your app listens on, which might be different from the proxy port. This way, you can avoid port conflicts and access your app and services using consistent and predictable URLs.

How service bindings work

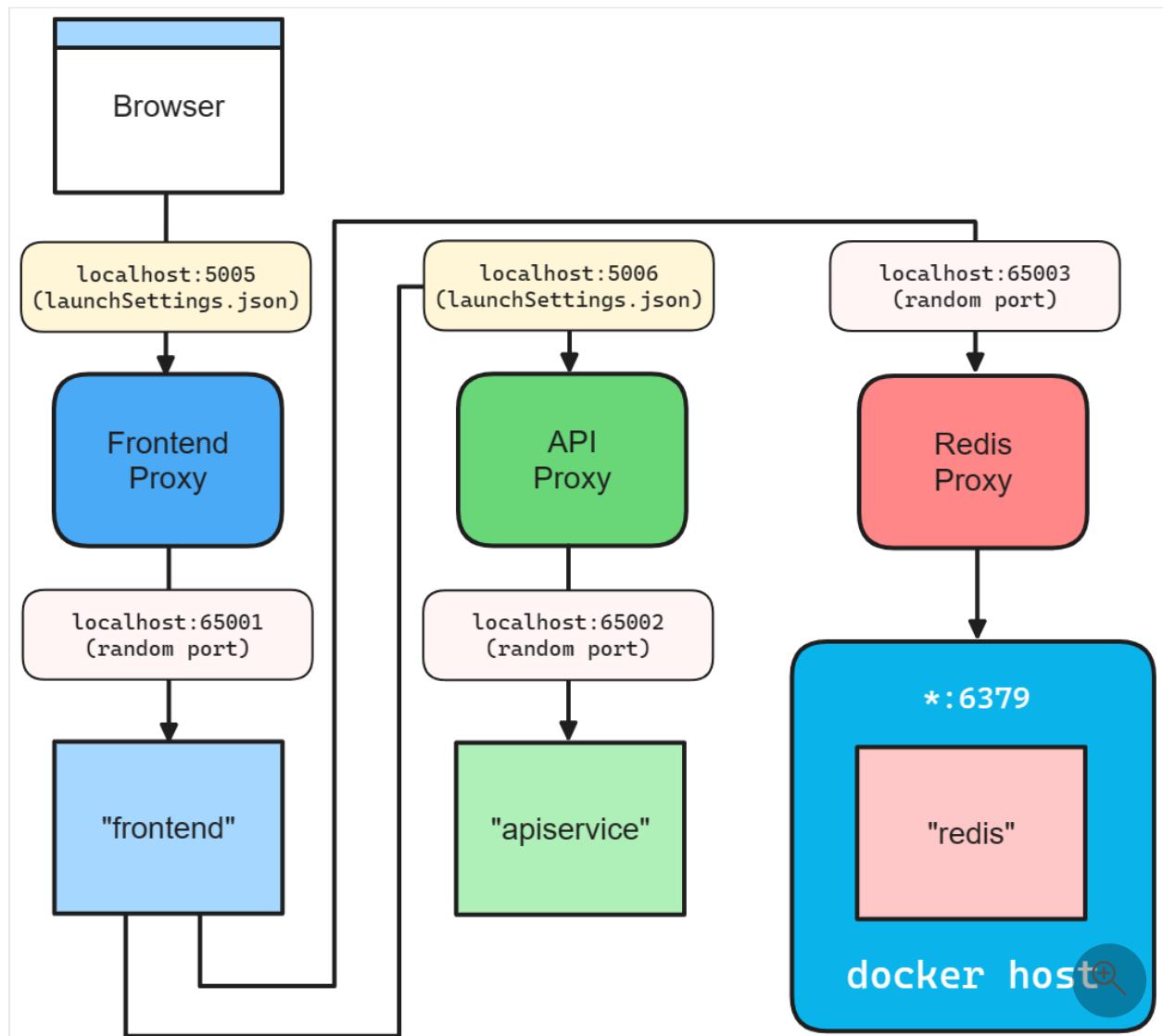
A service binding in .NET Aspire involves two integrations: a **service** representing an external resource your app requires (for example, a database, message queue, or API),

and a **binding** that establishes a connection between your app and the service and provides necessary information.

.NET Aspire supports two service binding types: **implicit**, automatically created based on specified launch profiles defining app behavior in different environments, and **explicit**, manually created using [WithEndpoint](#).

Upon creating a binding, whether implicit or explicit, .NET Aspire launches a lightweight reverse proxy on a specified port, handling routing and load balancing for requests from your app to the service. The proxy is a .NET Aspire implementation detail, requiring no configuration or management concern.

To help visualize how service bindings work, consider the .NET Aspire starter templates inner-loop networking diagram:



Launch profiles

When you call [AddProject](#), the app host looks for *Properties/launchSettings.json* to determine the default set of service bindings. The app host selects a specific launch

profile using the following rules:

1. An explicit `launchProfileName` argument passed when calling `AddProject`.
2. The `DOTNET_LAUNCH_PROFILE` environment variable. For more information, see [.NET environment variables](#).
3. The first launch profile defined in `launchSettings.json`.

Consider the following `launchSettings.json` file:

```
JSON

{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": false,
      "inspectUri": "{wsProtocol}://{url.hostname}:
{url.port}/_framework/debug/ws-proxy?browser={browserInspectUri}",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "inspectUri": "{wsProtocol}://{url.hostname}:
{url.port}/_framework/debug/ws-proxy?browser={browserInspectUri}",
      "applicationUrl": "https://localhost:7239;http://localhost:5066",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

For the remainder of this article, imagine that you've created an `IDistributedApplicationBuilder` assigned to a variable named `builder` with the `CreateBuilder()` API:

```
C#

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

To specify the `http` and `https` launch profiles, configure the `applicationUrl` values for both in the `launchSettings.json` file. These URLs are used to create service bindings for

this project. This is the equivalent of:

C#

```
builder.AddProject<Projects.Networking_Frontend>("frontend")
    .WithHttpEndpoint(port: 5066)
    .WithHttpsEndpoint(port: 7239);
```

ⓘ Important

If there's no *launchSettings.json* (or launch profile), there are no bindings by default.

For more information, see [.NET Aspire and launch profiles](#).

Kestrel configured endpoints

.NET Aspire supports Kestrel endpoint configuration. For example, consider an *appsettings.json* file for a project that defines a Kestrel endpoint with the HTTPS scheme and port 5271:

JSON

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "Kestrel": {
    "Endpoints": {
      "Https": {
        "Url": "https://*:5271"
      }
    }
  }
}
```

The preceding configuration specifies an `Https` endpoint. The `Url` property is set to `https://*:5271`, which means the endpoint listens on all interfaces on port 5271. For more information, see [Configure endpoints for the ASP.NET Core Kestrel web server](#).

With the Kestrel endpoint configured, the project should remove any configured `applicationUrl` from the *launchSettings.json* file.

! Note

If the `applicationUrl` is present in the `launchSettings.json` file and the Kestrel endpoint is configured, the app host will throw an exception.

When you add a project resource, there's an overload that lets you specify that the Kestrel endpoint should be used instead of the `launchSettings.json` file:

C#

```
builder.AddProject<Projects.Networking_ApiService>(
    name: "apiservice",
    configure: static project =>
{
    project.ExcludeLaunchProfile = true;
    project.ExcludeKestrelEndpoints = false;
})
.WithHttpsEndpoint();
```

For more information, see [AddProject](#).

Ports and proxies

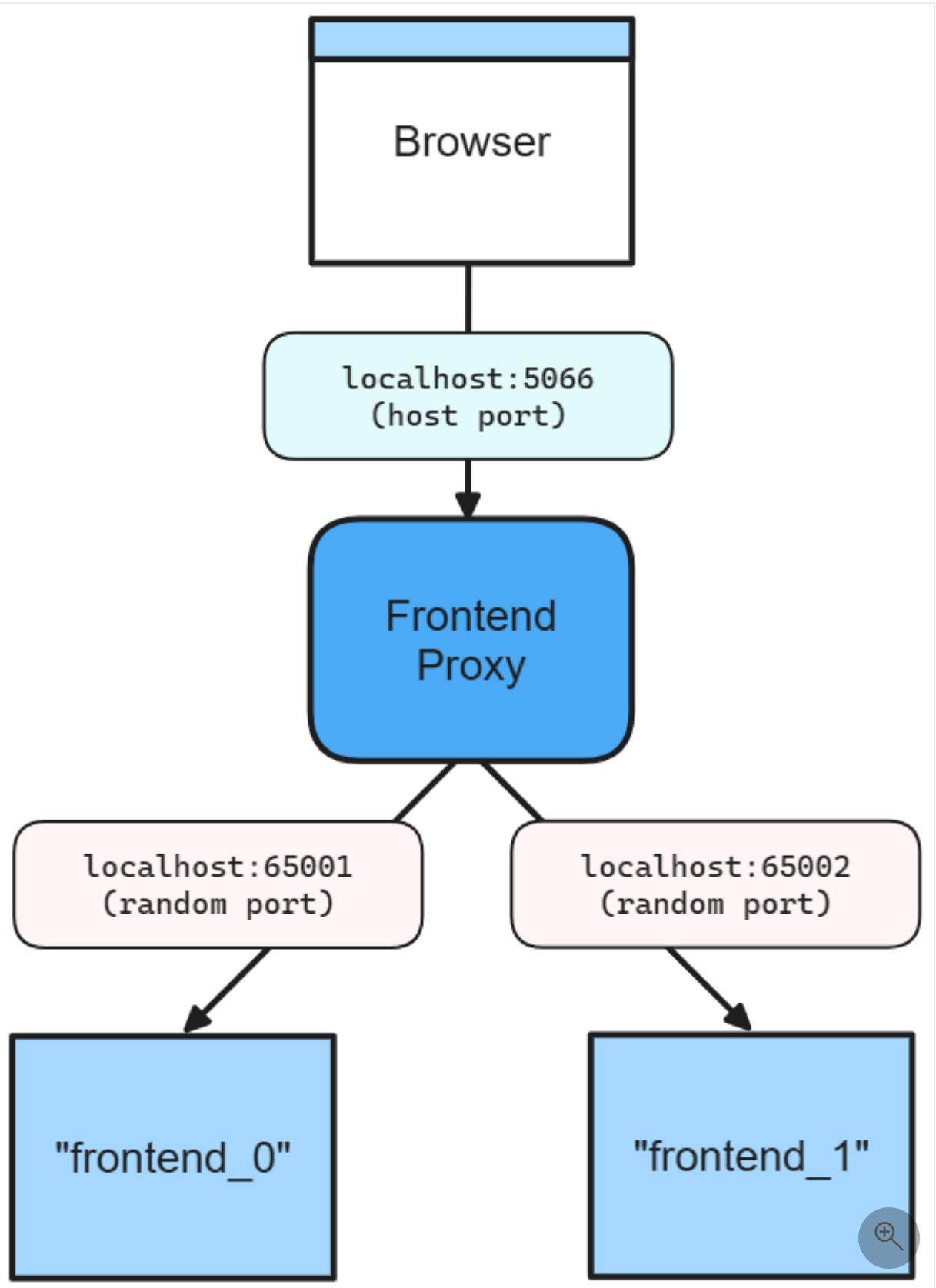
When defining a service binding, the host port is *always* given to the proxy that sits in front of the service. This allows single or multiple replicas of a service to behave similarly. Additionally, all resource dependencies that use the [WithReference](#) API rely of the proxy endpoint from the environment variable.

Consider the following method chain that calls [AddProject](#), [WithHttpEndpoint](#), and then [WithReplicas](#):

C#

```
builder.AddProject<Projects.Networking_Frontend>("frontend")
    .WithHttpEndpoint(port: 5066)
    .WithReplicas(2);
```

The preceding code results in the following networking diagram:



The preceding diagram depicts the following:

- A web browser as an entry point to the app.
- A host port of 5066.
- The frontend proxy sitting between the web browser and the frontend service replicas, listening on port 5066.

- The `frontend_0` frontend service replica listening on the randomly assigned port 65001.
- The `frontend_1` frontend service replica listening on the randomly assigned port 65002.

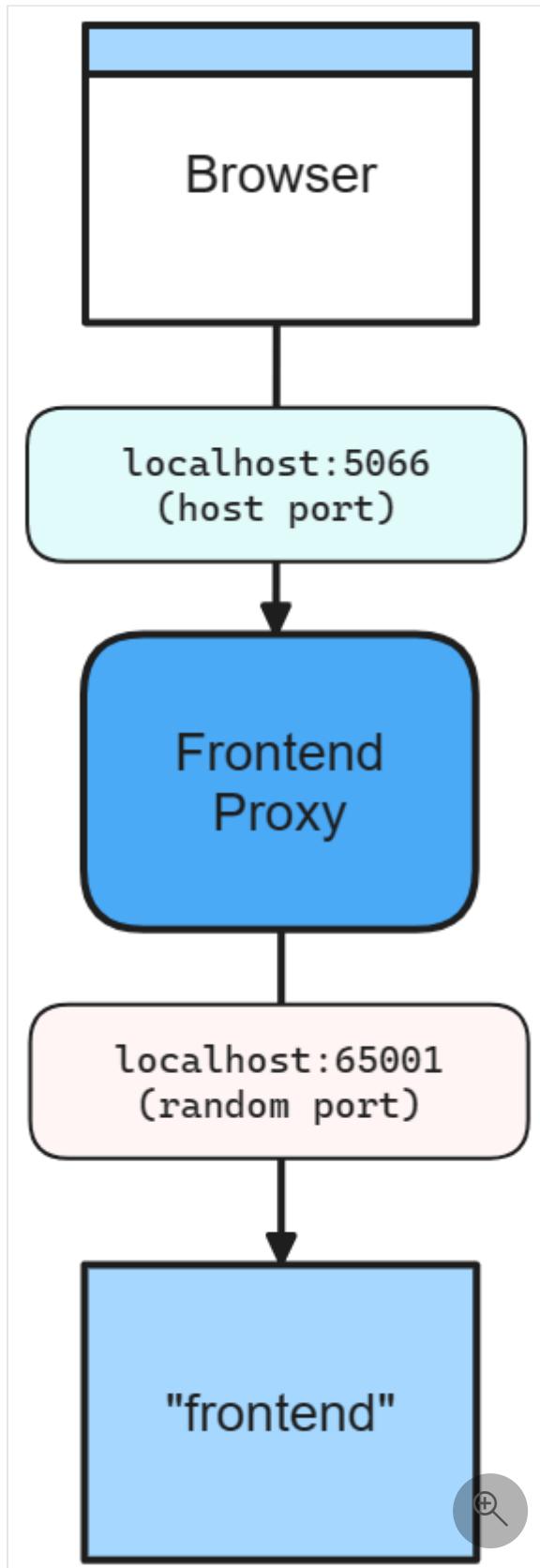
Without the call to `WithReplicas`, there's only one frontend service. The proxy still listens on port 5066, but the frontend service listens on a random port:

```
C#
```

```
builder.AddProject<Projects.Networking_Frontend>("frontend")
    .WithHttpEndpoint(port: 5066);
```

There are two ports defined:

- A host port of 5066.
- A random proxy port that the underlying service will be bound to.



The preceding diagram depicts the following:

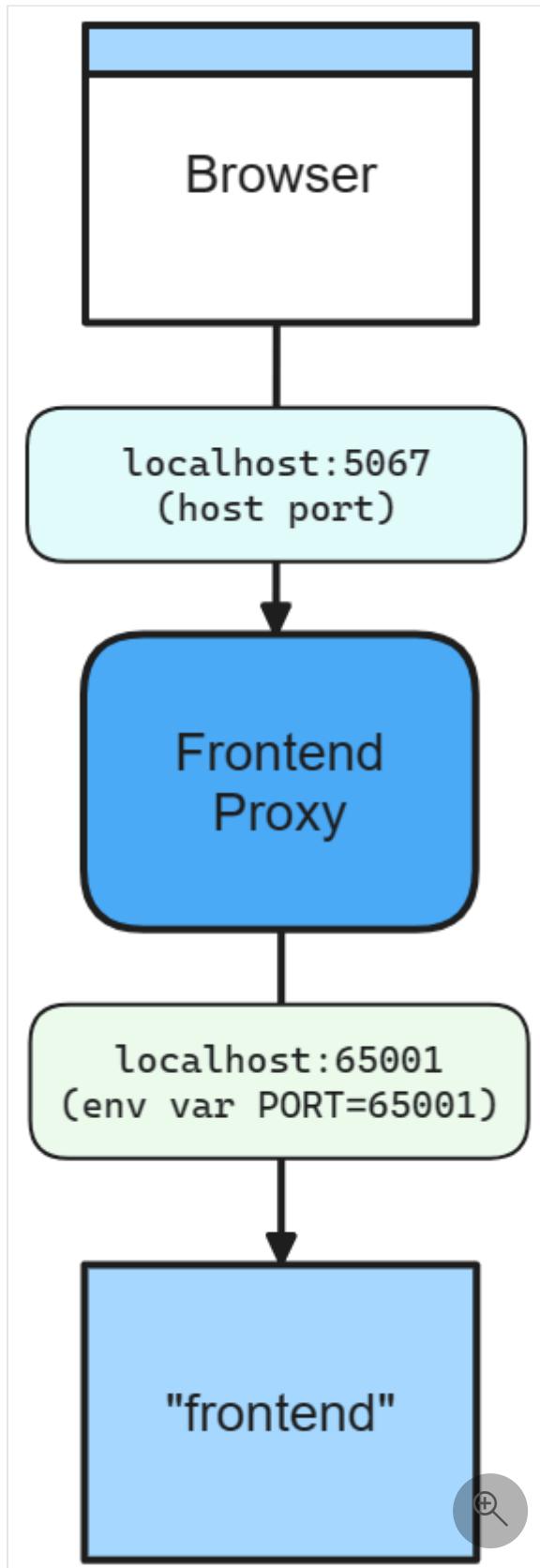
- A web browser as an entry point to the app.
- A host port of 5066.
- The frontend proxy sitting between the web browser and the frontend service, listening on port 5066.
- The frontend service listening on random port of 65001.

The underlying service is fed this port via `ASPNETCORE_URLS` for project resources. Other resources access to this port by specifying an environment variable on the service binding:

C#

```
builder.AddNpmApp("frontend", "../NodeFrontend", "watch")
    .WithHttpEndpoint(port: 5067, env: "PORT");
```

The previous code makes the random port available in the `PORt` environment variable. The app uses this port to listen to incoming connections from the proxy. Consider the following diagram:



The preceding diagram depicts the following:

- A web browser as an entry point to the app.
- A host port of 5067.
- The frontend proxy sitting between the web browser and the frontend service, listening on port 5067.
- The frontend service listening on an environment 65001.

💡 Tip

To avoid an endpoint being proxied, set the `IsProxied` property to `false` when calling the `WithEndpoint` extension method. For more information, see [Endpoint extensions: additional considerations](#).

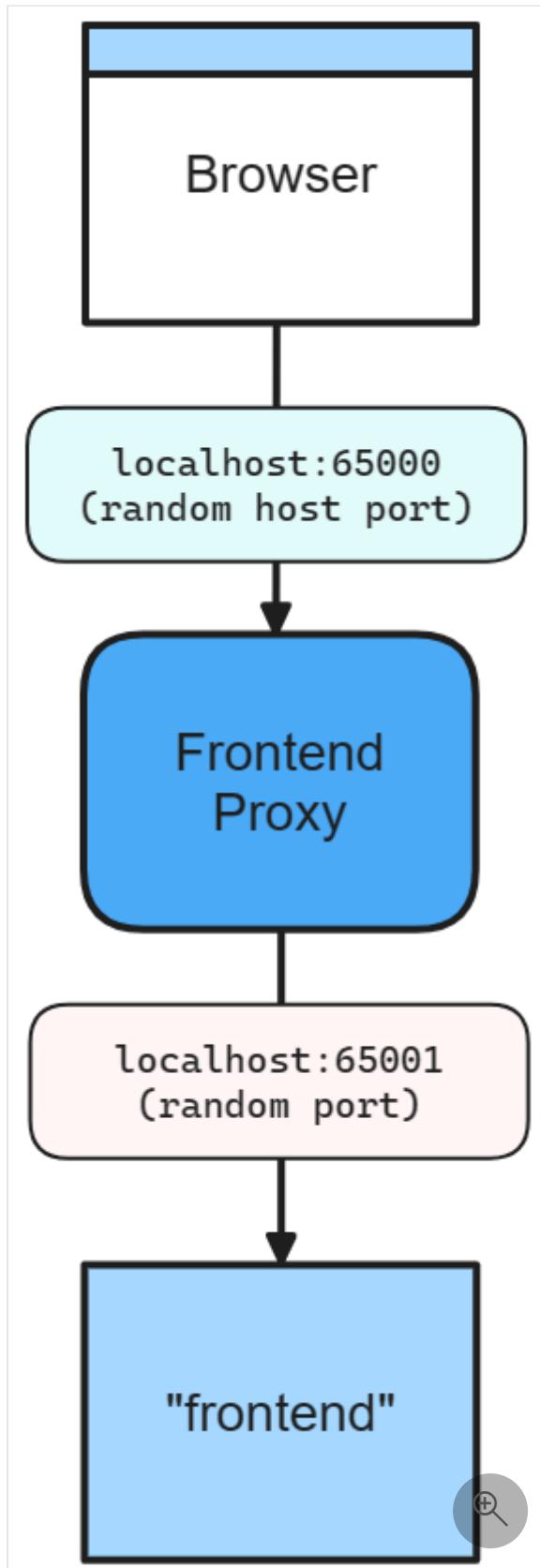
Omit the host port

When you omit the host port, .NET Aspire generates a random port for both host and service port. This is useful when you want to avoid port conflicts and don't care about the host or service port. Consider the following code:

C#

```
builder.AddProject<Projects.Networking_Frontend>("frontend")
    .WithHttpEndpoint();
```

In this scenario, both the host and service ports are random, as shown in the following diagram:



The preceding diagram depicts the following:

- A web browser as an entry point to the app.
- A random host port of 65000.
- The frontend proxy sitting between the web browser and the frontend service, listening on port 65000.
- The frontend service listening on a random port of 65001.

Container ports

When you add a container resource, .NET Aspire automatically assigns a random port to the container. To specify a container port, configure the container resource with the desired port:

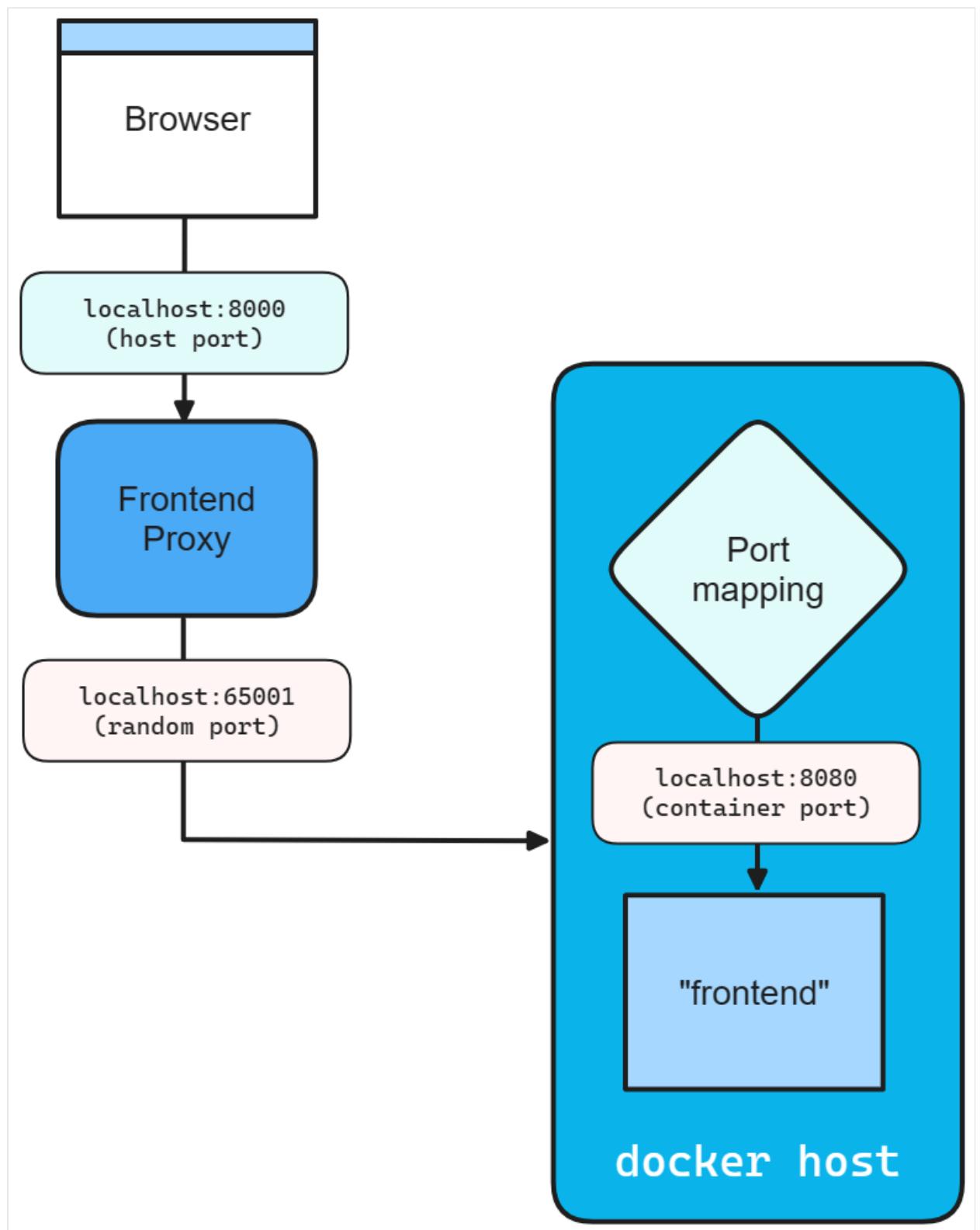
C#

```
builder.AddContainer("frontend", "mcr.microsoft.com/dotnet/samples",
"aspnetapp")
    .WithHttpEndpoint(port: 8000, targetPort: 8080);
```

The preceding code:

- Creates a container resource named `frontend`, from the `mcr.microsoft.com/dotnet/samples:aspnetapp` image.
- Exposes an `http` endpoint by binding the host to port 8000 and mapping it to the container's port 8080.

Consider the following diagram:



Endpoint extension methods

Any resource that implements the [IResourceWithEndpoints](#) interface can use the `WithEndpoint` extension methods. There are several overloads of this extension, allowing you to specify the scheme, container port, host port, environment variable name, and whether the endpoint is proxied.

There's also an overload that allows you to specify a delegate to configure the endpoint. This is useful when you need to configure the endpoint based on the environment or other factors. Consider the following code:

```
C#  
  
builder.AddProject<Projects.Networking_ApiService>("apiService")  
    .WithEndpoint(  
        endpointName: "admin",  
        callback: static endpoint =>  
    {  
        endpoint.Port = 17003;  
        endpoint.UriScheme = "http";  
        endpoint.Transport = "http";  
    });
```

The preceding code provides a callback delegate to configure the endpoint. The endpoint is named `admin` and configured to use the `http` scheme and transport, as well as the 17003 host port. The consumer references this endpoint by name, consider the following `AddHttpClient` call:

```
C#  
  
builder.Services.AddHttpClient<WeatherApiClient>(  
    client => client.BaseAddress = new Uri("http://_admin.apiservice"));
```

The `Uri` is constructed using the `admin` endpoint name prefixed with the `_` sentinel. This is a convention to indicate that the `admin` segment is the endpoint name belonging to the `apiservice` service. For more information, see [.NET Aspire service discovery](#).

Additional considerations

When calling the `WithEndpoint` extension method, the `callback` overload exposes the raw `EndpointAnnotation`, which allows the consumer to customize many aspects of the endpoint.

The `AllocatedEndpoint` property allows you to get or set the endpoint for a service. The `IsExternal` and `IsProxied` properties determine how the endpoint is managed and exposed: `IsExternal` decides if it should be publicly accessible, while `IsProxied` ensures DCP manages it, allowing for internal port differences and replication.

 Tip

If you're hosting an external executable that runs its own proxy and encounters port binding issues due to DCP already binding the port, try setting the `IsProxied` property to `false`. This prevents DCP from managing the proxy, allowing your executable to bind the port successfully.

The `Name` property identifies the service, whereas the `Port` and `TargetPort` properties specify the desired and listening ports, respectively.

For network communication, the `Protocol` property supports **TCP** and **UDP**, with potential for more in the future, and the `Transport` property indicates the transport protocol (**HTTP**, **HTTP2**, **HTTP3**). Lastly, if the service is URI-addressable, the `UriScheme` property provides the URI scheme for constructing the service URI.

For more information, see the available properties of the [EndpointAnnotation properties](#).

Endpoint filtering

All .NET Aspire project resource endpoints follow a set of default heuristics. Some endpoints are included in `ASPNETCORE_URLS` at runtime, some are published as `HTTP/HTTPS_PORTS`, and some configurations are resolved from Kestrel configuration. Regardless of the default behavior, you can filter the endpoints that are included in environment variables by using the `WithEndpointsInEnvironment` extension method:

C#

```
builder.AddProject<Projects.Networking_ApiService>("apiservice")
    .WithHttpsEndpoint() // Adds a default "https" endpoint
    .WithHttpsEndpoint(port: 19227, name: "admin")
    .WithEndpointsInEnvironment(
        filter: static endpoint =>
    {
        return endpoint.Name is not "admin";
    });
}
```

The preceding code adds a default HTTPS endpoint, as well as an `admin` endpoint on port 19227. However, the `admin` endpoint is excluded from the environment variables. This is useful when you want to expose an endpoint for internal use only.

External parameters

Article • 09/28/2024

Environments provide context for the application to run in. Parameters express the ability to ask for an external value when running the app. Parameters can be used to provide values to the app when running locally, or to prompt for values when deploying. They can be used to model a wide range of scenarios including secrets, connection strings, and other configuration values that might vary between environments.

Parameter values

Parameter values are read from the `Parameters` section of the app host's configuration and are used to provide values to the app while running locally. When you publish the app, if the value isn't available you're prompted to provide it.

Consider the following app host `Program.cs` example file:

```
C#  
  
var builder = DistributedApplication.CreateBuilder(args);  
  
// Add a parameter  
var value = builder.AddParameter("value");  
  
builder.AddProject<Projects ApiService>("api")  
    .WithEnvironment("EXAMPLE_VALUE", value);
```

Now consider the following app host configuration file `appsettings.json`:

```
JSON  
  
{  
  "Parameters": {  
    "value": "local-value"  
  }  
}
```

Parameters are represented in the manifest as a new primitive called `parameter.v0`:

```
JSON  
  
{  
  "resources": {  
    "value": {  
      "v0": "local-value"  
    }  
  }  
}
```

```
        "type": "parameter.v0",
        "value": "{value.inputs.value}",
        "inputs": {
            "value": {
                "type": "string"
            }
        }
    }
}
```

Secret values

Parameters can be used to model secrets. When a parameter is marked as a secret, it serves as a hint to the manifest that the value should be treated as a secret. When you publish the app, the value is prompted for and stored in a secure location. When you run the app locally, the value is read from the `Parameters` section of the app host configuration.

Consider the following app host `Program.cs` example file:

```
C#
```

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

// Add a secret parameter
var secret = builder.AddParameter("secret", secret: true);

builder.AddProject<Projects ApiService>("api")
    .WithEnvironment("SECRET", secret);

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

Now consider the following app host configuration file `appsettings.json`:

```
JSON
```

```
{ 
    "Parameters": {
        "secret": "local-secret"
    }
}
```

The manifest representation is as follows:

```
JSON
```

```
{  
  "resources": {  
    "value": {  
      "type": "parameter.v0",  
      "value": "{value.inputs.value}",  
      "inputs": {  
        "value": {  
          "type": "string",  
          "secret": true  
        }  
      }  
    }  
  }  
}
```

Connection string values

Parameters can be used to model connection strings. When you publish the app, the value is prompted for and stored in a secure location. When you run the app locally, the value is read from the `ConnectionStrings` section of the app host configuration.

⚠ Note

Connection strings are used to represent a wide range of connection information including database connections, message brokers, and other services. In .NET Aspire nomenclature, the term "connection string" is used to represent any kind of connection information.

Consider the following app host `Program.cs` example file:

C#

```
var builder = DistributedApplication.CreateBuilder(args);  
  
var redis = builder.AddConnectionString("redis");  
  
builder.AddProject<Projects.WebApplication>("api")  
    .WithReference(redis);  
  
builder.Build().Run();
```

Now consider the following app host configuration file `appsettings.json`:

JSON

```
{  
  "ConnectionStrings": {  
    "redis": "local-connection-string"  
  }  
}
```

For more information pertaining to connection strings and their representation in the deployment manifest, see[Connection string and binding references](#).

Parameter example

To express a parameter, consider the following example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);  
  
var db = builder.AddSqlServer("sql")  
    .PublishAsConnectionString()  
    .AddDatabase("db");  
  
var insertionRows = builder.AddParameter("insertionRows");  
  
builder.AddProject<Projects.Parameters_ApiService>("api")  
    .WithEnvironment("InsertionRows", insertionRows)  
    .WithReference(db);  
  
builder.Build().Run();
```

The following steps are performed:

- Adds a SQL Server resource named `sql` and publishes it as a connection string.
- Adds a database named `db`.
- Adds a parameter named `insertionRows`.
- Adds a project named `api` and associates it with the `Projects.Parameters_ApiService` project resource type-parameter.
- Passes the `insertionRows` parameter to the `api` project.
- References the `db` database.

The value for the `insertionRows` parameter is read from the `Parameters` section of the app host configuration file `appsettings.json`:

JSON

```
{  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "Microsoft.AspNetCore": "Warning",  
      "Aspire.Hosting.Dcp": "Warning"  
    }  
  },  
  "Parameters": {  
    "insertionRows": "1"  
  }  
}
```

The `Parameters_ApiService` project consumes the `insertionRows` parameter. Consider the `Program.cs` example file:

C#

```
using Microsoft.EntityFrameworkCore;  
  
var builder = WebApplication.CreateBuilder(args);  
  
int insertionRows = builder.Configuration.GetValue<int>("InsertionRows", 1);  
  
builder.AddServiceDefaults();  
  
builder.AddSqlServerDbContext<MyDbContext>("db");  
  
var app = builder.Build();  
  
app.MapGet("/", async (MyDbContext context) =>  
{  
  // You wouldn't normally do this on every call,  
  // but doing it here just to make this simple.  
  context.Database.EnsureCreated();  
  
  for (var i = 0; i < insertionRows; i++)  
  {  
    var entry = new Entry();  
    await context.Entries.AddAsync(entry);  
  }  
  
  await context.SaveChangesAsync();  
  
  var entries = await context.Entries.ToListAsync();  
  
  return new  
  {  
    totalEntries = entries.Count,  
    entries  
  };  
});
```

```
app.Run();
```

See also

- [.NET Aspire manifest format for deployment tool builders](#)
- [Tutorial: Connect an ASP.NET Core app to SQL Server using .NET Aspire and Entity Framework Core](#)

Persist .NET Aspire project data using volumes

Article • 04/29/2024

In this article, you learn how to configure .NET Aspire projects to persist data across app launches using volumes. A continuous set of data during local development is useful in many scenarios. Various .NET Aspire resource container types are able to leverage volume storage, such as PostgreSQL, Redis and Azure Storage.

When to use volumes

By default, every time you start and stop a .NET Aspire project, the app also creates and destroys the app resource containers. This setup creates problems when you want to persist data in a database or storage services between app launches for testing or debugging. For example, you may want to handle the following scenarios:

- Work with a continuous set of data in a database during an extended development session.
- Test or debug a changing set of files in an Azure Blob Storage emulator.
- Maintain cached data or messages in a Redis instance across app launches.

These goals can all be accomplished using volumes. With volumes, you decide which services retain data between launches of your .NET Aspire project.

Understand volumes

Volumes are the recommended way to persist data generated by containers and supported on both Windows and Linux. Volumes can store data from multiple containers at a time, offer high performance and are easy to back up or migrate. With .NET Aspire, you configure a volume for each resource container using the [ContainerResourceBuilderExtensions.WithBindMount](#) method, which accepts three parameters:

- **Source:** The source path of the volume, which is the physical location on the host.
- **Target:** The target path in the container of the data you want to persist.

For the remainder of this article, imagine that you're exploring a `Program` class in a .NET Aspire [app host project](#) that's already defined the distributed app builder bits:

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

// TODO:
// Consider various code snippets for configuring
// volumes here and persistent passwords.

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

The first code snippet to consider uses the `WithBindMount` API to configure a volume for a SQL Server resource. The following code demonstrates how to configure a volume for a SQL Server resource in a .NET Aspire app host project:

C#

```
var sql = builder.AddSqlServer("sql")
    .WithBindMount("VolumeMount.AppHost-sql-data",
    "/var/opt/mssql")
    .AddDatabase("sqldb");
```

In this example:

- `VolumeMount.AppHost-sql-data` sets where the volume will be stored on the host.
- `/var/opt/mssql` sets the path to the database files in the container.

All .NET Aspire container resources can utilize volume mounts, and some provide convenient APIs for adding named volumes derived from resources. Using the `WithDataVolume` as an example, the following code is functionally equivalent to the previous example but more succinct:

C#

```
var sql = builder.AddSqlServer("sql")
    .WithDataVolume()
    .AddDatabase("sqldb");
```

With the app host project being named `VolumeMount.AppHost`, the `WithDataVolume` method automatically creates a named volume as `VolumeMount.AppHost-sql-data` and is mounted to the `/var/opt/mssql` path in the SQL Server container. The naming convention is as follows:

- `{appHostProjectName}-{resourceName}-data`: The volume name is derived from the app host project name and the resource name.

Create a persistent password

Named volumes require a consistent password between app launches. .NET Aspire conveniently provides random password generation functionality. Consider the previous example once more, where a password is generated automatically:

```
C#  
  
var sql = builder.AddSqlServer("sql")  
    .WithDataVolume()  
    .AddDatabase("sqldb");
```

Since the `password` parameter isn't provided when calling `AddSqlServer`, .NET Aspire automatically generates a password for the SQL Server resource.

ⓘ Important

This isn't a persistent password! Instead, it changes every time the app host runs.

To create a *persistent* password, you must override the generated password. To do this, run the following command in your app host project directory to set a local password in your .NET user secrets:

```
.NET CLI
```

```
dotnet user-secrets set Parameters:sql-password <password>
```

The naming convention for these secrets is important to understand. The password is stored in configuration with the `Parameters:sql-password` key. The naming convention follows this pattern:

- `Parameters:{resourceName}-password`: In the case of the SQL Server resource (which was named "sql"), the password is stored in the configuration with the key `Parameters:sql-password`.

The same pattern applies to the other server-based resource types, such as those shown in the following table:

[] Expand table

Resource type	Hosting package	Example resource name	Override key
MySQL	Aspire.Hosting.MySql ↗	mysql	Parameters: mysql-password
Oracle	Aspire.Hosting.Oracle ↗	oracle	Parameters: oracle-password
PostgreSQL	Aspire.Hosting.PostgreSQL ↗	postgresql	Parameters: postgresql-password
RabbitMQ	Aspire.Hosting.RabbitMq ↗	rabbitmq	Parameters: rabbitmq-password
SQL Server	Aspire.Hosting.SqlServer ↗	sql	Parameters: sql-password

By overriding the generated password, you can ensure that the password remains consistent between app launches, thus creating a persistent password. An alternative approach is to use the `AddParameter` method to create a parameter that can be used as a password. The following code demonstrates how to create a persistent password for a SQL Server resource:

```
C#
var sqlPassword = builder.AddParameter("sql-password", secret: true);

var sql = builder.AddSqlServer("sql", password: sqlPassword)
    .WithDataVolume()
    .AddDatabase("sqldb");
```

The preceding code snippet demonstrates how to create a persistent password for a SQL Server resource. The `AddParameter` method is used to create a parameter named `sql-password` that's considered a secret. The `AddSqlServer` method is then called with the `password` parameter to set the password for the SQL Server resource. For more information, see [External parameters](#).

Next steps

You can apply the volume concepts in the preceding code to a variety of services, including seeding a database with data that will persist across app launches. Try combining these techniques with the resource implementations demonstrated in the following tutorials:

- [Tutorial: Connect an ASP.NET Core app to .NET Aspire storage integrations](#)

- Tutorial: Connect an ASP.NET Core app to SQL Server using .NET Aspire and Entity Framework Core
- .NET Aspire orchestration overview

.NET Aspire dashboard overview

Article • 05/30/2024

.NET Aspire project templates offer a sophisticated dashboard for comprehensive app monitoring and inspection. This dashboard allows you to closely track various aspects of your app, including logs, traces, and environment configurations, in real-time. It's purpose-built to enhance the local development experience, providing an insightful overview of your app's state and structure.

Using the dashboard with .NET Aspire projects

The dashboard is integrated into the .NET Aspire AppHost. During development the dashboard is automatically launched when you start the project. It's configured to display the .NET Aspire project's resources and telemetry.

Type	Name	State	Start time	Source	Endpoints	Logs	Details
Container	cache	Running	8:29:32 AM	docker.io/library/redis:7.2	tcp://localhost:54223	View	View
Project	apiservice	Running	8:29:32 AM	AspireSample.ApiService.csproj	https://localhost:7352/weatherforecast +1	View	View
Project	webfrontend	Running	8:29:32 AM	AspireSample.Web.csproj	https://localhost:7163 , http://localhost:5173	View	View

For more information about using the dashboard during .NET Aspire development, see [Explore dashboard features](#).

Standalone mode

The .NET Aspire dashboard is also shipped as a Docker image and can be used standalone, without the rest of .NET Aspire. The standalone dashboard provides a great UI for viewing telemetry and can be used by any application.

```
docker run --rm -it -p 18888:18888 -p 4317:18889 -d --name aspire-dashboard \
mcr.microsoft.com/dotnet/aspire-dashboard:8.0.0
```

The preceding Docker command:

- Starts a container from the `mcr.microsoft.com/dotnet/aspire-dashboard:8.0.0` image.
- The container publishes exposing two ports:
 - Maps the dashboard's OTLP port `18889` to the host's port `4317`. Port `4317` receives OpenTelemetry data from apps. Apps send data using [OpenTelemetry Protocol \(OTLP\)](#).
 - Maps the dashboard's port `18888` to the host's port `18888`. Port `18888` has the dashboard UI. Navigate to `http://localhost:18888` in the browser to view the dashboard.

For more information, see the [Standalone .NET Aspire dashboard](#).

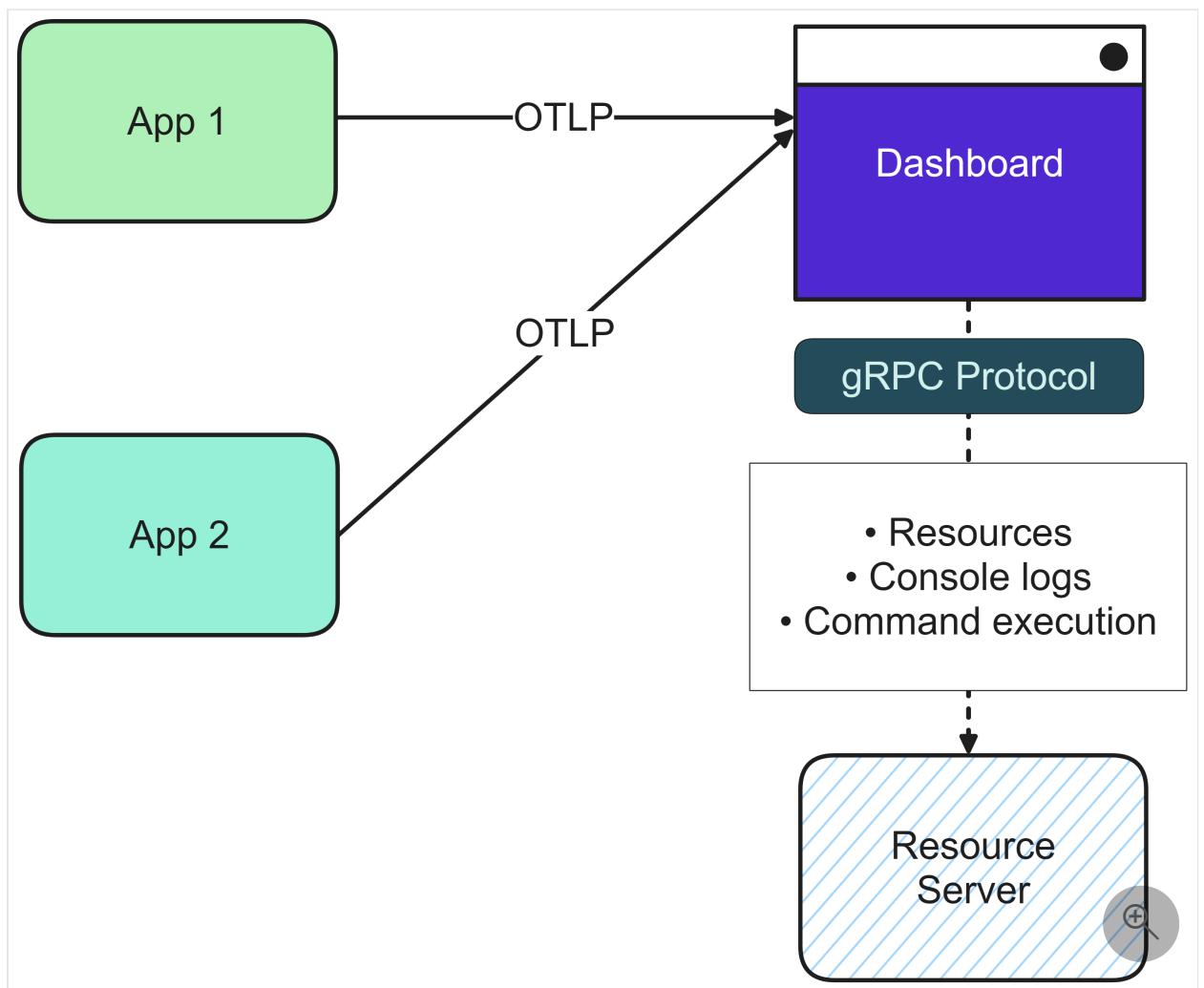
Configuration

The dashboard is configured when it starts up. Configuration includes frontend and OTLP addresses, the resource service endpoint, authentication, telemetry limits and more.

For more information, see [.NET Aspire dashboard configuration](#).

Architecture

The dashboard user experience is built with a variety of technologies. The frontend is built with [Microsoft's Fluent UI Blazor integration library](#). Each app communicates to the dashboard using the [OpenTelemetry Protocol \(OTLP\)](#). A resource server exists within this architecture to provide information about the app's resources, such as a resource listing, console logs, and command execution. The dashboard communicates using gRPC (specifically with the [Grpc.AspNetCore](#) NuGet package) to the resource server. Consider the following diagram that illustrates the architecture of the .NET Aspire dashboard:



Security

The .NET Aspire dashboard offers powerful insights to your apps. The UI displays information about resources, including their configuration, console logs and in-depth telemetry.

Data displayed in the dashboard can be sensitive. For example, configuration can include secrets in environment variables, and telemetry can include sensitive runtime data. Care should be taken to secure access to the dashboard.

For more information, see [.NET Aspire dashboard security considerations](#).

Next steps

[Explore the .NET Aspire dashboard](#)

Explore the .NET Aspire dashboard

Article • 09/12/2024

In the upcoming sections, you'll discover how to create a .NET Aspire project and embark on the following tasks:

- ✓ Investigate the dashboard's capabilities by using the app generated from the project template as explained in the [Quickstart: Build your first .NET Aspire project](#).
- ✓ Delve into the features of the .NET Aspire dashboard app.

The screenshots featured in this article showcase the dark theme. For more details on theme selection, refer to [Theme selection](#).

Dashboard authentication

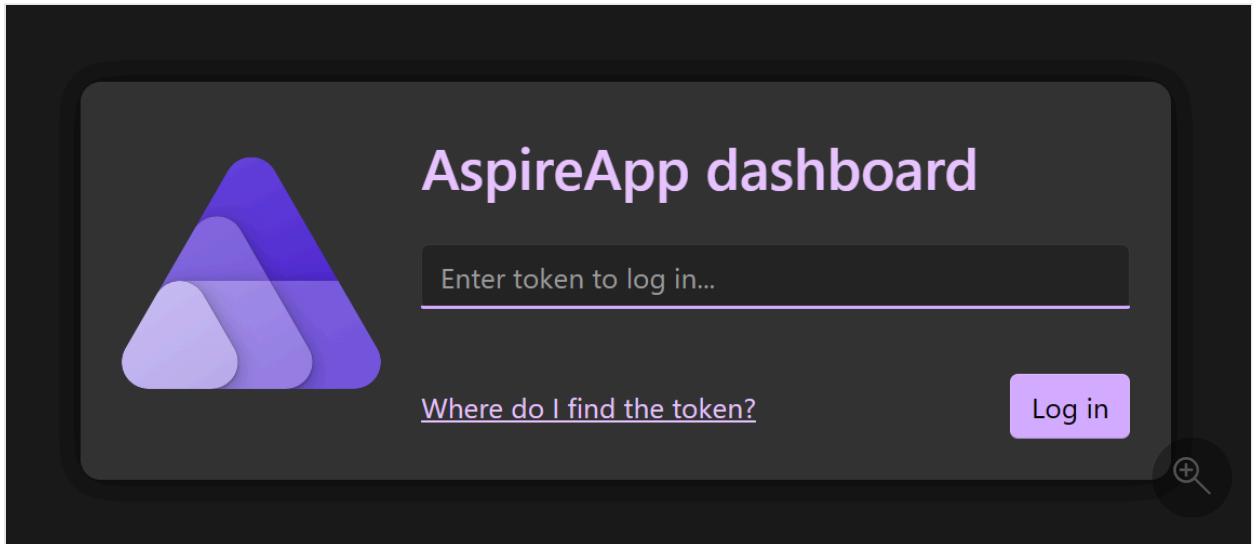
When you run a .NET Aspire app host, the orchestrator starts up all the app's dependent resources and then opens a browser window to the dashboard. The .NET Aspire dashboard requires token-based authentication for its users because it displays environment variables and other sensitive information.

When the dashboard is launched from Visual Studio or Visual Studio Code (with the [C# Dev Kit extension](#)), the browser is automatically logged in, and the dashboard opens directly. This is the typical developer [`F5`](#) experience, and the authentication login flow is automated by the .NET Aspire tooling.

However, if you start the app host from the command line, you're presented with the login page. The console window displays a URL that you can click on to open the dashboard in your browser.

```
info: Aspire.Hosting.DistributedApplication[0]
  Aspire version: 8.0.0+e215c528c07c7919c3ac30b35d92f4e51a60523b
info: Aspire.Hosting.DistributedApplication[0]
  Distributed application starting.
info: Aspire.Hosting.DistributedApplication[0]
  Application host directory is: D:\source\repos\docs-aspire\docs\get-started\snippets\quickstart\AspireSample\AspireSample.AppHost
info: Aspire.Hosting.DistributedApplication[0]
  Now listening on: https://localhost:17187
info: Aspire.Hosting.DistributedApplication[0]
  Login to the dashboard at https://localhost:17187/login?t=fd127643c01bd8d1afe61c7e1dbb340f
info: Aspire.Hosting.DistributedApplication[0]
  Distributed application started. Press Ctrl+C to shut down.
```

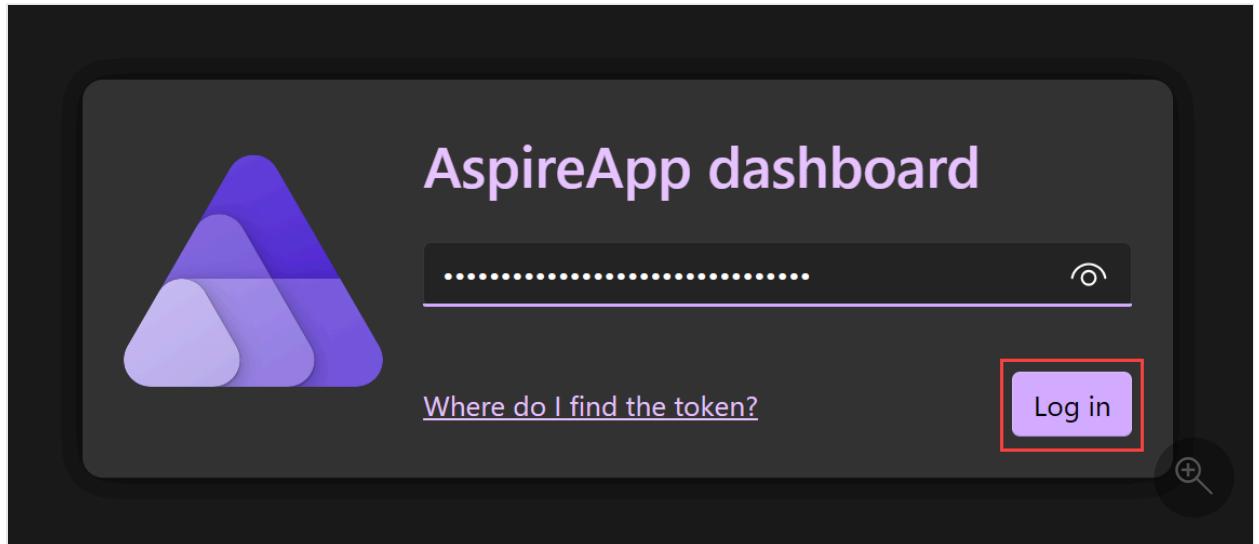
The URL contains a token query string (with the token value mapped to the [`t`](#) name part) that's used to *log in* to the dashboard. If your console supports it, you can hold the [`Ctrl`](#) key and then click the link to open the dashboard in your browser. This method is easier than copying the token from the console and pasting it into the login page. If you end up on the dashboard login page without either of the previously described methods, you can always return to the console to copy the token.



The login page accepts a token and provides helpful instructions on how to obtain the token, as shown in the following screenshot:

A screenshot of the AspireApp dashboard login page. A red arrow points from the text "Copy the highlighted token from the console to the login page and log in." down to the "Where do I find the token?" link. The link is underlined and located below the "Enter token to log in..." input field. The background of the main content area is highlighted with a purple rectangle. At the bottom of the page is a footer with a "More information" link and a magnifying glass icon with a plus sign.

After copying the token from the console and pasting it into the login page, select the **Log in** button.



The dashboard persists the token as a browser persistent cookie, which remains valid for three days. Persistent cookies have an expiration date and remain valid even after closing the browser. This means that users don't need to log in again if they close and reopen the browser. For more information, refer to the [Security considerations for running the .NET Aspire dashboard](#) documentation.

Resources page

The **Resources** page is the default home page of the .NET Aspire dashboard. This page lists all of the .NET projects, containers, and executables included in your .NET Aspire project. For example, the starter application includes two projects:

- **apiservice**: A back-end API for the .NET Aspire project built using Minimal APIs.
- **webfrontend**: The front-end UI for the .NET Aspire project built using Blazor.

The dashboard also provides essential details about each resource:

- **Type**: Displays whether the resource is a project, container, or executable.
- **Name**: The name of the resource.
- **State**: Displays whether or not the resource is currently running.
 - **Errors**: Within the **State** column, errors are displayed as a badge with the error count. It's useful to understand quickly what resources are reporting errors. Selecting the badge takes you to the [semantic logs](#) for that resource with the filter at an error level.
- **Start Time**: When the resource started running.
- **Source**: The location of the resource on the device.
- **Endpoints**: The URL(s) to reach the running resource directly.
- **Environment**: The environment variables that were loaded during startup.
- **Logs**: A link to the resource logs page.

Consider the following screenshot of the resources page:

Type	Name	State	Start time	Source	Endpoints	Logs	Details
Container	cache	Running	8:29:32 AM	docker.io/library/redis:7.2	tcp://localhost:54223	View	View
Project	apiservice	Running	8:29:32 AM	AspireSample.ApiService.csproj	https://localhost:7352/weatherforecast +1	View	View
Project	webfrontend	Running	8:29:32 AM	AspireSample.Web.csproj	https://localhost:7163 , http://localhost:5173	View	View

To view a *text visualizer* of certain columns, on hover you'll see a vertical ellipsis icon. Select the icon to display the available options:

- [Copy to clipboard](#)
- [Open in text visualizer](#)

Consider the following screenshot of the ellipsis menu options:

Type	Name	State	Start time	Source	Endpoints	Logs	Details	
Container	cache	Running	7:11:53 AM	docker.io/library/redis:7.4	tcp://localhost:64848	View	View	
Project	apiservice	Running	7:11:53 AM	AspireSample.ApiService.csproj	Copy to clipboard Open in text visualizer	https://localhost:7352/weatherforecast +1	View	View
Project	webfrontend	Running	7:11:53 AM	AspireSample.Web.csproj	https://localhost:7163 +1	View	View	

When you select the **Open in text visualizer** option, a modal dialog opens with the text displayed in a larger format. Consider the following screenshot of the text visualizer modal dialog:

Type	Name	State	Start time	Source	Endpoints	Logs	Details
Container	cache	Running	7:11:53 AM	docker.io/library/redis:7.4	tcp://localhost:64848	View	View
Project	apiservice	Running	7:11:53 AM	AspireSample.ApiService.csproj	https://localhost:7352/weatherforecast +1	View	View
Project	webfrontend	Running	7:11:53 AM	AspireSample.Web.csproj	https://localhost:7163 +1	View	View

Some values are formatted as JSON or XML. In these cases, the text visualizer enables the **Select format** dropdown to switch between the different formats.

You can obtain full details about each resource by selecting the **View** link in the **Details** column:

Type	Name	State	Start time	Source	Endpoints	Logs	Details
Container	cache	Running	8:29:32 AM	docker.io/library/redis:7.2	tcp://localhost:54223	View	View
Project	apiservice	Running	8:29:32 AM	AspireSample.ApiService.csproj	https://localhost:7352/weatherforecast +1	View	View
Project	webfrontend	Running	8:29:32 AM	AspireSample.Web.csproj	https://localhost:7163 , http://localhost:5173	View	View

Project: apiservice

[View logs](#) [Logs](#) [View](#) [Filter...](#)

Resource

Name	Value
Display name	apiservice
State	Running
Start time	5/16/2024 8:29:32 AM
Project path	D:\source\repos\docs-aspire\docs\get-started\snippets\quickstart\AspireSample\AspireSample.ApiService\AspireSample.Api...
Process ID	24864

Endpoints

Name	Value
http	http://localhost:5593/weatherforecast
http target port	http://localhost:54223
https	https://localhost:7352/weatherforecast
https target port	https://localhost:54224

The search bar in the upper right of the dashboard also provides the option to filter the list, which is useful for .NET Aspire projects with many resources. To select the types of resources that are displayed, drop down the arrow to the left of the filter textbox:

The screenshot shows the Aspire Sample application interface. On the left, there's a sidebar with icons for Console, Structured, Traces, and Metrics. The main area is titled "Resources" and contains a table with the following columns: Type, Name, State, Start time, Source, and Endpoints. A single row is present, representing a Container named "cache" which is "Running". The "Source" column shows "docker.io/library/redis:7.2" and the "Endpoints" column shows "tcp://localhost:54223". To the right of the table is a "Resource types" section with checkboxes for All, Project, Container (which is checked), and Executable. Below the table, a detailed view for the "cache" container is shown with sections for "Resource" (listing Name, Display name, State, Start time, Container image, Container ID, and Container ports) and "Endpoints" (listing Name, Value for tcp, and tcp target port). There's also a "Logs" tab with a "View logs" button and a "Filter..." search bar.

In this example, only containers are displayed in the list. For example, if you enable **Use Redis for caching** when creating a .NET Aspire project, you should see a Redis container listed:

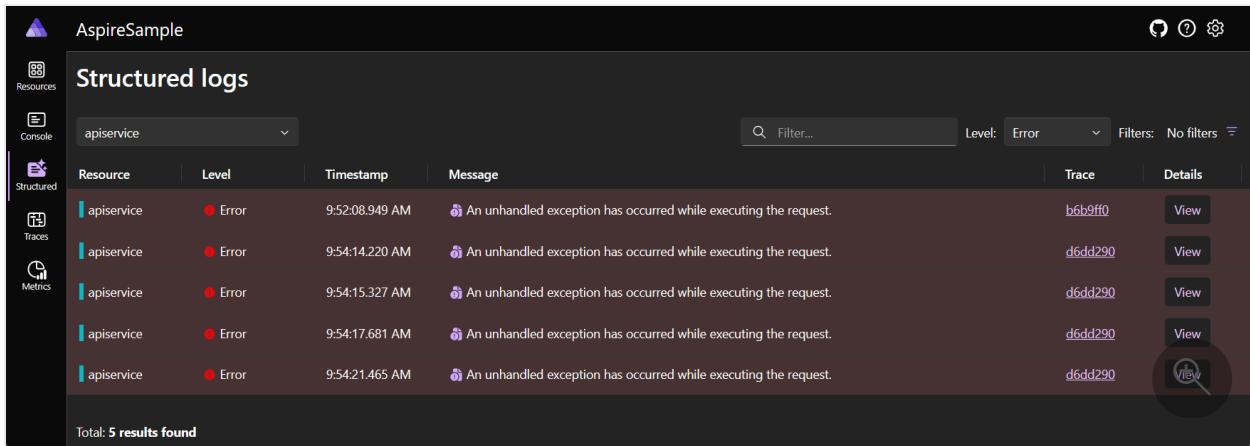
This screenshot is identical to the one above, showing a single running container named "cache". The interface includes the sidebar, the "Resources" table with the same data, and the "Resource types" section on the right. The "Logs" tab is visible at the bottom right of the main content area.

Executables are stand-alone processes. You can configure a .NET Aspire project to run a stand-alone executable during startup, though the default starter templates do not include any executables by default.

The following screenshot shows an example of a project that has errors:

This screenshot shows a "Resources" table with three entries. The first entry is a "Container" named "cache" which is "Running". The second entry is a "Project" named "apiservice" which is "Running" with a red error icon. The third entry is another "Project" named "webfrontend" which is "Running". The "Logs" and "Details" tabs are visible to the right of the table. The "apiservice" project row includes a link to its logs ("https://localhost:7352/weatherforecast") and a link to its details ("View"). The "webfrontend" project row includes a link to its logs ("https://localhost:7163, http://localhost:5173") and a link to its details ("View"). A magnifying glass icon is located at the bottom right of the table.

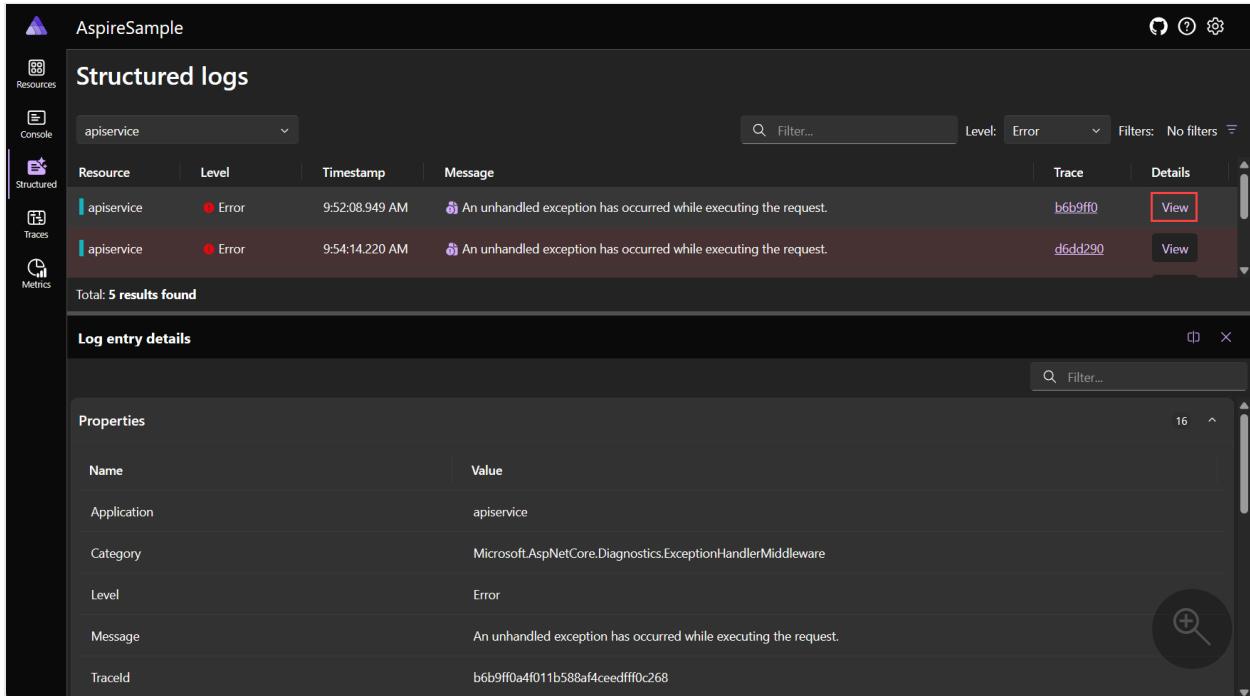
Selecting the error count badge navigates to the [Structured logs](#) page with a filter applied to show only the logs relevant to the resource:



The screenshot shows the 'Structured logs' page for the 'AspireSample' application. The left sidebar has tabs for 'Resources', 'Console', 'Structured' (which is selected), 'Traces', and 'Metrics'. The main area has a dropdown set to 'apiservice', a search bar with 'Filter...', and a 'Level' dropdown set to 'Error'. A 'Filters' section says 'No filters'. Below is a table with columns: Resource, Level, Timestamp, Message, Trace, and Details. There are five rows, each representing an error log entry for 'apiservice'. Each row has a 'View' button. At the bottom, it says 'Total: 5 results found'.

Resource	Level	Timestamp	Message	Trace	Details
apiservice	Error	9:52:08.949 AM	An unhandled exception has occurred while executing the request.	b6b9ff0	<button>View</button>
apiservice	Error	9:54:14.220 AM	An unhandled exception has occurred while executing the request.	d6dd290	<button>View</button>
apiservice	Error	9:54:15.327 AM	An unhandled exception has occurred while executing the request.	d6dd290	<button>View</button>
apiservice	Error	9:54:17.681 AM	An unhandled exception has occurred while executing the request.	d6dd290	<button>View</button>
apiservice	Error	9:54:21.465 AM	An unhandled exception has occurred while executing the request.	d6dd290	<button>View</button>

To see the log entry in detail for the error, select the **View** button to open a window below the list with the structured log entry details:



The screenshot shows the 'Structured logs' page for the 'AspireSample' application. The 'Structured' tab is selected. The main table shows two error log entries for 'apiservice'. The first entry's 'View' button is highlighted with a red border. Below the table, a 'Log entry details' panel is open for the first entry. It has a 'Properties' table with columns 'Name' and 'Value'. The properties listed are Application (apiservice), Category (Microsoft.AspNetCore.Diagnostics.ExceptionHandlerMiddleware), Level (Error), Message (An unhandled exception has occurred while executing the request.), and Traceld (b6b9ff0a4f011b588af4ceedfff0c268). A magnifying glass icon is in the bottom right corner of the details panel.

Name	Value
Application	apiservice
Category	Microsoft.AspNetCore.Diagnostics.ExceptionHandlerMiddleware
Level	Error
Message	An unhandled exception has occurred while executing the request.
Traceld	b6b9ff0a4f011b588af4ceedfff0c268

For more information and examples of Structured logs, see the [Structured logs page](#) section.

Note

The resources page isn't available if the dashboard is started without a configured resource service. The dashboard starts on the [Structured logs page](#) instead. This is the default experience when the dashboard is run in standalone mode without additional configuration.

For more information about configuring a resource service, see [Dashboard configuration](#).

Monitoring pages

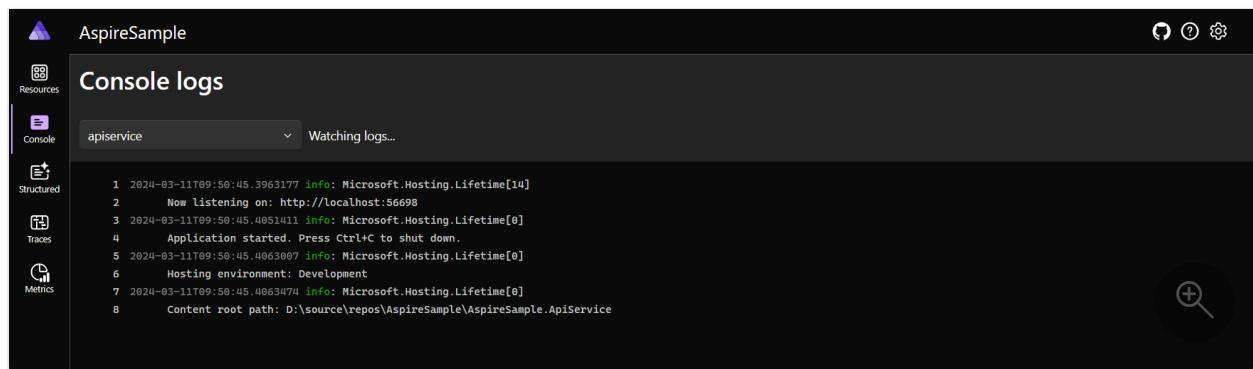
The .NET Aspire dashboard provides a variety of ways to view logs, traces, and metrics for your app. This information enables you to track the behavior and performance of your app and to diagnose any issues that arise.

Console logs page

The **Console logs** page displays text that each resource in your app has sent to standard output. Logs are a useful way to monitor the health of your app and diagnose issues. Logs are displayed differently depending on the source, such as project, container, or executable.

When you open the Console logs page, you must select a source in the **Select a resource** drop-down list.

If you select a project, the live logs are rendered with a stylized set of colors that correspond to the severity of the log; green for information as an example. Consider the following example screenshot of project logs with the `apiservice` project selected:



When errors occur, they're styled in the logs such that they're easy to identify. Consider the following example screenshot of project logs with errors:

```
1 2024-03-11T09:50:45.3963177 info: Microsoft.Hosting.Lifetime[14]
2     Now listening on: http://localhost:56698
3 2024-03-11T09:50:45.4051411 info: Microsoft.Hosting.Lifetime[0]
4     Application started. Press Ctrl+C to shut down.
5 2024-03-11T09:50:45.4063007 info: Microsoft.Hosting.Lifetime[0]
6     Hosting environment: Development
7 2024-03-11T09:50:45.4063474 info: Microsoft.Hosting.Lifetime[0]
8     Content root path: D:\source\repos\AspireSample\AspireSample.ApiService
9 2024-03-11T09:52:08.9343984 fail: Microsoft.AspNetCore.Diagnostics.ExceptionHandlerMiddleware[1]
10    An unhandled exception has occurred while executing the request.
11    System.Exception: Example server exception.
12        at Program.<>c__DisplayClass9_0.<>Main() in D:\source\repos\AspireSample\AspireSample.ApiService\Program.cs:line 24
13        at lambda_method(Closure, Object, HttpContext)
14        at Microsoft.AspNetCore.Diagnostics.ExceptionHandlerMiddlewareImpl.Invoke(HttpContext context)
15 2024-03-11T09:54:14.2206390 fail: Microsoft.AspNetCore.Diagnostics.ExceptionHandlerMiddleware[1]
16    An unhandled exception has occurred while executing the request.
17    System.Exception: Example server exception.
18        at Program.<>c__DisplayClass9_0.<>Main()
19        at lambda_method(Closure, Object, HttpContext)
20        at Microsoft.AspNetCore.Diagnostics.ExceptionHandlerMiddlewareImpl.Invoke(HttpContext context)
21 2024-03-11T09:54:15.3277408 fail: Microsoft.AspNetCore.Diagnostics.ExceptionHandlerMiddleware[1]
22    An unhandled exception has occurred while executing the request.
23    System.Exception: Example server exception.
24        at Program.<>c__DisplayClass9_0.<>Main()
25        at lambda_method(Closure, Object, HttpContext)
26        at Microsoft.AspNetCore.Diagnostics.ExceptionHandlerMiddlewareImpl.Invoke(HttpContext context)
27 2024-03-11T09:54:17.6809670 fail: Microsoft.AspNetCore.Diagnostics.ExceptionHandlerMiddleware[1]
28    An unhandled exception has occurred while executing the request.
29    System.Exception: Example server exception.
30        at Program.<>c__DisplayClass9_0.<>Main()
```

If you select a container or executable, formatting is different from a project but verbose behavior information is still available. Consider the following example screenshot of a container log with the `cache` container selected:

```
1 2024-03-11T09:50:44.4250070 1:C 11 Mar 2024 14:50:44.424 * 00800000000 Redis is starting 008000000000
2 2024-03-11T09:50:44.4256518 1:C 11 Mar 2024 14:50:44.424 * Redis version=7.2.4, bits=64, commit=00000000, modified=0, pid=1, just started
3 2024-03-11T09:50:44.4258563 1:C 11 Mar 2024 14:50:44.424 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
4 2024-03-11T09:50:44.4251527 1:M 11 Mar 2024 14:50:44.425 * monotonic clock: POSIX clock_gettime
5 2024-03-11T09:50:44.4254544 1:M 11 Mar 2024 14:50:44.425 * Running mode=standalone, port=6379.
6 2024-03-11T09:50:44.4259520 1:M 11 Mar 2024 14:50:44.425 * Server initialized
7 2024-03-11T09:50:44.4259663 1:M 11 Mar 2024 14:50:44.425 * Ready to accept connections tcp
```

Resource replicas

When project resources are replicated using the [WithReplicas](#) API, they're represented in the resource selector under a top-level named resource entry with an icon to indicate. Each replicated resource is listed under the top-level resource entry, with its corresponding unique name. Consider the following example screenshot of a replicated project resource:

The screenshot shows the 'Console logs' page of the TestShop application. On the left, there's a sidebar with icons for Resources, Console, Structured, Traces, and Metrics. The 'Console' icon is selected. In the main area, there's a dropdown labeled 'Resource' with '(None)' selected. Below it is a list of service names: apigateway, aspire-dashboard, basketcache-commander, basketcache, basketservice, catalogdb, catalogdbapp (FailedToStart), catalogservice (application), frontend, messaging, orderprocessor, and postgres-pgadmin. The 'catalogservice (application)' item is highlighted with a red box. At the bottom right of the main area is a magnifying glass icon.

The preceding screenshot shows the `catalogservice (application)` project with two replicas, `catalogservice-2bpj2qdq6k` and `catalogservice-6ljdin0hc0`. Each replica has its own set of logs that can be viewed by selecting the replica name.

Structured logs page

.NET Aspire automatically configures your projects with logging using OpenTelemetry. Navigate to the **Structured logs** page to view the semantic logs for your .NET Aspire project. [Semantic, or structured logging ↗](#) makes it easier to store and query log-events,

as the log-event message-template and message-parameters are preserved, instead of just transforming them into a formatted message. You'll notice a clean structure for the different logs displayed on the page using columns:

- **Resource:** The resource the log originated from.
- **Level:** The log level of the entry, such as information, warning, or error.
- **Timestamp:** The time that the log occurred.
- **Message:** The details of the log.
- **Trace:** A link to the relevant trace for the log, if applicable.
- **Details:** Additional details or metadata about the log entry.

Consider the following example screenshot of semantic logs:

The screenshot shows the 'Structured logs' page for the 'AspireSample' service. The left sidebar has tabs for 'Resources', 'Console', 'Structured', 'Traces', and 'Metrics'. The 'Structured' tab is selected. The main area has a title 'Structured logs' and a search/filter bar with dropdowns for 'Service' (set to '(All)'), 'Level' (set to '(All)'), and 'Filters' (set to 'No filters'). Below the search bar is a table with columns: Resource, Level, Timestamp, Message, Trace, and Details. The table contains 116 results, with the first few rows shown:

Resource	Level	Timestamp	Message	Trace	Details
apiservice	Information	9:50:45.402 AM	Now listening on: http://localhost:56698		View
apiservice	Information	9:50:45.405 AM	Application started. Press Ctrl+C to shut down.		View
apiservice	Information	9:50:45.406 AM	Hosting environment: Development		View
apiservice	Information	9:50:45.406 AM	Content root path: D:\source\repos\AspireSample\AspireSample.ApiService		View
webfrontend	Information	9:50:45.896 AM	Connecting (sync) on .NET 8.0.2 (StackExchange.Redis: v2.7.17.27058)		View
webfrontend	Information	9:50:45.909 AM	localhost:56697		View
webfrontend	Information	9:50:45.913 AM	localhost:56697/Interactive: Connecting...		View
webfrontend	Information	9:50:45.920 AM	localhost:56697: BeginConnectAsync		View
webfrontend	Information	9:50:45.928 AM	1 unique nodes specified (with tiebreaker)		View

Total: 116 results found

Filter structured logs

The structured logs page also provides a search bar to filter the logs by service, level, or message. You use the **Level** drop down to filter by log level. You can also filter by any log property by selecting the filter icon button, which will open the advanced filter dialog.

Consider the following screenshots showing the structured logs, filtered to display items with "Hosting" in the message text:

The screenshot shows the 'Structured logs' section of the AspireSample application. On the left, there's a sidebar with icons for Resources, Console, Structured logs (which is selected), Traces, and Metrics. The main area has a search bar and a 'Level' dropdown set to '(All)'. The table displays two rows:

Resource	Level	Timestamp	Message
apiservice	Information	9:50:45.406 AM	Hosting environment: Development
webfrontend	Information	9:50:46.061 AM	Hosting environment: Development

Total: 2 results found

Traces page

Navigate to the **Traces** page to view all of the traces for your app. .NET Aspire automatically configures tracing for the different projects in your app. Distributed tracing is a diagnostic technique that helps engineers localize failures and performance issues within applications, especially those that may be distributed across multiple machines or processes. For more information, see [.NET distributed tracing](#). This technique tracks requests through an application and correlates work done by different application integrations. Traces also help identify how long different stages of the request took to complete. The traces page displays the following information:

- **Timestamp:** When the trace completed.
- **Name:** The name of the trace, prefixed with the project name.
- **Spans:** The resources involved in the request.
- **Duration:** The time it took to complete the request. This column includes a radial icon that illustrates the duration of the request in comparison with the others in the list.

Traces					
(All)		Filter...			
Timestamp	Name	Spans	Duration	Details	
9:52:01.569 AM	webfrontend: GET / abd25c2	webfrontend (2)	80.74ms	View	
9:52:01.657 AM	webfrontend: GET d25ed39	webfrontend (1)	18.48ms	View	
9:52:01.657 AM	webfrontend: GET 57fd95c	webfrontend (1)	17.35ms	View	
9:52:01.658 AM	webfrontend: GET 09c3e0d	webfrontend (1)	14.4ms	View	
9:52:01.658 AM	webfrontend: GET b73a9b7	webfrontend (1)	1.01ms	View	
9:52:01.658 AM	webfrontend: GET b2e0c11	webfrontend (1)	10.1ms	View	
9:52:01.669 AM	webfrontend: GET c03efdd4	webfrontend (1)	17.69ms	View	
9:52:01.728 AM	webfrontend: GET 4417ab2	webfrontend (1)	10.31ms	View	
9:52:02.931 AM	webfrontend: GET /weather 5b56b13	webfrontend (5) apiservice (1)	434.4ms	View	

Total: 409 results found

Filter traces

The traces page also provides a search bar to filter the traces by name or span. Apply a filter, and notice the trace results are updated immediately. Consider the following screenshot of traces with a filter applied to `weather` and notice how the search term is highlighted in the results:

Traces					
(All)		Filter...			
Timestamp	Name	Spans	Duration	Details	
9:52:02.931 AM	webfrontend: GET /weather 5b56b13	webfrontend (5) apiservice (1)	434.4ms	View	
9:52:05.098 AM	webfrontend: GET /weather d177e30	webfrontend (3)	5.24ms	View	
9:52:05.121 AM	webfrontend: GET /weather 3d7a3d9	webfrontend (3)	3.18ms	View	
9:52:05.970 AM	webfrontend: GET /weather 692b1d9	webfrontend (3)	2.77ms	View	
9:52:05.995 AM	webfrontend: GET /weather d4f4137	webfrontend (3)	3.12ms	View	
9:52:06.706 AM	webfrontend: GET /weather d57b778	webfrontend (3)	2.65ms	View	
9:52:06.730 AM	webfrontend: GET /weather 5954e27	webfrontend (3)	2.75ms	View	
9:52:07.266 AM	webfrontend: GET /weather 2d69e96	webfrontend (3)	3.94ms	View	
9:52:07.288 AM	webfrontend: GET /weather 79de7a8	webfrontend (3)	2.47ms	View	

Total: 106 results found

Trace details

The trace details page contains various details pertinent to the request, including:

- **Trace Detail:** When the trace started.
- **Duration:** The time it took to complete the request.
- **Resources:** The number of resources involved in the request.
- **Depth:** The number of layers involved in the request.
- **Total Spans:** The total number of spans involved in the request.

Each span is represented as a row in the table, and contains a **Name**. Spans also display the error icon if an error occurred within that particular span of the trace. Spans that have a type of client/consumer, but don't have a span on the server, show an arrow icon and then the destination address. This represents a client call to a system outside of the .NET Aspire project. For example, an HTTP request an external web API, or a database call.

Within the trace details page, there's a **View Logs** button that takes you to the structured logs page with a filter applied to show only the logs relevant to the request. Consider an example screenshot depicting the structured logs page with a filter applied to show only the logs relevant to the trace:

Resource	Level	Timestamp	Message	Trace	Details
webfrontend	Information	9:53:51.068 AM	Start processing HTTP request GET http://apiservice/weatherforecast	2d0373b	View
webfrontend	Information	9:53:51.068 AM	Sending HTTP request GET http://localhost:5516/weatherforecast	2d0373b	View
webfrontend	Information	9:53:51.069 AM	Received HTTP response headers after 0.8431ms - 200	2d0373b	View
webfrontend	Information	9:53:51.069 AM	Execution attempt. Source: "standard//Standard-Retry", Operation Key: "", Result: '200', Handled: 'False', Att...	2d0373b	View
webfrontend	Information	9:53:51.069 AM	End processing HTTP request after 1.1828ms - 200	2d0373b	View

The structured logs page is discussed in more detail in the [Structured logs page](#) section.

Trace examples

Each trace has a color, which is generated to help differentiate between spans — one color for each resource. The colors are reflected in both the *traces page* and the *trace detail page*. When traces depict an arrow icon, those icons are colorized as well to match the span of the target trace. Consider the following example screenshot of traces:

AspireSample

Traces

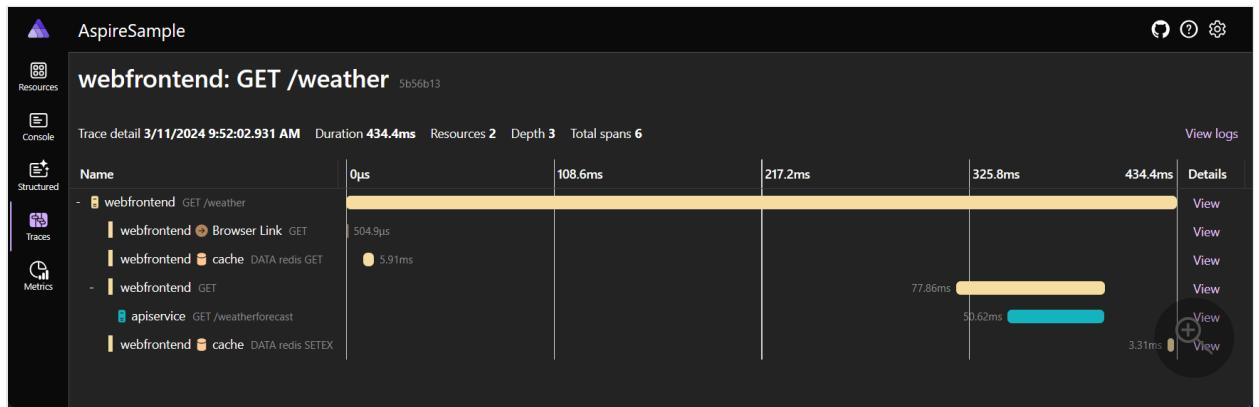
(All)

Filter...

Timestamp	Name	Spans	Duration	Details
9:52:01.569 AM	webfrontend: GET / abd25c2	webfrontend (2)	80.74ms	View
9:52:01.657 AM	webfrontend: GET d25ed39	webfrontend (1)	18.48ms	View
9:52:01.657 AM	webfrontend: GET 57fd95c	webfrontend (1)	17.35ms	View
9:52:01.658 AM	webfrontend: GET 09c3e0d	webfrontend (1)	14.4ms	View
9:52:01.658 AM	webfrontend: GET b73a9b7	webfrontend (1)	1.01ms	View
9:52:01.658 AM	webfrontend: GET b2e0c11	webfrontend (1)	10.1ms	View
9:52:01.669 AM	webfrontend: GET c03efdd4	webfrontend (1)	17.69ms	View
9:52:01.728 AM	webfrontend: GET 4417ab2	webfrontend (1)	10.31ms	View
9:52:02.931 AM	webfrontend: GET /weather 5b56b13	webfrontend (5) apiservice (1)	434.4ms	View

Total: 409 results found

You can also select the **View** button to navigate to a detailed view of the request and the duration of time it spent traveling through each application layer. Consider an example selection of a trace to view its details:



For each span in the trace, select **View** to see more details:

AspireSample

Resources Console Structured Traces Metrics

webfrontend: GET /weather 5b56b13

Trace detail 3/11/2024 9:52:02.931 AM Duration 434.4ms Resources 2 Depth 3 Total spans 6

Name	0µs	108.6ms	217.2ms	325.8ms	434.4ms	Details
- webfrontend GET /weather	0µs	108.6ms	217.2ms	325.8ms	434.4ms	View
webfrontend Browser Link GET	504.9µs					View
webfrontend cache DATA redis GET		5.91ms				View
- webfrontend GET			77.86ms			View
apiservice GET /weatherforecast				50.62ms		View
webfrontend cache DATA redis SETEX					3.31ms	View

webfrontend: DATA redis SETEX af9dba0

Service webfrontend Duration 3.31ms Start time 429.63ms

View logs Filter...

Span

Name	Value
SpanId	af9dba058ef6bbf6
Name	SETEX
Kind	Client
db.redis.database_index	0
db.redis.flags	DemandMaster
db.statement	SETEX
db.system	redis
net.peer.name	localhost
net.peer.port	56697

Scroll down in the span details pane to see full information. At the bottom of the span details pane, some span types, such as this call to a cache, show span event timings:

AspireSample

Resources Console Structured Traces Metrics

webfrontend: GET /weather 5b56b13

Trace detail 3/11/2024 9:52:02.931 AM Duration 434.4ms Resources 2 Depth 3 Total spans 6

Name	0µs	108.6ms	217.2ms	325.8ms	434.4ms	Details
- webfrontend GET /weather	0µs	108.6ms	217.2ms	325.8ms	434.4ms	View
webfrontend Browser Link GET	504.9µs					View
webfrontend cache DATA redis GET		5.91ms				View
- webfrontend GET			77.86ms			View
apiservice GET /weatherforecast				50.62ms		View
webfrontend cache DATA redis SETEX					3.31ms	View

webfrontend: DATA redis SETEX af9dba0

Service webfrontend Duration 3.31ms Start time 429.63ms

View logs Filter...

Span

Name	Value
service.name	webfrontend
service.instance.id	webfrontend-scch6f0
telemetry.sdk.name	opentelemetry
telemetry.sdk.language	dotnet
telemetry.sdk.version	1.7.0

Events

Time offset	Event
1.02ms	Enqueued
1.38ms	Sent
3.27ms	ResponseReceived

When errors are present, the page renders an error icon next to the trace name.

Consider an example screenshot of traces with errors:

The screenshot shows the 'Traces' section of the Aspire Sample application. It lists 417 results found. The columns include Timestamp, Name, Spans, Duration, and Details. Several spans are highlighted with error icons (red exclamation marks). One specific trace, 'webfrontend: GET /weather b6b9ff0', is expanded to show its detailed duration breakdown across multiple components: webfrontend, Browser Link, cache, apiservice, and another webfrontend component.

And the corresponding detailed view of the trace with errors:

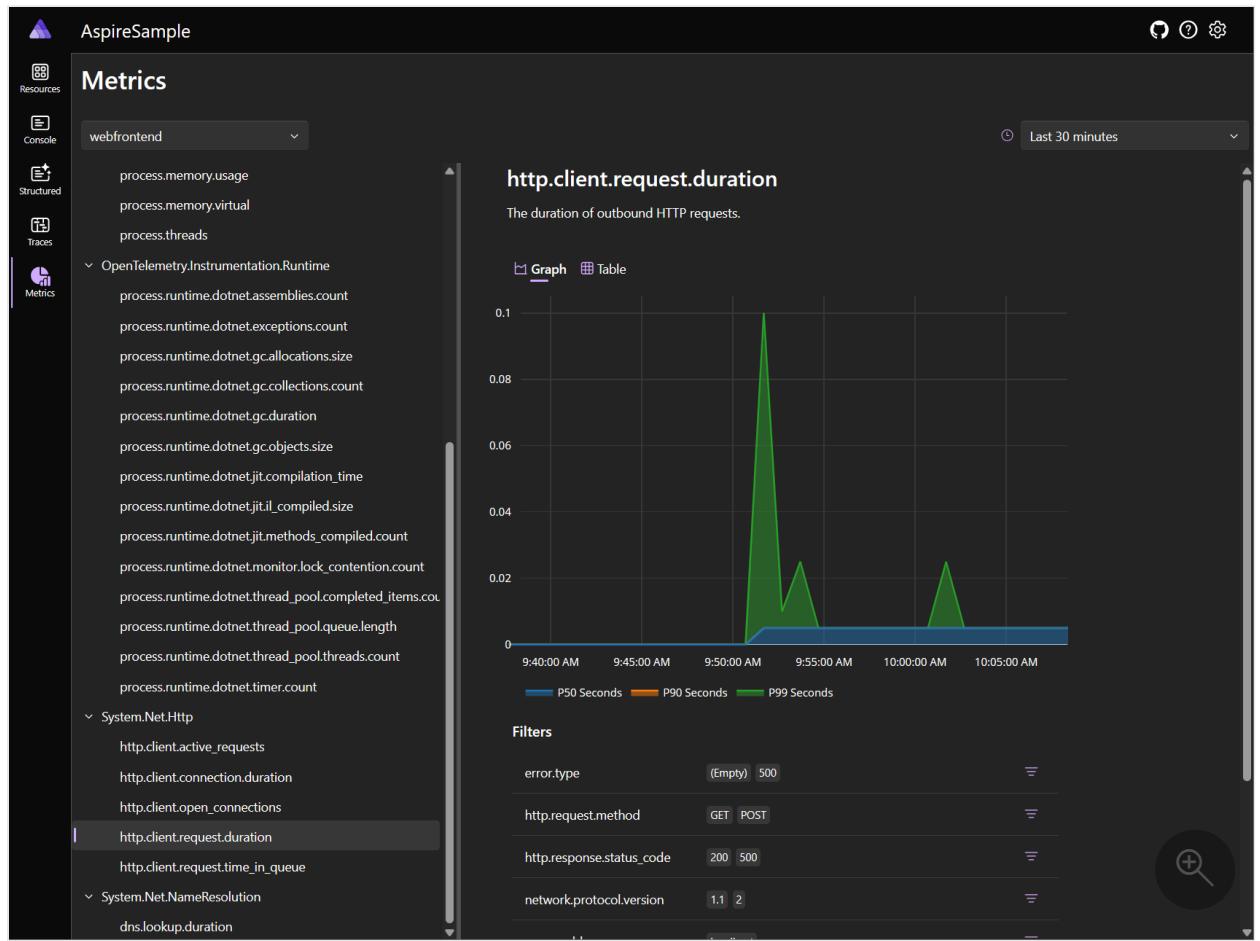
This screenshot provides a detailed view of the 'webfrontend: GET /weather' trace. It shows the total duration of 1.16s and the breakdown of spans. The trace involves multiple components: webfrontend, Browser Link, cache, apiservice, and another webfrontend component. The apiservice component is shown to have a duration of 24.69ms. The browser link component has a duration of 333.7μs. The cache component has a duration of 1.47ms. The other webfrontend component has a duration of 25.96ms. The total duration for the main trace is 289.62ms.

Metrics page

Navigate to the **Metrics** page to view the metrics for your app. .NET Aspire automatically configures metrics for the different projects in your app. Metrics are a way to measure the health of your application and can be used to monitor the performance of your app over time.

Each metric-publishing project in your app will have its own metrics. The metrics page displays a selection pane for each top-level meter and the corresponding instruments that you can select to view the metric.

Consider the following example screenshot of the metrics page, with the `webfrontend` project selected and the `System.Net.Http` meter's `http.client.request.duration` metric selected:



In addition to the metrics chart, the metrics page includes an option to view the data as a table instead. Consider the following screenshot of the metrics page with the table view selected:

AspireSample

Metrics

webfrontend

Last 30 minutes

http.client.request.duration

The duration of outbound HTTP requests.

Graph Table

Time	P50 Seconds	P90 Seconds	P99 Seconds
10:09:42 AM	0.005	0.005	0.005
10:08:42 AM	0.005	0.005	0.025
10:02:42 AM	0.005	0.005	0.005
10:01:42 AM	0.005	0.005	0.025
9:54:42 AM	0.005	0.005	0.005
9:53:42 AM	0.005	0.005	0.025
9:52:42 AM	0.005	0.005	0.01
9:51:42 AM	0.005	0.005	0.1

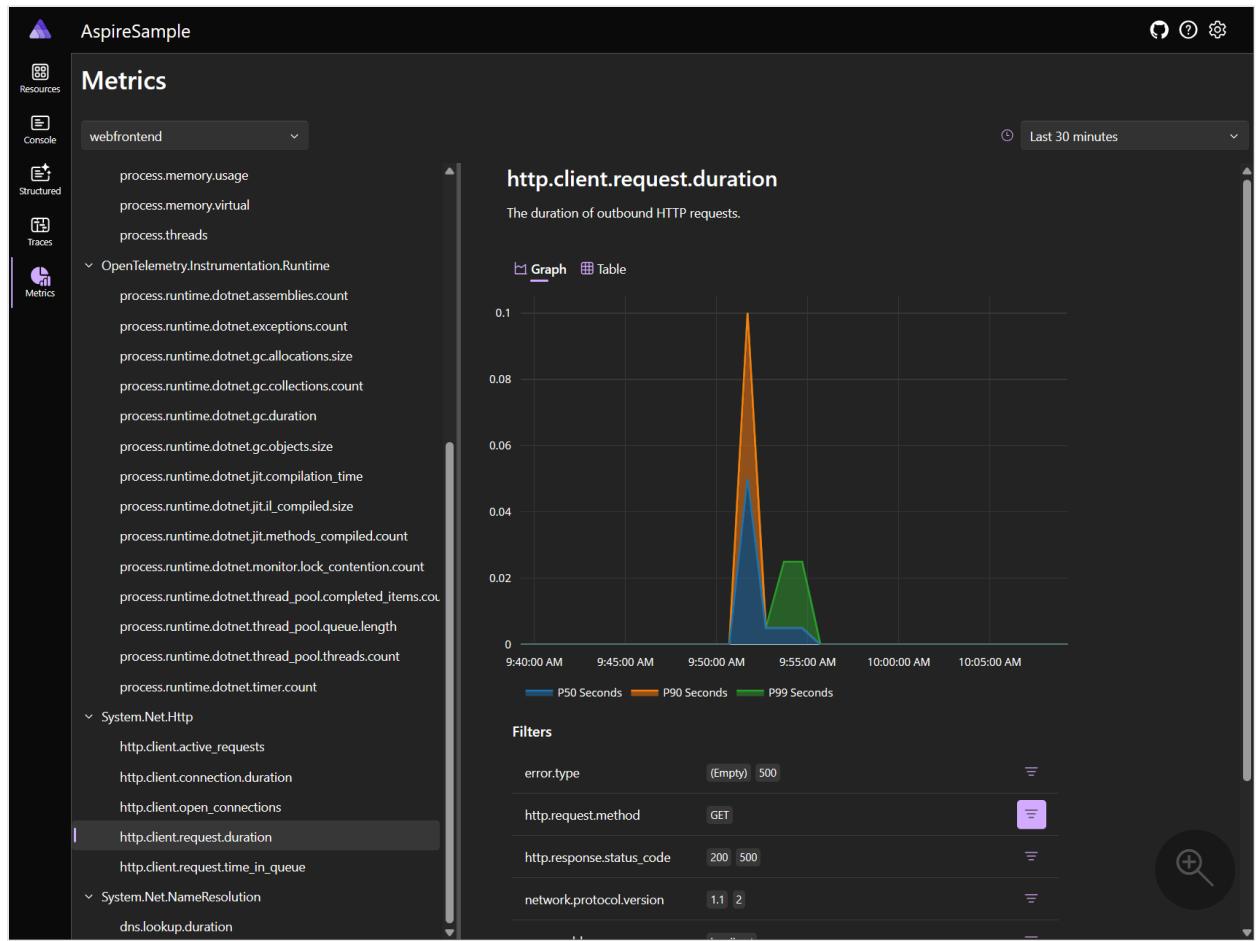
Only show value updates

Filters

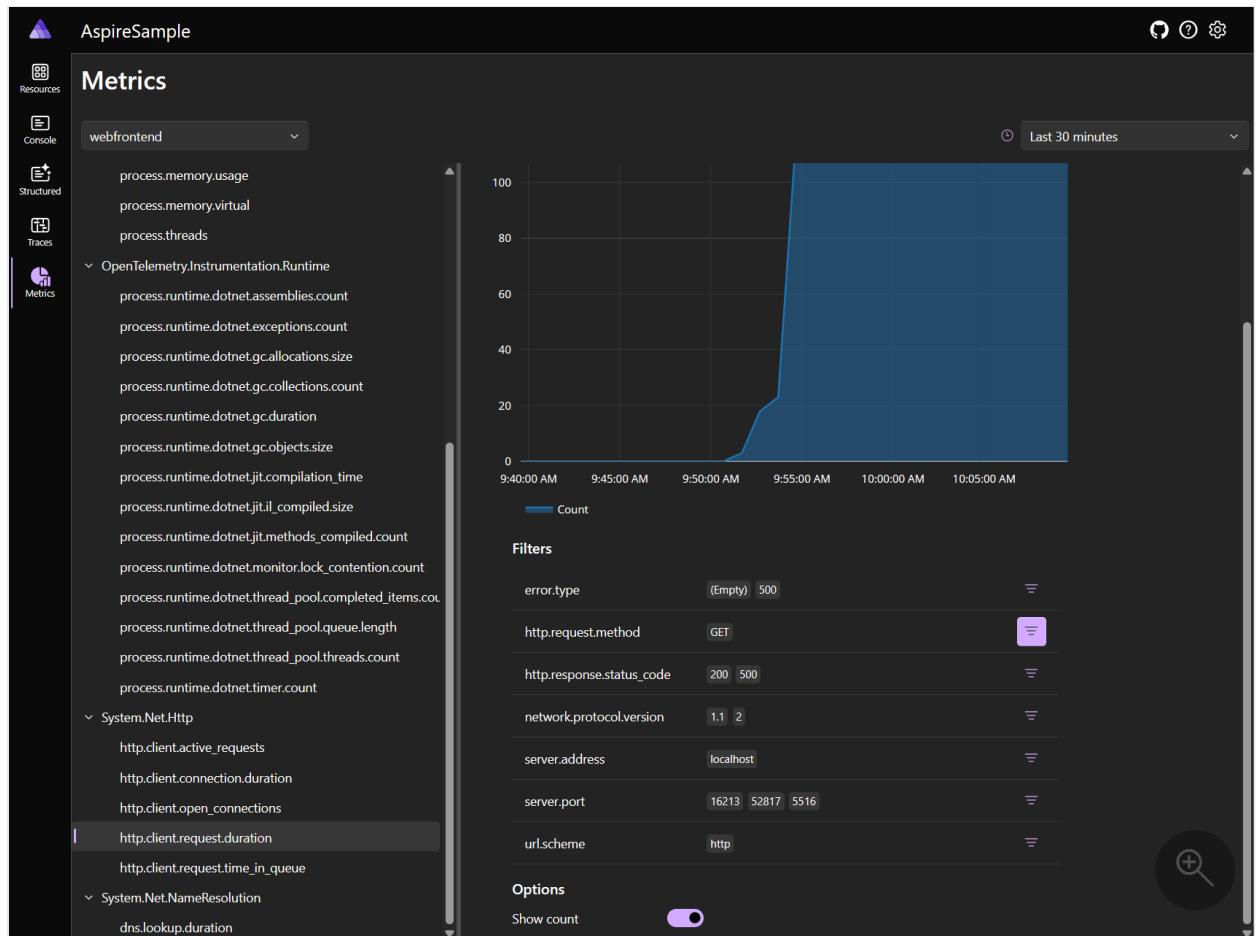
error.type	(Empty)	500
http.request.method	GET	POST
http.response.status_code	200	500
network.protocol.version	1.1	2

+

Under the chart, there is a list of filters you can apply to focus on the data that interests you. For example, in the following screenshot, the **http.request.method** field has been filtered to show only **GET** requests:



You can also choose to select the count of the displayed metric on the vertical access, instead of its values:



For more information about metrics, see [Built-in Metrics in .NET](#).

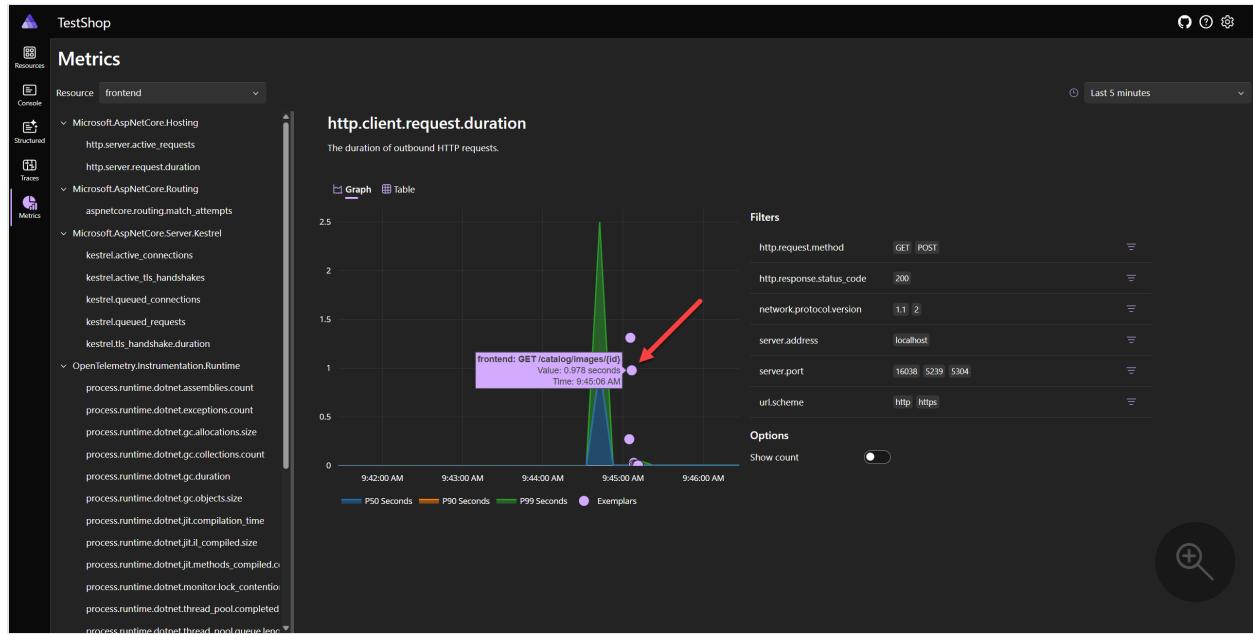
Exemplars

The .NET Aspire dashboard supports and displays OpenTelemetry *Exemplars*. An *exemplar* is a recorded value that contains additional associate context from traces with metric events, making them useful for linking trace signals with metrics.

An *exemplar* is made up of the following data points:

- `trace_id` and `span_id`: (Optional) The trace associated with the recording, identified by trace and span identifiers.
- `time_unix_nano`: The time of the observation, represented in Unix nanoseconds.
- `value`: The recorded value.
- `filtered_attributes`: A set of filtered attributes that provide additional context when the observation was made.

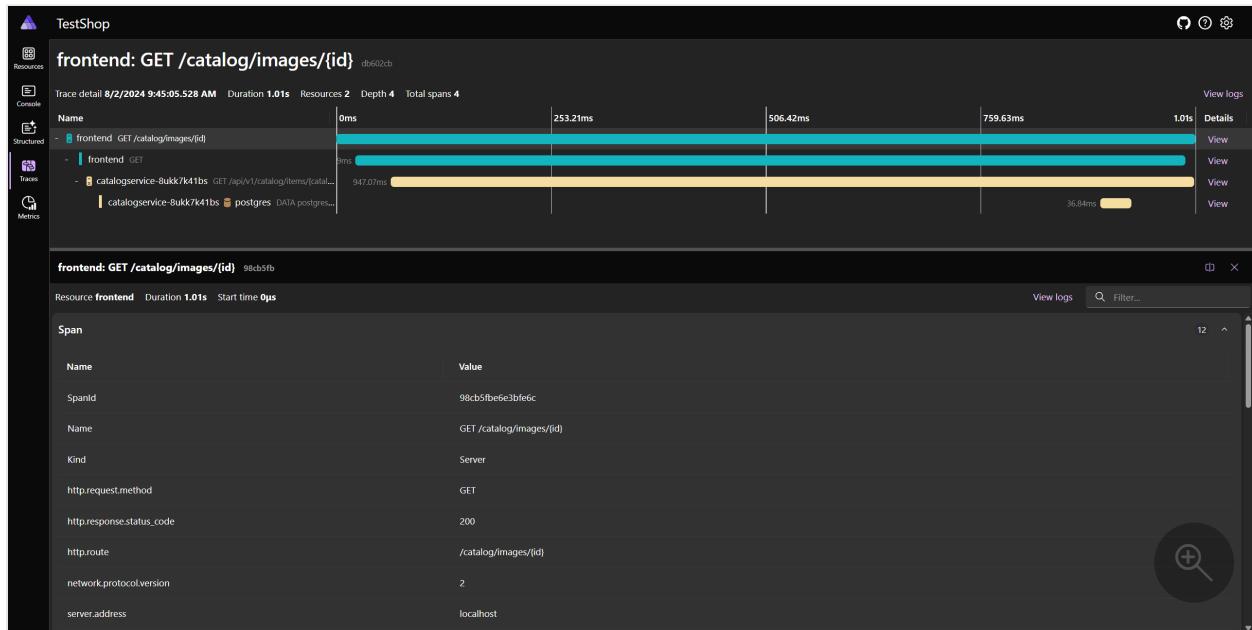
Exemplars are displayed in the metrics chart as a small round dot next to the data point. When you hover over the indicator, a tooltip displays the exemplar details as shown in the following screenshot:



The preceding screenshot shows the exemplar details for the `http.client.request.duration` metric. The exemplar details include the:

- Resource name.
- Operation performed, in this case an HTTP GET to the `/catalog/images/{id}`.
- Corresponding value and the time stamp.

Selecting the exemplar indicator opens the trace details page, where you can view the trace associated, for example consider the following screenshot:

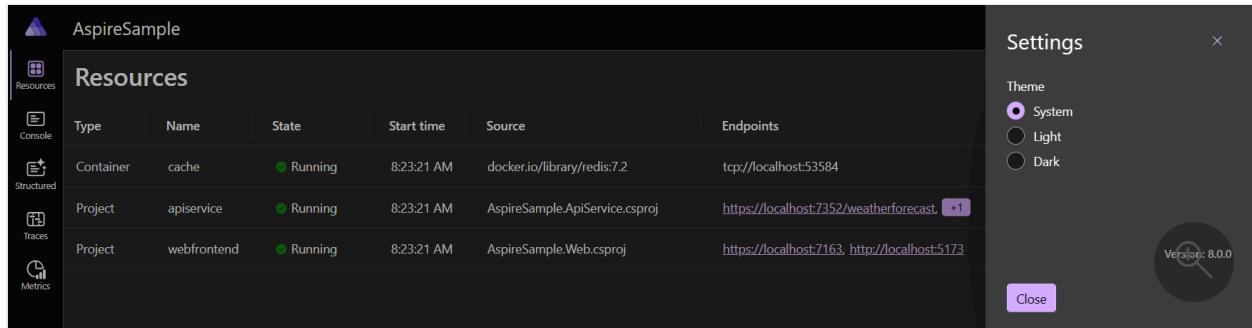


For more information, see [OpenTelemetry Docs: Exemplars](#).

Theme selection

By default, the theme is set to follow the System theme, which means the dashboard will use the same theme as your operating system. You can also select the **Light** or **Dark** theme to override the system theme. Theme selections are persisted.

The following screenshot shows the theme selection dialog, with the default System theme selected:



If you prefer the Light theme, you can select it from the theme selection dialog:

The screenshot shows the .NET Aspire dashboard interface. On the left, there's a sidebar with icons for Console, Structured Logs, Traces, and Metrics. The main area is titled "AspireSample" and has a "Resources" section. This section contains a table with three rows: a Container named "cache" running on docker.io/library/redis:7.2 with endpoint tcp://localhost:53584; a Project named "apiservice" running on AspireSample.ApiService.csproj with endpoint https://localhost:7352/weatherforecast; and a Project named "webfrontend" running on AspireSample.Web.csproj with endpoints https://localhost:7163 and http://localhost:5173. To the right of the main content is a "Settings" overlay window. It has a "Theme" section with three options: "System" (gray), "Light" (selected, indicated by a blue outline), and "Dark". Below that is a search bar with a magnifying glass icon and the placeholder text "Version: 8.0.0". At the bottom right of the overlay is a "Close" button.

Dashboard shortcuts

The .NET Aspire dashboard provides a variety of shortcuts to help you navigate the different parts of the dashboard. To display the keyboard shortcuts, press `Shift` + `?`. The following shortcuts are available:

Panels:

- `+`: Increase panel size.
- `-`: Decrease panel size.
- `Shift` + `r`: Reset panel size.
- `Shift` + `t`: Toggle panel orientation.
- `Shift` + `x`: Close panel.

Page navigation:

- `r`: Go to **Resources**.
- `c`: Go to **Console Logs**.
- `s`: Go to **Structured Logs**.
- `t`: Go to **Traces**.
- `m`: Go to **Metrics**.

Site-wide navigation:

- `?`: Got to **Help**.
- `Shift` + `s`: Go to **Settings**.

Next steps

[Standalone .NET Aspire dashboard](#)

Standalone .NET Aspire dashboard

Article • 05/30/2024

The [.NET Aspire dashboard](#) provides a great UI for viewing telemetry. The dashboard:

- Ships as a container image that can be used with any OpenTelemetry enabled app.
- Can be used standalone, without the rest of .NET Aspire.

The screenshot shows the 'Structured logs' section of the dashboard. At the top, there's a red banner with a warning icon and the text 'Telemetry endpoint is unsecured Untrusted apps can send telemetry to the dashboard.' with a 'More information' link. Below the banner, there's a search bar with a magnifying glass icon and a placeholder 'Filter...'. To the right of the search bar are dropdown menus for 'Level' (set to '(All)') and 'Filters' (set to 'No filters'). Below the search bar, there are four columns: 'Resource', 'Level', 'Timestamp', and 'Message'. On the far right, there are buttons for 'Trace' and 'Details'. In the center, it says 'No structured logs found'. At the bottom left, it says 'Total: 0 results found'. There are also some icons on the left side of the dashboard: a purple triangle icon for 'Aspire', a blue square icon for 'Structured', a green square icon for 'Traces', and a yellow square icon for 'Metrics'.

Start the dashboard

The dashboard is started using the Docker command line.

```
Bash
docker run --rm -it \
-p 18888:18888 \
-p 4317:18889 -d \
--name aspire-dashboard \
mcr.microsoft.com/dotnet/aspire-dashboard:8.1.0
```

The preceding Docker command:

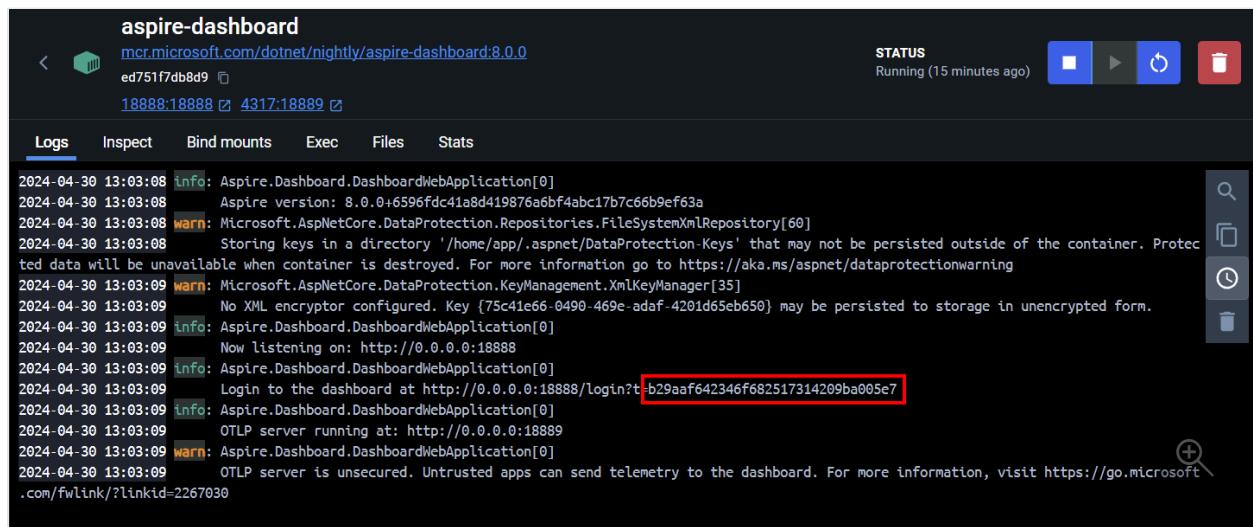
- Starts a container from the `mcr.microsoft.com/dotnet/aspire-dashboard:8.1.0` image.
- The container expose two ports:
 - Mapping the dashboard's OTLP port `18889` to the host's port `4317`. Port `4317` receives OpenTelemetry data from apps. Apps send data using [OpenTelemetry Protocol \(OTLP\)](#).
 - Mapping the dashboard's port `18888` to the host's port `18888`. Port `18888` has the dashboard UI. Navigate to `http://localhost:18888` in the browser to view

the dashboard.

Login to the dashboard

Data displayed in the dashboard can be sensitive. By default, the dashboard is secured with authentication that requires a token to login.

When the dashboard is run from a standalone container, the login token is printed to the container logs. After copying the highlighted token into the login page, select the *Login* button.



The screenshot shows the Docker container logs for 'aspire-dashboard'. The logs output by the application include:

```
2024-04-30 13:03:08 [Info]: Aspire.Dashboard.DashboardWebApplication[0]
2024-04-30 13:03:08     Aspire version: 8.0.0+6596fdc41a8d419876a6bf4abc17b7c66b9ef63a
2024-04-30 13:03:08 [Warn]: Microsoft.AspNetCore.DataProtection.Repositories.FileSystemXmlRepository[60]
2024-04-30 13:03:08     Storing keys in a directory '/home/app/.aspnet/DataProtection-Keys' that may not be persisted outside of the container. Protected data will be unavailable when container is destroyed. For more information go to https://aka.ms/aspnet/dataprotectionwarning
2024-04-30 13:03:09 [Warn]: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[35]
2024-04-30 13:03:09     No XML encryptor configured. Key {75c41e66-0490-469e-adaf-4201d65ceb650} may be persisted to storage in unencrypted form.
2024-04-30 13:03:09 [Info]: Aspire.Dashboard.DashboardWebApplication[0]
2024-04-30 13:03:09     Now listening on: http://0.0.0.0:18888
2024-04-30 13:03:09 [Info]: Aspire.Dashboard.DashboardWebApplication[0]
2024-04-30 13:03:09     Login to the dashboard at http://0.0.0.0:18888/login?t=b29aaaf642346f682517314209ba005e7
2024-04-30 13:03:09 [Info]: Aspire.Dashboard.DashboardWebApplication[0]
2024-04-30 13:03:09     OTLP server running at: http://0.0.0.0:18889
2024-04-30 13:03:09 [Warn]: Aspire.Dashboard.DashboardWebApplication[0]
2024-04-30 13:03:09     OTLP server is unsecured. Untrusted apps can send telemetry to the dashboard. For more information, visit https://go.microsoft.com/fwlink/?linkid=2267030
```

Tip

To avoid the login, you can disable the authentication requirement by setting the `DOTNET_DASHBOARD_UNSECURED_ALLOW_ANONYMOUS` environment variable to `true`.

Additional configuration is available, see [Dashboard configuration](#).

For more information about logging into the dashboard, see [Dashboard authentication](#).

Explore the dashboard

The dashboard offers a UI for viewing telemetry. Refer to the documentation to explore the telemetry functionality:

- [Structured logs page](#)
- [Traces page](#)
- [Metrics page](#)

Although there is no restriction on where the dashboard is run, the dashboard is designed as a development and short-term diagnostic tool. The dashboard persists

telemetry in-memory which creates some limitations:

- Telemetry is automatically removed if [telemetry limits](#) are exceeded.
- No telemetry is persisted when the dashboard is restarted.

The dashboard also has functionality for viewing .NET Aspire resources. The dashboard resource features are disabled when it is run in standalone mode. To enable the resources UI, [add configuration for a resource service](#).

Send telemetry to the dashboard

Apps send telemetry to the dashboard using [OpenTelemetry Protocol \(OTLP\)](#). The dashboard must expose a port for receiving OpenTelemetry data, and apps are configured to send data to that address.

A Docker command was shown earlier to [start the dashboard](#). It configured the container to receive OpenTelemetry data on port `4317`. The OTLP endpoint's full address is `http://localhost:4317`.

Configure OpenTelemetry SDK

Apps collect and send telemetry using [their language's OpenTelemetry SDK](#).

Important OpenTelemetry SDK options to configure:

- OTLP endpoint, which should match the dashboard's configuration, for example, `http://localhost:4317`.
- OTLP protocol, with the dashboard currently supporting only the [OTLP/gRPC protocol](#). Configure applications to use the `grpc` protocol.

To configure applications:

- Use the OpenTelemetry SDK APIs within the application, or
- Start the app with [known environment variables](#):
 - `OTEL_EXPORTER_OTLP_PROTOCOL` with a value of `grpc`.
 - `OTEL_EXPORTER_OTLP_ENDPOINT` with a value of `http://localhost:4317`.

Sample

For a sample of using the standalone dashboard, see the [Standalone .NET Aspire dashboard sample app](#).

Next steps

Configure the .NET Aspire dashboard

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Tutorial: Use the .NET Aspire dashboard with Python apps

Article • 09/04/2024

The [.NET Aspire dashboard](#) provides a great user experience for viewing telemetry, and is available as a standalone container image that can be used with any OpenTelemetry-enabled app. In this article, you'll learn how to:

- ✓ Start the .NET Aspire dashboard in standalone mode.
- ✓ Use the .NET Aspire dashboard with a Python app.

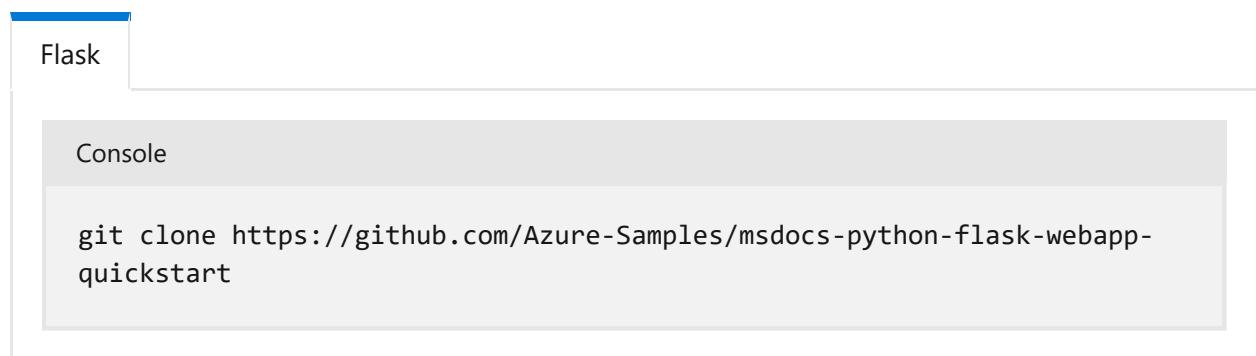
Prerequisites

To complete this tutorial, you need the following:

- [Docker ↗](#) or [Podman ↗](#).
 - You can use an alternative container runtime, but the commands in this article are for Docker.
- [Python 3.9 or higher ↗](#) locally installed.
- A sample application.

Sample application

This tutorial can be completed using either Flask, Django, or FastAPI. A sample application in each framework is provided to help you follow along with this tutorial. Download or clone the sample application to your local workstation.



Flask

Console

```
git clone https://github.com/Azure-Samples/msdocs-python-flask-webapp-quickstart
```

To run the application locally:



Flask

1. Go to the application folder:

```
Console
```

```
cd msdocs-python-flask-webapp-quickstart
```

2. Create a virtual environment for the app:

Windows

```
PowerShell
```

```
py -m venv .venv  
.venv\Scripts\Activate.ps1
```

3. Install the dependencies:

```
Console
```

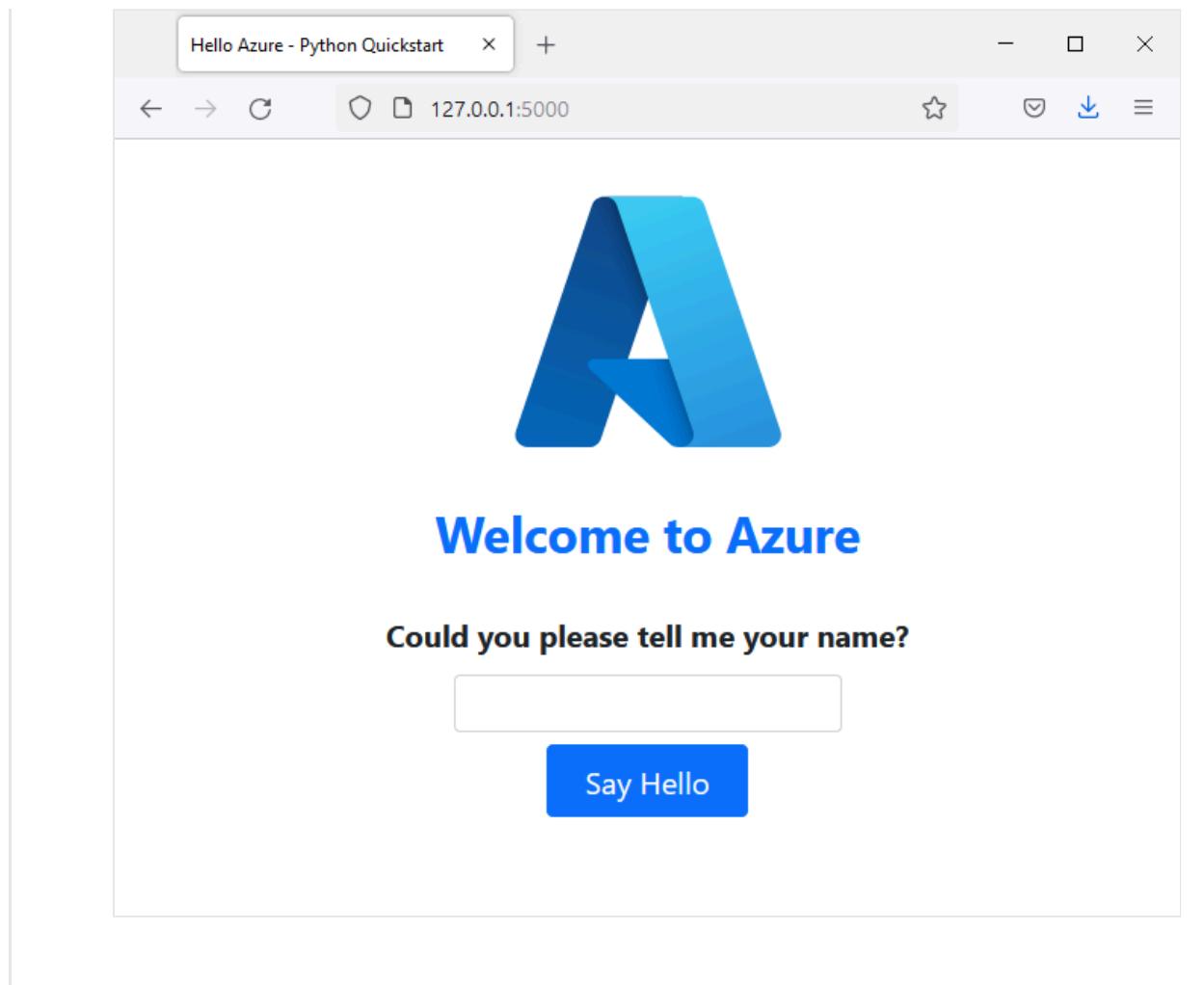
```
pip install -r requirements.txt
```

4. Run the app:

```
Console
```

```
flask run
```

5. Browse to the sample application at <http://localhost:5000> in a web browser.



Adding OpenTelemetry

To use the .NET Aspire dashboard with your Python app, you need to install the OpenTelemetry SDK and exporter. The OpenTelemetry SDK provides the API for instrumenting your application, and the exporter sends telemetry data to the .NET Aspire dashboard.

1. Install the OpenTelemetry SDK and exporter:

Console

```
pip install opentelemetry-api opentelemetry-sdk opentelemetry-exporter-otlp-proto-grpc
```

2. Add a new file to your application called `otlp_tracing.py` and add the following code:

Python

```
import logging
from opentelemetry import metrics, trace
```

```

from opentelemetry._logs import set_logger_provider
from opentelemetry.exporter.otlp.proto.grpc._log_exporter import (
    OTLPLogExporter,
)
from opentelemetry.exporter.otlp.proto.grpc.metric_exporter import
OTLPMetricExporter
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import
OTLPSpanExporter
from opentelemetry.sdk._logs import LoggerProvider, LoggingHandler
from opentelemetry.sdk._logs.export import BatchLogRecordProcessor
from opentelemetry.sdk.metrics import MeterProvider
from opentelemetry.sdk.metrics.export import
PeriodicExportingMetricReader
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor

def configure_otlp_grpc_tracing(
    endpoint: str = None
) -> trace.Tracer:
    # Configure Tracing
    traceProvider = TracerProvider()
    processor = BatchSpanProcessor(OTLPSpanExporter(endpoint=endpoint))
    traceProvider.add_span_processor(processor)
    trace.set_tracer_provider(traceProvider)

    # Configure Metrics
    reader =
    PeriodicExportingMetricReader(OTLPMetricExporter(endpoint=endpoint))
    meterProvider = MeterProvider(metric_readers=[reader])
    metrics.set_meter_provider(meterProvider)

    # Configure Logging
    logger_provider = LoggerProvider()
    set_logger_provider(logger_provider)

    exporter = OTLPLogExporter(endpoint=endpoint)

    logger_provider.add_log_record_processor(BatchLogRecordProcessor(exporter))
    handler = LoggingHandler(level=logging.NOTSET,
    logger_provider=logger_provider)
    handler.setFormatter(logging.Formatter("Python: %(message)s"))

    # Attach OTLP handler to root logger
    logging.getLogger().addHandler(handler)

    tracer = trace.get_tracer(__name__)
    return tracer

```

3. Update your application (`app.py` for Flask, `main.py` for FastAPI) to include the imports and call the `configure_otlp_grpc_tracing` function:

Python

```
import logging
from otlp_tracing import configure_otel_otlp

logging.basicConfig(level=logging.INFO)
tracer = configure_otel_otlp()
logger = logging.getLogger(__name__)
```

4. Replace the `print` calls with `logger.info` calls in your application.

5. Restart your application.

Framework Specific Instrumentation

This instrumentation has only focused on adding OpenTelemetry to our code. For more detailed instrumentation, you can use the OpenTelemetry Instrumentation packages for the specific frameworks that you are using.

Flask

1. Install the Flask instrumentation package:

Console

```
pip install opentelemetry-instrumentation-flask
```

2. Add the following code to your application:

Python

```
from opentelemetry.instrumentation.flask import FlaskInstrumentor

# add this line after configure_otel_otlp() call
FlaskInstrumentor().instrument()
```

Start the Aspire dashboard

To start the Aspire dashboard in standalone mode, run the following Docker command:

Bash

```
docker run --rm -it -p 18888:18888 -p 4317:18889 --name aspire-dashboard \
mcr.microsoft.com/dotnet/aspire-dashboard:8.0.0
```

In the Docker logs, the endpoint and key for the dashboard are displayed. Copy the key and navigate to `http://localhost:18888` in a web browser. Enter the key to log in to the dashboard.

View Structured Logs

Navigate around the Python application, and you'll see structured logs in the Aspire dashboard. The structured logs page displays logs from your application, and you can filter and search the logs.

The screenshot shows the Aspire dashboard interface. On the left, there are navigation tabs for 'Traces' and 'Metrics'. The main area is titled 'Structured logs' and displays a table of log entries. The table has columns for 'Resource', 'Level', 'Timestamp', and 'Message'. Each log entry includes a 'View' link. A total count of '10 results found' is shown at the bottom. At the top of the dashboard, there is a warning message: 'Telemetry endpoint is unsecured. Untrusted apps can send telemetry to the dashboard. [More information](#)'.

Resource	Level	Timestamp	Message	Trace	Details
my-python-app	Information	5:02:07.150 PM	⚠️ [31m 1m] WARNING: This is a development server. Do not use it in a production deployment. Us...		View
my-python-app	Information	5:02:07.150 PM	⚠️ [33m Press CTRL+C to quit ⚠️]		View
my-python-app	Information	5:02:35.154 PM	Request for index page received	2e0f908	View
my-python-app	Information	5:02:35.159 PM	127.0.0.1 - - [28/Aug/2024 07:02:35] "GET / HTTP/1.1" 200 -		View
my-python-app	Information	5:02:35.211 PM	127.0.0.1 - - [28/Aug/2024 07:02:35] "GET /static/bootstrap/css/bootstrap.min.css HTTP/1.1..."		View
my-python-app	Information	5:02:35.373 PM	127.0.0.1 - - [28/Aug/2024 07:02:35] "GET /static/images/azure-icon.svg HTTP/1.1" 304 -		View
my-python-app	Information	5:02:38.703 PM	Request for hello page received with name=aaron	87400ec	View
my-python-app	Information	5:02:38.707 PM	127.0.0.1 - - [28/Aug/2024 07:02:38] "POST /hello HTTP/1.1" 200 -		View
my-python-app	Information	5:02:38.754 PM	127.0.0.1 - - [28/Aug/2024 07:02:38] "GET /static/bootstrap/css/bootstrap.min.css HTTP/1.1..."		View
my-python-app	Information	5:02:38.912 PM	127.0.0.1 - - [28/Aug/2024 07:02:38] "GET /static/images/azure-icon.svg HTTP/1.1" 304 -		View

Next steps

You have successfully used the .NET Aspire dashboard with a Python application. To learn more about the .NET Aspire dashboard, see the [Aspire dashboard overview](#) and how to orchestrate a Python application with the [.NET Aspire app host](#).

Dashboard configuration

Article • 09/12/2024

The dashboard is configured when it starts up. Configuration includes frontend and OpenTelemetry Protocol (OTLP) addresses, the resource service endpoint, authentication, telemetry limits, and more.

When the dashboard is launched with the .NET Aspire app host project, it's automatically configured to display the app's resources and telemetry. Configuration is provided when launching the dashboard in [standalone mode](#).

There are many ways to provide configuration:

- Command line arguments.
- Environment variables. The `:` delimiter should be replaced with double underscore (`__`) in environment variable names.
- Optional JSON configuration file. The `DOTNET_DASHBOARD_CONFIG_FILE_PATH` setting can be used to specify a JSON configuration file.

Consider the following example, which shows how to configure the dashboard when started from a Docker container:

Bash

Bash

```
docker run --rm -it -p 18888:18888 -p 4317:18889 -d --name aspire-dashboard \
-e DASHBOARD_TELEMETRYLIMITS_MAXLOGCOUNT='1000' \
-e DASHBOARD_TELEMETRYLIMITS_MAXTRACECOUNT='1000' \
-e DASHBOARD_TELEMETRYLIMITS_MAXMETRICSCOUNT='1000' \
mcr.microsoft.com/dotnet/aspire-dashboard:8.0.0
```

Alternatively, these same values could be configured using a JSON configuration file that is specified using `DOTNET_DASHBOARD_CONFIG_FILE_PATH`:

JSON

```
{
  "Dashboard": {
    "TelemetryLimits": {
      "MaxLogCount": 1000,
      "MaxTraceCount": 1000,
      "MaxMetricsCount": 1000
    }
  }
}
```

ⓘ Important

The dashboard displays information about resources, including their configuration, console logs and in-depth telemetry.

Data displayed in the dashboard can be sensitive. For example, secrets in environment variables, and sensitive runtime data in telemetry. Care should be taken to configure the dashboard to secure access.

For more information, see [dashboard security](#).

Common configuration

[] Expand table

Option	Default Value	Description
<code>ASPNETCORE_URLS</code>	<code>http://localhost:18888</code>	One or more HTTP endpoints through which the dashboard frontend is served. The frontend endpoint is used to view the dashboard in a browser. When the dashboard is launched by the .NET Aspire app host this address is secured with HTTPS. Securing the dashboard with HTTPS is recommended.
<code>DOTNET_DASHBOARD_OTLP_ENDPOINT_URL</code>	<code>http://localhost:18889</code>	The OTLP/gRPC endpoint. This endpoint hosts an OTLP service and receives telemetry using gRPC. When the dashboard is launched by the .NET Aspire app host this address is secured with HTTPS. Securing the dashboard with HTTPS is recommended.
<code>DOTNET_DASHBOARD_OTLP_HTTP_ENDPOINT_URL</code>	<code>http://localhost:18890</code>	The OTLP/HTTP endpoint. This endpoint hosts an OTLP service and receives telemetry using Protobuf over HTTP. When the dashboard is launched by the .NET Aspire app host the OTLP/HTTP endpoint isn't configured by default. To configure an OTLP/HTTP endpoint with the app host, set an <code>DOTNET_DASHBOARD_OTLP_HTTP_ENDPOINT_URL</code> env var value in <code>launchSettings.json</code> . Securing the dashboard with HTTPS is recommended.
<code>DOTNET_DASHBOARD_UNSECURED_ALLOW_ANONYMOUS</code>	<code>false</code>	Configures the dashboard to not use authentication and accepts anonymous access. This setting is a shortcut to configuring <code>Dashboard:Frontend:AuthMode</code> and <code>Dashboard:Otlp:AuthMode</code> to <code>Unsecured</code> .
<code>DOTNET_DASHBOARD_CONFIG_FILE_PATH</code>	<code>null</code>	The path for a JSON configuration file. If the dashboard is being run in a Docker container, then this is the path to the configuration file in a mounted volume. This value is optional.
<code>DOTNET_RESOURCE_SERVICE_ENDPOINT_URL</code>	<code>null</code>	The gRPC endpoint to which the dashboard connects for its data. If this value is unspecified, the dashboard shows telemetry data but no resource list or console logs. This setting is a shortcut to <code>Dashboard:ResourceServiceClient:Url</code> .

Frontend authentication

The dashboard frontend endpoint authentication is configured with `Dashboard:Frontend:AuthMode`. The frontend can be secured with OpenID Connect (OIDC) or browser token authentication.

Browser token authentication works by the frontend asking for a token. The token can either be entered in the UI or provided as a query string value to the login page. For example, `https://localhost:1234/login?t=TheToken`. When the token is successfully authenticated an auth cookie is persisted to the browser, and the browser is redirected to the app.

 Expand table

Option	Default Value	Description
<code>Dashboard:Frontend:AuthMode</code>	<code>BrowserToken</code>	Can be set to <code>BrowserToken</code> , <code>OpenIdConnect</code> or <code>Unsecured</code> . <code>Unsecured</code> should only be used during local development. It's not recommended when hosting the dashboard publicly or in other settings.
<code>Dashboard:Frontend:BrowserToken</code>	<code>null</code>	Specifies the browser token. If the browser token isn't specified, then the dashboard generates one. Tooling that wants to automate logging in with browser token authentication can specify a token and open a browser with the token in the query string. A new token should be generated each time the dashboard is launched.
<code>Dashboard:Frontend:OpenIdConnect:NameClaimType</code>	<code>name</code>	Specifies one or more claim types that should be used to display the authenticated user's full name. Can be a single claim type or a comma-delimited list of claim types.
<code>Dashboard:Frontend:OpenIdConnect:UsernameClaimType</code>	<code>preferred_username</code>	Specifies one or more claim types that should be used to display the authenticated user's username. Can be a single claim type or a comma-delimited list of claim types.
<code>Dashboard:Frontend:OpenIdConnect:RequiredClaimType</code>	<code>null</code>	Specifies the claim that must be present for authorized users. Authorization fails without this claim. This value is optional.
<code>Dashboard:Frontend:OpenIdConnect:RequiredClaimValue</code>	<code>null</code>	Specifies the value of the required claim. Only used if <code>Dashboard:Frontend:OpenIdConnect:RequireClaimType</code> is also specified. This value is optional.
<code>Authentication:Schemes:OpenIdConnect:Authority</code>	<code>null</code>	URL to the identity provider (IdP).
<code>Authentication:Schemes:OpenIdConnect:ClientId</code>	<code>null</code>	Identity of the relying party (RP).
<code>Authentication:Schemes:OpenIdConnect:ClientSecret</code>	<code>null</code>	A secret that only the real RP would know.
Other properties of <code>OpenIdConnectOptions</code>	<code>null</code>	Values inside configuration section <code>Authentication:Schemes:OpenIdConnect:*</code> are bound to <code>OpenIdConnectOptions</code> , such as <code>Scope</code> .

OTLP authentication

The OTLP endpoint authentication is configured with `Dashboard:Otlp:AuthMode`. The OTLP endpoint can be secured with an API key or `client certificate` authentication.

API key authentication works by requiring each OTLP request to have a valid `x-otlp-api-key` header value. It must match either the primary or secondary key.

[Expand table](#)

Option	Default Value	Description
Dashboard:Otlp:AuthMode	Unsecured	Can be set to <code>ApiKey</code> , <code>Certificate</code> or <code>Unsecured</code> . <code>Unsecured</code> should only be used during local development. It's not recommended when hosting the dashboard publicly or in other settings.
Dashboard:Otlp:PrimaryApiKey	null	Specifies the primary API key. The API key can be any text, but a value with at least 128 bits of entropy is recommended. This value is required if auth mode is API key.
Dashboard:Otlp:SecondaryApiKey	null	Specifies the secondary API key. The API key can be any text, but a value with at least 128 bits of entropy is recommended. This value is optional. If a second API key is specified, then the incoming <code>x-otlp-api-key</code> header value can match either the primary or secondary key.

OTLP CORS

CORS (Cross-Origin Resource Sharing) can be configured to allow browser apps to send telemetry to the dashboard.

By default, browser apps are restricted from making cross domain API calls. This impacts sending telemetry to the dashboard because the dashboard and the browser app are always on different domains. Configuring CORS in the .NET Aspire dashboard removes the restriction and allows browser apps with the [OpenTelemetry SDK for JavaScript](#) to send telemetry directly to the dashboard OTLP HTTP endpoint.

Using CORS, the dashboard and browser telemetry together is demonstrated in the [browser telemetry](#) sample.

To configure CORS, use the `Dashboard:Otlp:Cors` section and specify the allowed origins and headers:

JSON

```
{
  "Dashboard": {
    "Otlp": {
      "Cors": {
        "AllowedOrigins": "http://localhost:5000,https://localhost:5001"
      }
    }
  }
}
```

Consider the following configuration options:

[Expand table](#)

Option	Default Value	Description
Dashboard:Otlp:Cors:AllowedOrigins	null	Specifies the allowed origins for CORS. It's a comma-delimited string and can include the <code>*</code> wildcard to allow any domain. This option is optional and can be set using the <code>DASHBOARD__OTLP__CORS__ALLOWEDORIGINS</code> environment variable.

Option	Default Value	Description
<code>Dashboard:Otlp:Cors:AllowedHeaders</code>	<code>null</code>	A comma-delimited string representing the allowed headers for CORS. This setting is optional and can be set using the <code>DASHBOARD_OTLP_CORS_ALLOWEDHEADERS</code> environment variable.

ⓘ Note

The dashboard only supports the `POST` method for sending telemetry and doesn't allow configuration of the *allowed methods* (`Access-Control-Allow-Methods`) for CORS.

Resources

The dashboard connects to a resource service to load and display resource information. The client is configured in the dashboard for how to connect to the service.

The resource service client authentication is configured with `Dashboard:ResourceServiceClient:AuthMode`. The client can be configured to support API key or client certificate authentication.

[Expand table](#)

Option	Default Value	Description
<code>Dashboard:ResourceServiceClient:Url</code>	<code>null</code>	The gRPC endpoint to which the dashboard connects for its data. If this value is unspecified, the dashboard shows telemetry data but no resource list or console logs.
<code>Dashboard:ResourceServiceClient:AuthMode</code>	<code>null</code>	Can be set to <code>ApiKey</code> , <code>Certificate</code> or <code>Unsecured</code> . <code>Unsecured</code> should only be used during local development. It's not recommended when hosting the dashboard publicly or in other settings. This value is required if a resource service URL is specified.
<code>Dashboard:ResourceServiceClient:ApiKey</code>	<code>null</code>	The API to send to the resource service in the <code>x-resource-service-api-key</code> header. This value is required if auth mode is API key.
<code>Dashboard:ResourceServiceClient:ClientCertificate:Source</code>	<code>null</code>	Can be set to <code>File</code> or <code>KeyStore</code> . This value is required if auth mode is client certificate.
<code>Dashboard:ResourceServiceClient:ClientCertificate:FilePath</code>	<code>null</code>	The certificate file path. This value is required if source is <code>File</code> .
<code>Dashboard:ResourceServiceClient:ClientCertificate:Password</code>	<code>null</code>	The password for the certificate file. This value is optional.
<code>Dashboard:ResourceServiceClient:ClientCertificate:Subject</code>	<code>null</code>	The certificate subject. This value is required if source is <code>KeyStore</code> .
<code>Dashboard:ResourceServiceClient:ClientCertificate:Store</code>	<code>My</code>	The certificate <code>StoreName</code> .

Option	Default Value	Description
Dashboard:ResourceServiceClient:ClientCertificate:Location	CurrentUser	The certificate StoreLocation .

Telemetry limits

Telemetry is stored in memory. To avoid excessive memory usage, the dashboard has limits on the count and size of stored telemetry. When a count limit is reached, new telemetry is added, and the oldest telemetry is removed. When a size limit is reached, data is truncated to the limit.

Telemetry limits have different scopes depending upon the telemetry type:

- `MaxLogCount` and `MaxTraceCount` are shared across resources. For example, a `MaxLogCount` value of 5,000 configures the dashboard to store up to 5,000 total log entries for all resources.
- `MaxMetricsCount` is per-resource. For example, a `MaxMetricsCount` value of 10,000 configures the dashboard to store up to 10,000 metrics data points per-resource.

[\[+\] Expand table](#)

Option	Default Value	Description
Dashboard:TelemetryLimits:MaxLogCount	10,000	The maximum number of log entries. Limit is shared across resources.
Dashboard:TelemetryLimits:MaxTraceCount	10,000	The maximum number of log traces. Limit is shared across resources.
Dashboard:TelemetryLimits:MaxMetricsCount	50,000	The maximum number of metric data points. Limit is per-resource.
Dashboard:TelemetryLimits:MaxAttributeCount	128	The maximum number of attributes on telemetry.
Dashboard:TelemetryLimits:MaxAttributeLength	null	The maximum length of attributes.
Dashboard:TelemetryLimits:MaxSpanEventCount	null	The maximum number of events on span attributes.

Other

[\[+\] Expand table](#)

Option	Default Value	Description
Dashboard:ApplicationName	Aspire	The application name to be displayed in the UI. This applies only when no resource service URL is specified. When a resource service exists, the service specifies the application name.

Next steps

[Security considerations for running the .NET Aspire dashboard](#)

Security considerations for running the .NET Aspire dashboard

Article • 05/31/2024

The [.NET Aspire dashboard](#) offers powerful insights to your apps. The dashboard displays information about resources, including their configuration, console logs and in-depth telemetry.

Data displayed in the dashboard can be sensitive. For example, configuration can include secrets in environment variables, and telemetry can include sensitive runtime data. Care should be taken to secure access to the dashboard.

Scenarios for running the dashboard

The dashboard can be run in different scenarios, such as being automatically starting by .NET Aspire tooling, or as a standalone application that is separate from other .NET Aspire integrations. Steps to secure the dashboard depend on how it's being run.

.NET Aspire tooling

The dashboard is automatically started when an .NET Aspire app host is run. The dashboard is secure by default when run from .NET Aspire tooling:

- Transport is secured with HTTPS. Using HTTPS is configured by default in `launchSettings.json`. The launch profile includes `https` addresses in `applicationUrl` and `DOTNET_DASHBOARD_OTLP_ENDPOINT_URL` values.
- Browser frontend authenticated with a browser token.
- Incoming telemetry authenticated with an API key.

HTTPS in the dashboard uses the ASP.NET Core development certificate. The certificate must be trusted for the dashboard to work correctly. The steps required to trust the development cert are different depending on the machine's operating system:

- [Trust the ASP.NET Core HTTPS development certificate on Windows and macOS](#)
- [Trust HTTPS certificate on Linux](#)

There are scenarios where you might want to allow an unsecured transport. The dashboard can run without HTTPS from the .NET Aspire app host by configuring the `ASPIRE_ALLOW_UNSECURED_TRANSPORT` setting to `true`. For more information, see [Allow unsecured transport in .NET Aspire](#).

Standalone mode

The dashboard is shipped as a Docker image and can be used without the rest of .NET Aspire. When the dashboard is launched in standalone mode, it defaults to a mix of secure and unsecured settings.

- Browser frontend authenticated with a browser token.
- Incoming telemetry is unsecured. Warnings are displayed in the console and dashboard UI.

The telemetry endpoint accepts incoming OTLP data without authentication. When the endpoint is unsecured, the dashboard is open to receiving telemetry from untrusted apps.

For information about securing the telemetry when running the dashboard in standalone mode, see [Securing the telemetry endpoint](#).

Secure telemetry endpoint

The .NET Aspire dashboard provides a variety of ways to view logs, traces, and metrics for your app. This information enables you to track the behavior and performance of your app and to diagnose any issues that arise. It's important that you can trust this information, and a warning is displayed in the dashboard UI if telemetry isn't secured.

The dashboard collects telemetry through an [OTLP \(OpenTelemetry protocol\)](#) endpoint. Apps send telemetry to this endpoint, and the dashboard stores the external information it receives in memory, which is then accessible via the UI.

To prevent untrusted apps from sending telemetry to .NET Aspire, the OTLP endpoint should be secured. The OTLP endpoint is automatically secured with an API key when the dashboard is started by .NET Aspire tooling. Additional configuration is required for standalone mode.

API key authentication can be enabled on the telemetry endpoint with some additional configuration:

```
Bash
Bash
docker run --rm -it -p 18888:18888 -p 4317:18889 -d --name aspire-dashboard \
-e DASHBOARD__OTLP__AUTHMODE='ApiKey' \
```

```
-e DASHBOARD__OTLP__PRIMARYAPIKEY='{MY_APIKEY}' \
mcr.microsoft.com/dotnet/aspire-dashboard:8.0.0
```

The preceding Docker command:

- Starts the .NET Aspire dashboard image and exposes OTLP endpoint as port 4317
- Configures the OTLP endpoint to use `ApiKey` authentication. This requires that incoming telemetry has a valid `x-otlp-api-key` header value.
- Configures the expected API key. `{MY_APIKEY}` in the example value should be replaced with a real API key. The API key can be any text, but a value with at least 128 bits of entropy is recommended.

When API key authentication is configured, the dashboard validates incoming telemetry has a required API key. Apps that send the dashboard telemetry must be configured to send the API key. This can be configured in .NET with `otlpExporterOptions.Headers`:

```
C#  
  
builder.Services.Configure<OtlpExporterOptions>(  
    o => o.Headers = $"x-otlp-api-key={MY_APIKEY}");
```

Other languages have different OpenTelemetry APIs. Passing the `OTEL_EXPORTER_OTLP_HEADERS` environment variable [↗](#) to apps is a universal way to configure the header.

Memory exhaustion

The dashboard stores external information it receives in memory, such as resource details and telemetry. While the number of resources the dashboard tracks are bounded, there isn't a limit to how much telemetry apps send to the dashboard. Limits must be placed on how much information is stored to prevent the dashboard using an excessive amount of memory and exhausting available memory on the current machine.

Telemetry limits

To help prevent memory exhaustion, the dashboard limits how much telemetry it stores by default. For example, there is a maximum of 10,000 structured log entries per resource. Once the limit is reached, each new new log entry received causes an old entry to be removed.

Configuration can customize telemetry limits.

Testing .NET Aspire solutions

Article • 09/10/2024

In this article, you learn how to create a test project, write tests, and run them for your .NET Aspire solutions. The tests in this article aren't unit tests, but rather functional or integration tests. .NET Aspire includes several variations of [testing project templates](#) that you can use to test your .NET Aspire resource dependencies—and their communications. The testing project templates are available for MSTest, NUnit, and xUnit testing frameworks and include a sample test that you can use as a starting point for your tests.

The .NET Aspire test project templates rely on the [Aspire.Hosting.Testing](#) NuGet package. This package exposes the `DistributedApplicationTestingBuilder` class, which is used to create a test host for your distributed application. The distributed application testing builder relies on the `DistributedApplication` class to create and start the [app host](#).

Create a test project

The easiest way to create a .NET Aspire test project is to use the testing project template. If you're starting a new .NET Aspire project and want to include test projects, the [Visual Studio tooling supports that option](#). If you're adding a test project to an existing .NET Aspire project, you can use the `dotnet new` command to create a test project:

```
dotnet new aspire-xunit
```

Explore the test project

The following example test project was created as part of the [.NET Aspire Starter Application](#) template. If you're unfamiliar with it, see [Quickstart: Build your first .NET Aspire project](#). The .NET Aspire test project takes a project reference dependency on the target app host. Consider the template project:

```
<Project Sdk="Microsoft.NET.Sdk">  
  
<PropertyGroup>  
  <TargetFramework>net8.0</TargetFramework>  
  <ImplicitUsings>enable</ImplicitUsings>
```

```

<Nullable>enable</Nullable>
<IsPackable>false</IsPackable>
<IsTestProject>true</IsTestProject>
</PropertyGroup>

<ItemGroup>
  <PackageReference Include="Aspire.Hosting.Testing" Version="8.2.0" />
  <PackageReference Include="coverlet.collector" Version="6.0.2" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.10.0" />
  <PackageReference Include="xunit" Version="2.9.0" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.8.2" />
</ItemGroup>

<ItemGroup>
  <ProjectReference
Include="..\AspireApp.AppHost\AspireApp.AppHost.csproj" />
</ItemGroup>

<ItemGroup>
  <Using Include="System.Net" />
  <Using Include="Microsoft.Extensions.DependencyInjection" />
  <Using Include="Aspire.Hosting.ApplicationModel" />
  <Using Include="Aspire.Hosting.Testing" />
  <Using Include="Xunit" />
</ItemGroup>

</Project>

```

The preceding project file is fairly standard. There's a `PackageReference` to the [Aspire.Hosting.Testing](#) NuGet package, which includes the required types to write tests for .NET Aspire projects.

The template test project includes a `WebTests` class with a single test. The test verifies the following scenario:

- The app host is successfully created and started.
- The `webfrontend` resource is available and running.
- An HTTP request can be made to the `webfrontend` resource and returns a successful response (HTTP 200 OK).

Consider the following test class:

```

namespace AspireApp.Tests;

public class WebTests
{
  [Fact]
  public async Task GetWebResourceRootReturnsOkStatusCode()
  {

```

```

// Arrange
var appHost = await DistributedApplicationTestingBuilder
    .CreateAsync<Projects.AspireApp_AppHost>();

appHost.Services.ConfigureHttpClientDefaults(clientBuilder =>
{
    clientBuilder.AddStandardResilienceHandler();
});

// To output logs to the xUnit.net ITestOutputHelper,
// consider adding a package from https://www.nuget.org/packages?
q=xunit+logging

await using var app = await appHost.BuildAsync();

var resourceNotificationService = app.Services
    .GetRequiredService<ResourceNotificationService>();

await app.StartAsync();

// Act
var httpClient = app.CreateHttpClient("webfrontend");

await resourceNotificationService.WaitForResourceAsync(
    "webfrontend",
    KnownResourceStates.Running
)
    .WaitAsync(TimeSpan.FromSeconds(30));

var response = await httpClient.GetAsync("/");

// Assert
Assert.Equal(HttpStatusCode.OK, response.StatusCode);
}
}

```

The preceding code:

- Relies on the `DistributedApplicationTestingBuilder` to asynchronously create the app host.
 - The `appHost` is an instance of `IDistributedApplicationTestingBuilder` that represents the app host.
 - The `appHost` instance has its service collection configured with the standard HTTP resilience handler. For more information, see [Build resilient HTTP apps: Key development patterns](#).
- The `appHost` has its `BuildAsync` method invoked, which returns the `DistributedApplication` instance as the `app`.
 - The `app` has its service provider get the `ResourceNotificationService` instance.
 - The `app` is started asynchronously.

- An `HttpClient` is created for the `webfrontend` resource by calling `app.CreateHttpClient`.
- The `resourceNotificationService` is used to wait for the `webfrontend` resource to be available and running.
- A simple HTTP GET request is made to the root of the `webfrontend` resource.
- The test asserts that the response status code is `OK`.

Test resource environment variables

To further test resources and their expressed dependencies in your .NET Aspire solution, you can assert that environment variables are injected correctly. The following example demonstrates how to test that the `webfrontend` resource has an HTTPS environment variable that resolves to the `apiservice` resource:

```
namespace AspireApp.Tests;

public class EnvVarTests
{
    [Fact]
    public async Task WebResourceEnvVarsResolveTo ApiService()
    {
        // Arrange
        var appHost = await DistributedApplicationTestingBuilder
            .CreateAsync<Projects.AspireApp_AppHost>();

        var frontend = (IResourceWithEnvironment)appHost.Resources
            .Single(static r => r.Name == "webfrontend");

        // Act
        var envVars = await frontend.GetEnvironmentVariableValuesAsync(
            DistributedApplicationOperation.Publish);

        // Assert
        Assert.Contains(envVars, static (kvp) =>
        {
            var (key, value) = kvp;

            return key is "services__apiservice__https__0"
                && value is "{apiservice.bindings.https.url}";
        });
    }
}
```

The preceding code:

- Relies on the `DistributedApplicationTestingBuilder` to asynchronously create the app host.

- The `builder` instance is used to retrieve an `IResourceWithEnvironment` instance named "webfrontend" from the `IDistributedApplicationTestingBuilder.Resources`.
- The `webfrontend` resource is used to call `GetEnvironmentVariableValuesAsync` to retrieve its configured environment variables.
- The `DistributedApplicationOperation.Publish` argument is passed when calling `GetEnvironmentVariableValuesAsync` to specify environment variables that are published to the resource as binding expressions.
- With the returned environment variables, the test asserts that the `webfrontend` resource has an HTTPS environment variable that resolves to the `apiservice` resource.

Summary

The .NET Aspire testing project template makes it easier to create test projects for .NET Aspire solutions. The template project includes a sample test that you can use as a starting point for your tests. The `DistributedApplicationTestingBuilder` follows a familiar pattern to the `WebApplicationFactory` in ASP.NET Core. It allows you to create a test host for your distributed application and run tests against it.

Finally, when using the `DistributedApplicationTestingBuilder` all resource logs are redirected to the `DistributedApplication` by default. The redirection of resource logs enables scenarios where you want to assert that a resource is logging correctly.

See also

- [Unit testing C# in .NET using dotnet test and xUnit](#)
- [MSTest overview](#)
- [Unit testing C# with NUnit and .NET Core](#)

.NET Aspire service discovery

Article • 04/10/2024

In this article, you learn how service discovery works within a .NET Aspire project. .NET Aspire includes functionality for configuring service discovery at development and testing time. Service discovery functionality works by providing configuration in the format expected by the *configuration-based endpoint resolver* from the .NET Aspire AppHost project to the individual service projects added to the application model. For more information, see [Service discovery in .NET](#).

Implicit service discovery by reference

Configuration for service discovery is only added for services that are referenced by a given project. For example, consider the following AppHost program:

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

In the preceding example, the *frontend* project references the *catalog* project and the *basket* project. The two [WithReference](#) calls instruct the .NET Aspire project to pass service discovery information for the referenced projects (*catalog*, and *basket*) into the *frontend* project.

Named endpoints

Some services expose multiple, named endpoints. Named endpoints can be resolved by specifying the endpoint name in the host portion of the HTTP request URI, following the format `scheme://_endpointName.serviceName`. For example, if a service named "basket" exposes an endpoint named "dashboard", then the URI

`scheme+http://_dashboard.basket` can be used to specify this endpoint, for example:

```
C#
```

```
builder.Services.AddHttpClient<BasketServiceClient>(
    static client => client.BaseAddress = new("https+http://basket"));

builder.Services.AddHttpClient<BasketServiceDashboardClient>(
    static client => client.BaseAddress =
new("https+http://_dashboard.basket"));
```

In the preceding example, two `HttpClient` classes are added, one for the core basket service and one for the basket service's dashboard.

Named endpoints using configuration

With the configuration-based endpoint resolver, named endpoints can be specified in configuration by prefixing the endpoint value with `_endpointName.`, where `endpointName` is the endpoint name. For example, consider this `appsettings.json` configuration which defined a default endpoint (with no name) and an endpoint named "dashboard":

JSON

```
{
  "Services": {
    "basket": {
      "https": "https://10.2.3.4:8080", /* the https endpoint, requested via
https://basket */
      "dashboard": "https://10.2.3.4:9999" /* the "dashboard" endpoint,
requested via https://_dashboard.basket */
    }
  }
}
```

In the preceding JSON:

- The default endpoint, when resolving `https://basket` is `10.2.3.4:8080`.
- The "dashboard" endpoint, resolved via `https://_dashboard.basket` is `10.2.3.4:9999`.

Named endpoints in .NET Aspire

C#

```
var basket = builder.AddProject<Projects.BasketService>("basket")
    .WithHttpsEndpoint(hostPort: 9999, name: "dashboard");
```

Named endpoints in Kubernetes using DNS SRV

When deploying to [Kubernetes](#), the DNS SRV service endpoint resolver can be used to resolve named endpoints. For example, the following resource definition will result in a DNS SRV record being created for an endpoint named "default" and an endpoint named "dashboard", both on the service named "basket".

yml

```
apiVersion: v1
kind: Service
metadata:
  name: basket
spec:
  selector:
    name: basket-service
  clusterIP: None
  ports:
    - name: default
      port: 8080
    - name: dashboard
      port: 9999
```

To configure a service to resolve the "dashboard" endpoint on the "basket" service, add the DNS SRV service endpoint resolver to the host builder as follows:

C#

```
builder.Services.AddServiceDiscoveryCore();
builder.Services.AddDnsSrvServiceEndpointProvider();
```

For more information, see [AddServiceDiscoveryCore](#) and [AddDnsSrvServiceEndpointProvider](#).

The special port name "default" is used to specify the default endpoint, resolved using the URI `https://basket`.

As in the previous example, add service discovery to an `HttpClient` for the basket service:

C#

```
builder.Services.AddHttpClient<BasketServiceClient>(
  static client => client.BaseAddress = new("https://basket"));
```

Similarly, the "dashboard" endpoint can be targeted as follows:

```
builder.Services.AddHttpClient<BasketServiceDashboardClient>(  
    static client => client.BaseAddress = new("https://_dashboard.basket"));
```

See also

- [Service discovery in .NET](#)
- [Make HTTP requests with the HttpClient class](#)
- [IHttpClientFactory with .NET](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

.NET Aspire service defaults

Article • 09/24/2024

In this article, you learn about the .NET Aspire service defaults project, a set of extension methods that wire up [telemetry](#), [health checks](#), [service discovery](#), and are designed to be customizable and extensible.

Cloud-native applications often require extensive configurations to ensure they work across different environments reliably and securely. .NET Aspire provides many helper methods and tools to streamline the management of configurations for OpenTelemetry, health checks, environment variables, and more.

Explore the service defaults project

When you either [Enlist in .NET Aspire orchestration](#) or [create a new .NET Aspire project](#), the `YourAppName.ServiceDefaults.csproj` project is added to your solution. For example, when building an API, you call the `AddServiceDefaults` method in the `Program.cs` file of your apps:

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

The `AddServiceDefaults` method handles the following concerns for you:

- Configures OpenTelemetry metrics and tracing.
- Add default health check endpoints.
- Add service discovery functionality.
- Configures [HttpClient](#) to work with service discovery.

For more information, see [Provided extension methods](#) for details on the `AddServiceDefaults` method.

Important

The .NET Aspire service defaults project is specifically designed for sharing the `Extensions.cs` file and its functionality. Please refrain from including other shared functionality or models in this project, use a conventional shared class library project for those purposes.

Project characteristics

The `YourAppName.ServiceDefaults` project is a .NET 8.0 library that contains the following XML:

XML

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

<PropertyGroup>
  <TargetFramework>net8.0</TargetFramework>
  <ImplicitUsings>enable</ImplicitUsings>
  <Nullable>enable</Nullable>
  <IsAspireSharedProject>true</IsAspireSharedProject>
</PropertyGroup>

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

    <PackageReference Include="Microsoft.Extensions.Http.Resilience"
Version="8.9.1" />
    <PackageReference Include="Microsoft.Extensions.ServiceDiscovery"
Version="8.2.0" />
    <PackageReference Include="OpenTelemetry.Exporter.OpenTelemetryProtocol"
Version="1.9.0" />
    <PackageReference Include="OpenTelemetry.Extensions.Hosting"
Version="1.9.0" />
    <PackageReference Include="OpenTelemetry.Instrumentation.AspNetCore"
Version="1.9.0" />
    <PackageReference Include="OpenTelemetry.Instrumentation.Http"
Version="1.9.0" />
    <PackageReference Include="OpenTelemetry.Instrumentation.Runtime"
Version="1.9.0" />
  </ItemGroup>

</Project>
```

The service defaults project template imposes a `FrameworkReference` dependency on `Microsoft.AspNetCore.App`.

💡 Tip

If you don't want to take a dependency on `Microsoft.AspNetCore.App`, you can create a custom service defaults project. For more information, see [Custom service defaults](#).

The `IsAspireSharedProject` property is set to `true`, which indicates that this project is a shared project. The .NET Aspire tooling uses this project as a reference for other projects

added to a .NET Aspire solution. When you enlist the new project for orchestration, it automatically references the *YourAppName.ServiceDefaults* project and updates the *Program.cs* file to call the `AddServiceDefaults` method.

Provided extension methods

The *YourAppName.ServiceDefaults* project exposes a single *Extensions.cs* file that contains several opinionated extension methods:

- `AddServiceDefaults`: Adds service defaults functionality.
- `ConfigureOpenTelemetry`: Configures OpenTelemetry metrics and tracing.
- `AddDefaultHealthChecks`: Adds default health check endpoints.
- `MapDefaultEndpoints`: Maps the health checks endpoint to `/health` and the liveness endpoint to `/alive`.

Add service defaults functionality

The `AddServiceDefaults` method defines default configurations with the following opinionated functionality:

C#

```
public static IHostApplicationBuilder AddServiceDefaults(
    this IHostApplicationBuilder builder)
{
    builder.ConfigureOpenTelemetry();

    builder.AddDefaultHealthChecks();

    builder.Services.AddServiceDiscovery();

    builder.Services.ConfigureHttpClientDefaults(http =>
    {
        // Turn on resilience by default
        http.AddStandardResilienceHandler();

        // Turn on service discovery by default
        http.AddServiceDiscovery();
    });

    // Uncomment the following to restrict the allowed schemes for service
    // discovery.
    // builder.Services.Configure<ServiceDiscoveryOptions>(options =>
    // {
    //     options.AllowedSchemes = ["https"];
    // });
}
```

```
    return builder;
}
```

The preceding code:

- Configures OpenTelemetry metrics and tracing, by calling the `ConfigureOpenTelemetry` method.
- Adds default health check endpoints, by calling the `AddDefaultHealthChecks` method.
- Adds [service discovery](#) functionality, by calling the `AddServiceDiscovery` method.
- Configures [HttpClient](#) defaults, by calling the `ConfigureHttpClientDefaults` method—which is based on [Build resilient HTTP apps: Key development patterns](#):
 - Adds the standard HTTP resilience handler, by calling the `AddStandardResilienceHandler` method.
 - Specifies that the `IHttpClientBuilder` should use service discovery, by calling the `UseServiceDiscovery` method.
- Returns the `IHostApplicationBuilder` instance to allow for method chaining.

OpenTelemetry configuration

Telemetry is a critical part of any cloud-native application. .NET Aspire provides a set of opinionated defaults for OpenTelemetry, which are configured with the `ConfigureOpenTelemetry` method:

C#

```
public static IHostApplicationBuilder ConfigureOpenTelemetry(
    this IHostApplicationBuilder builder)
{
    builder.Logging.AddOpenTelemetry(logging =>
    {
        logging.IncludeFormattedMessage = true;
        logging.IncludeScopes = true;
    });

    builder.Services.AddOpenTelemetry()
        .WithMetrics(metrics =>
    {
        metrics.AddAspNetCoreInstrumentation()
            .AddHttpClientInstrumentation()
            .AddRuntimeInstrumentation();
    })
        .WithTracing(tracing =>
    {
        if (builder.Environment.IsDevelopment())
        {
```

```

        // We want to view all traces in development
        tracing.SetSampler(new AlwaysOnSampler());
    }

    tracing.AddAspNetCoreInstrumentation()
        // Uncomment the following line to enable gRPC
instrumentation
        // (requires the OpenTelemetry.Instrumentation.GrpcNetClient
package)
        //.AddGrpcClientInstrumentation()
        .AddHttpClientInstrumentation();
    });

builder.AddOpenTelemetryExporters();

return builder;
}

```

The `ConfigureOpenTelemetry` method:

- Adds [.NET Aspire telemetry](#) logging to include formatted messages and scopes.
- Adds OpenTelemetry metrics and tracing that include:
 - Runtime instrumentation metrics.
 - ASP.NET Core instrumentation metrics.
 - HttpClient instrumentation metrics.
 - In a development environment, the `AlwaysOnSampler` is used to view all traces.
 - Tracing details for ASP.NET Core, gRPC and HTTP instrumentation.
- Adds OpenTelemetry exporters, by calling `AddOpenTelemetryExporters`.

The `AddOpenTelemetryExporters` method is defined privately as follows:

C#

```

private static IApplicationBuilder AddOpenTelemetryExporters(
    this IApplicationBuilder builder)
{
    var useOtlpExporter = !string.IsNullOrWhiteSpace(
        builder.Configuration["OTEL_EXPORTER_OTLP_ENDPOINT"]);

    if (useOtlpExporter)
    {
        builder.Services.Configure<OpenTelemetryLoggerOptions>(
            logging => logging.AddOtlpExporter());
        builder.Services.ConfigureOpenTelemetryMeterProvider(
            metrics => metrics.AddOtlpExporter());
        builder.Services.ConfigureOpenTelemetryTracerProvider(
            tracing => tracing.AddOtlpExporter());
    }

    // Uncomment the following lines to enable the Prometheus exporter

```

```

    // (requires the OpenTelemetry.Exporter.Prometheus.AspNetCore package)
    // builder.Services.AddOpenTelemetry()
    //     .WithMetrics(metrics => metrics.AddPrometheusExporter());

    // Uncomment the following lines to enable the Azure Monitor exporter
    // (requires the Azure.Monitor.OpenTelemetry.AspNetCore package)
    //if (!string.IsNullOrEmpty(
    //    builder.Configuration["APPLICATIONINSIGHTS_CONNECTION_STRING"]))
    //{
    //    builder.Services.AddOpenTelemetry()
    //        .UseAzureMonitor();
    //}

    return builder;
}

```

The `AddOpenTelemetryExporters` method adds OpenTelemetry exporters based on the following conditions:

- If the `OTEL_EXPORTER_OTLP_ENDPOINT` environment variable is set, the OpenTelemetry exporter is added.
- Optionally consumers of .NET Aspire service defaults can uncomment some code to enable the Prometheus exporter, or the Azure Monitor exporter.

For more information, see [.NET Aspire telemetry](#).

Health checks configuration

Health checks are used by various tools and systems to assess the readiness of your app. .NET Aspire provides a set of opinionated defaults for health checks, which are configured with the `AddDefaultHealthChecks` method:

```

C#

public static IApplicationBuilder AddDefaultHealthChecks(
    this IApplicationBuilder builder)
{
    builder.Services.AddHealthChecks()
        // Add a default liveness check to ensure app is responsive
        .AddCheck("self", () => HealthCheckResult.Healthy(), ["live"]);

    return builder;
}

```

The `AddDefaultHealthChecks` method adds a default liveness check to ensure the app is responsive. The call to `AddHealthChecks` registers the `HealthCheckService`. For more information, see [.NET Aspire health checks](#).

Web app health checks configuration

To expose health checks in a web app, .NET Aspire automatically determines the type of project being referenced within the solution, and adds the appropriate call to

`MapDefaultEndpoints`:

C#

```
public static WebApplication MapDefaultEndpoints(this WebApplication app)
{
    // Uncomment the following line to enable the Prometheus endpoint
    // (requires the OpenTelemetry.Exporter.Prometheus.AspNetCore package)
    // app.MapPrometheusScrapingEndpoint();

    // Adding health checks endpoints to applications in non-development
    // environments has security implications.
    // See https://aka.ms/dotnet/aspire/healthchecks for details before
    // enabling these endpoints in non-development environments.
    if (app.Environment.IsDevelopment())
    {
        // All health checks must pass for app to be considered ready to
        // accept traffic after starting
        app.MapHealthChecks("/health");

        // Only health checks tagged with the "live" tag must pass for
        // app to be considered alive
        app.MapHealthChecks("/alive", new HealthCheckOptions
        {
            Predicate = r => r.Tags.Contains("live")
        });
    }

    return app;
}
```

The `MapDefaultEndpoints` method:

- Allows consumers to optionally uncomment some code to enable the Prometheus endpoint.
- Maps the health checks endpoint to `/health`.
- Maps the liveness endpoint to `/alive` route where the health check tag contains `live`.

For more information, see [.NET Aspire health checks](#).

Custom service defaults

If the default service configuration provided by the project template is not sufficient for your needs, you have the option to create your own service defaults project. This is especially useful when your consuming project, such as a Worker project or WinForms project, cannot or does not want to have a `FrameworkReference` dependency on `Microsoft.AspNetCore.App`.

To do this, create a new .NET 8.0 class library project and add the necessary dependencies to the project file, consider the following example:

XML

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

  <PropertyGroup>
    <OutputType>Library</OutputType>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" />
    <PackageReference Include="Microsoft.Extensions.ServiceDiscovery" />
    <PackageReference Include="Microsoft.Extensions.Http.Resilience" />
    <PackageReference Include="OpenTelemetry.Exporter.OpenTelemetryProtocol" />
    <PackageReference Include="OpenTelemetry.Extensions.Hosting" />
    <PackageReference Include="OpenTelemetry.Instrumentation.Http" />
    <PackageReference Include="OpenTelemetry.Instrumentation.Runtime" />
  </ItemGroup>
</Project>
```

Then create an extensions class that contains the necessary methods to configure the app defaults:

C#

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using OpenTelemetry.Logs;
using OpenTelemetry.Metrics;
using OpenTelemetry.Trace;

namespace Microsoft.Extensions.Hosting;

public static class AppDefaultsExtensions
{
    public static IApplicationBuilder AddAppDefaults(
        this IApplicationBuilder builder)
    {
        builder.ConfigureAppOpenTelemetry();
```

```

        builder.Services.AddServiceDiscovery();

        builder.Services.ConfigureHttpClientDefaults(http =>
{
    // Turn on resilience by default
    http.AddStandardResilienceHandler();

    // Turn on service discovery by default
    http.AddServiceDiscovery();
});

return builder;
}

public static IHostApplicationBuilder ConfigureAppOpenTelemetry(
    this IHostApplicationBuilder builder)
{
    builder.Logging.AddOpenTelemetry(logging =>
{
    logging.IncludeFormattedMessage = true;
    logging.IncludeScopes = true;
});

builder.Services.AddOpenTelemetry()
    .WithMetrics(static metrics =>
{
    metrics.AddRuntimeInstrumentation();
})
    .WithTracing(tracing =>
{
    if (builder.Environment.IsDevelopment())
    {
        // We want to view all traces in development
        tracing.SetSampler(new AlwaysOnSampler());
    }

    tracing.AddGrpcClientInstrumentation()
        .AddHttpClientInstrumentation();
});

builder.AddOpenTelemetryExporters();

return builder;
}

private static IHostApplicationBuilder AddOpenTelemetryExporters(
    this IHostApplicationBuilder builder)
{
    var useOtlpExporter =
        !string.IsNullOrWhiteSpace(
            builder.Configuration["OTEL_EXPORTER_OTLP_ENDPOINT"]);

    if (useOtlpExporter)
    {
        builder.Services.Configure<OpenTelemetryLoggerOptions>(

```

```
        logging => logging.AddOtlpExporter());
    builder.Services.ConfigureOpenTelemetryMeterProvider(
        metrics => metrics.AddOtlpExporter());
    builder.Services.ConfigureOpenTelemetryTracerProvider(
        tracing => tracing.AddOtlpExporter());
}

return builder;
}
}
```

This is only an example, and you can customize the `AppDefaultsExtensions` class to meet your specific needs.

Next steps

This code is derived from the .NET Aspire Starter Application template and is intended as a starting point. You're free to modify this code however you deem necessary to meet your needs. It's important to know that service defaults project and its functionality are automatically applied to all project resources in a .NET Aspire solution.

- [Service discovery in .NET Aspire](#)
- [Health checks in .NET Aspire](#)
- [.NET Aspire telemetry](#)
- [Build resilient HTTP apps: Key development patterns](#)

.NET Aspire and launch profiles

Article • 04/23/2024

.NET Aspire makes use of *launch profiles* defined in both the app host and service projects to simplify the process of configuring multiple aspects of the debugging and publishing experience for .NET Aspire-based distributed applications.

Launch profile basics

When creating a new .NET application from a template developers will often see a `Properties` directory which contains a file named `launchSettings.json`. The launch settings file contains a list of *launch profiles*. Each launch profile is a collection of related options which defines how you would like `dotnet` to start your application.

The code below is an example of launch profiles in a `launchSettings.json` file for an **ASP.NET Core** application.

JSON

```
{  
  "$schema": "http://json.schemastore.org/launchsettings.json",  
  "profiles": {  
    "http": {  
      "commandName": "Project",  
      "dotnetRunMessages": true,  
      "launchBrowser": false,  
      "applicationUrl": "http://localhost:5130",  
      "environmentVariables": {  
        "ASPNETCORE_ENVIRONMENT": "Development"  
      }  
    },  
    "https": {  
      "commandName": "Project",  
      "dotnetRunMessages": true,  
      "launchBrowser": false,  
      "applicationUrl": "https://localhost:7106;http://localhost:5130",  
      "environmentVariables": {  
        "ASPNETCORE_ENVIRONMENT": "Development"  
      }  
    }  
  }  
}
```

The `launchSettings.json` file above defines two *launch profiles*, `http` and `https`. Each has its own set of environment variables, launch URLs and other options. When launching a

.NET Core application developers can choose which launch profile to use.

.NET CLI

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

If no launch profile is specified, then the first launch profile is selected by default. It is possible to launch a .NET Core application without a launch profile using the `--no-launch-profile` option. Some fields from the `launchSettings.json` file are translated to environment variables. For example, the `applicationUrl` field is converted to the `ASPNETCORE_URLS` environment variable which controls which address and port ASP.NET Core binds to.

In Visual Studio it's possible to select the launch profile when launching the application making it easy to switch between configuration scenarios when manually debugging issues:



When a .NET application is launched with a launch profile a special environment variable called `DOTNET_LAUNCH_PROFILE` is populated with the name of the launch profile that was used when launching the process.

Launch profiles for .NET Aspire app host

In .NET Aspire, the AppHost is just a .NET application. As a result it has a `launchSettings.json` file just like any other application. Here is an example of the `launchSettings.json` file generated when creating a new .NET Aspire project from the starter template (`dotnet new aspire-starter`).

JSON

```
{
  "$schema": "https://json.schemastore.org/launchsettings.json",
  "profiles": {
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "applicationUrl": "https://localhost:17134;http://localhost:15170",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "DOTNET_ENVIRONMENT": "Development",
        "DOTNET_DASHBOARD_OTLP_ENDPOINT_URL": "https://localhost:21030",
        "DOTNET_RESOURCE_SERVICE_ENDPOINT_URL": "https://localhost:22057"
      }
    }
  }
}
```

```
        },
    },
    "http": {
        "commandName": "Project",
        "dotnetRunMessages": true,
        "launchBrowser": true,
        "applicationUrl": "http://localhost:15170",
        "environmentVariables": {
            "ASPNETCORE_ENVIRONMENT": "Development",
            "DOTNET_ENVIRONMENT": "Development",
            "DOTNET_DASHBOARD_OTLP_ENDPOINT_URL": "http://localhost:19240",
            "DOTNET_RESOURCE_SERVICE_ENDPOINT_URL": "http://localhost:20154"
        }
    }
}
```

The .NET Aspire templates have a very similar set of *launch profiles* to a regular ASP.NET Core application. When the .NET Aspire project launches it hosts a web-server which is used by the .NET Aspire Dashboard to fetch information about resources which are being orchestrated by .NET Aspire. There are some additional environment variables which are defined which are covered in [.NET Aspire dashboard configuration](#).

Relationship between app host launch profiles and service projects

In .NET Aspire the app host is responsible for coordinating the launch of multiple service projects. When you run the app host either via the command line or from Visual Studio (or other development environment) a launch profile is selected for the app host. In turn, the app host will attempt to find a matching launch profile in the service projects it is launching and use those options to control the environment and default networking configuration for the service project.

When the app host launches a service project it doesn't simply launch the service project using the `--launch-profile` option. Therefore, there will be no `DOTNET_LAUNCH_PROFILE` environment variable set for service projects. This is because .NET Aspire modifies the `ASPNETCORE_URLS` environment variable (derived from the `applicationUrl` field in the launch profile) to use a different port. By default, .NET Aspire inserts a reverse proxy in front of the ASP.NET Core application to allow for multiple instances of the application using the `WithReplicas` method.

Other settings such as options from the `environmentVariables` field are passed through to the application without modification.

Control launch profile selection

Ideally, it's possible to align the launch profile names between the app host and the service projects to make it easy to switch between configuration options on all projects coordinated by the app host at once. However, it may be desirable to control launch profile that a specific project uses. The [AddProject](#) extension method provides a mechanism to do this.

C#

```
var builder = DistributedApplication.CreateBuilder(args);
builder.AddProject<Projects.InventoryService>(
    "inventoryservice",
    launchProfileName: "mylaunchprofile");
```

The preceding code shows that the `inventoryservice` resource (a .NET project) is launched using the options from the `mylaunchprofile` launch profile. The launch profile precedence logic is as follows:

1. Use the launch profile specified by `launchProfileName` argument if specified.
2. Use the launch profile with the same name as the AppHost (determined by reading the `DOTNET_LAUNCH_PROFILE` environment variable).
3. Use the default (first) launch profile in `launchSettings.json`.
4. Don't use a launch profile.

To force a service project to launch without a launch profile the `launchProfileName` argument on the [AddProject](#) method can be set to null.

Launch profiles and endpoints

When adding an ASP.NET Core project to the app host, .NET Aspire will parse the `launchSettings.json` file selecting the appropriate launch profile and automatically generate endpoints in the application model based on the URL(s) present in the `applicationUrl` field. To modify the endpoints that are automatically injected the [WithEndpoint](#) extension method.

C#

```
var builder = DistributedApplication.CreateBuilder(args);
builder.AddProject<Projects.InventoryService>("inventoryservice")
    .WithEndpoint("https", endpoint => endpoint.IsProxied = false);
```

The preceding code shows how to disable the reverse proxy that .NET Aspire deploys in front for the .NET Core application and instead allows the .NET Core application to respond directly on requests over HTTP(S). For more information on networking options within .NET Aspire see [.NET Aspire inner loop networking overview](#).

See also

- [Kestrel configured endpoints](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Health checks in .NET Aspire

Article • 09/24/2024

Health checks provide availability and state information about an app. Health checks are often exposed as HTTP endpoints, but can also be used internally by the app to write logs or perform other tasks based on the current health. Health checks are typically used in combination with an external monitoring service or container orchestrator to check the status of an app. The data reported by health checks can be used for various scenarios:

- Influence decisions made by container orchestrators, load balancers, API gateways, and other management services. For instance, if the health check for a containerized app fails, it might be skipped by a load balancer routing traffic.
- Verify that underlying dependencies are available, such as a database or cache, and return an appropriate status message.
- Trigger alerts or notifications when an app isn't responding as expected.

.NET Aspire health check endpoints

.NET Aspire exposes two default health check HTTP endpoints in **Development** environments when the `AddServiceDefaults` and `MapDefaultEndpoints` methods are called from the `Program.cs` file:

- The `/health` endpoint indicates if the app is running normally where it's ready to receive requests. All health checks must pass for app to be considered ready to accept traffic after starting.

HTTP
<code>GET /health</code>

The `/health` endpoint returns an HTTP status code 200 and a `text/plain` value of `Healthy` when the app is *healthy*.

- The `/alive` indicates if an app is running or has crashed and must be restarted. Only health checks tagged with the `live` tag must pass for app to be considered alive.

HTTP

```
GET /alive
```

The `/alive` endpoint returns an HTTP status code 200 and a `text/plain` value of [Healthy](#) when the app is *alive*.

The `AddServiceDefaults` and `MapDefaultEndpoints` methods also apply various configurations to your app beyond just health checks, such as [OpenTelemetry](#) and [service discovery](#) configurations.

Non-development environments

In non-development environments, the `/health` and `/alive` endpoints are disabled by default. If you need to enable them, its recommended to protect these endpoints with various routing features, such as host filtering and/or authorization. For more information, see [Health checks in ASP.NET Core](#).

Additionally, it may be advantageous to configure request timeouts and output caching for these endpoints to prevent abuse or denial-of-service attacks. To do so, consider the following modified `AddDefaultHealthChecks` method:

C#

```
public static IApplicationBuilder AddDefaultHealthChecks(this
IApplicationBuilder builder)
{
    builder.Services.AddRequestTimeouts(
        configure: static timeouts =>
            timeouts.AddPolicy("HealthChecks", TimeSpan.FromSeconds(5)));

    builder.Services.AddOutputCache(
        configureOptions: static caching =>
            caching.AddPolicy("HealthChecks",
                build: static policy =>
                    policy.Expire(TimeSpan.FromSeconds(10))));

    builder.Services.AddHealthChecks()
        // Add a default liveness check to ensure app is responsive
        .AddCheck("self", () => HealthCheckResult.Healthy(), ["live"]);

    return builder;
}
```

The preceding code:

- Adds a timeout of 5 seconds to the health check requests with a policy named `HealthChecks`.
- Adds a 10-second cache to the health check responses with a policy named `HealthChecks`.

Now consider the updated `MapDefaultEndpoints` method:

C#

```
public static WebApplication MapDefaultEndpoints(this WebApplication app)
{
    var healthChecks = app.MapGroup("");
    healthChecks
        .CacheOutput("HealthChecks")
        .WithRequestTimeout("HealthChecks");

    // All health checks must pass for app to be
    // considered ready to accept traffic after starting
    healthChecks.MapHealthChecks("/health");

    // Only health checks tagged with the "live" tag
    // must pass for app to be considered alive
    healthChecks.MapHealthChecks("/alive", new()
    {
        Predicate = static r => r.Tags.Contains("live")
    });

    return app;
}
```

The preceding code:

- Groups the health check endpoints under the `/` path.
- Caches the output and specifies a request time with the corresponding `HealthChecks` policy.

In addition to the updated `AddDefaultHealthChecks` and `MapDefaultEndpoints` methods, you must also add the corresponding services for both request timeouts and output caching.

In the appropriate consuming app's entry point (usually the `Program.cs` file), add the following code:

C#

```
// Wherever your services are being registered.
// Before the call to Build().
```

```
builder.Services.AddRequestTimeouts();
builder.Services.AddOutputCache();

var app = builder.Build();

// Wherever your app has been built, before the call to Run().
app.UseRequestTimeouts();
app.UseOutputCache();

app.Run();
```

For more information, see [Request timeouts middleware in ASP.NET Core](#) and [Output caching middleware in ASP.NET Core](#).

Integration health checks

.NET Aspire integrations can also register additional health checks for your app. These health checks contribute to the returned status of the `/health` and `/alive` endpoints. For example, the .NET Aspire PostgreSQL integration automatically adds a health check to verify the following conditions:

- A database connection could be established
- A database query could be executed successfully

If either of these operations fail, the corresponding health check also fails.

Configure health checks

You can disable health checks for a given integration using one of the available configuration options. .NET Aspire integrations support [Microsoft.Extensions.Configurations](#) to apply settings through config files such as `appsettings.json`:

JSON

```
{
  "Aspire": {
    "Npgsql": {
      "DisableHealthChecks": true,
    }
  }
}
```

You can also use an inline delegate to configure health checks:

C#

```
builder.AddNpgsqlDbContext<MyDbContext>(
    "postgresdb",
    static settings => settings.DisableHealthChecks = true);
```

See also

- [.NET app health checks in C#](#)
- [Health checks in ASP.NET Core](#)

.NET Aspire telemetry

Article • 08/29/2024

One of the primary objectives of .NET Aspire is to ensure that apps are straightforward to debug and diagnose. .NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as the pillars of observability, using the [.NET OpenTelemetry SDK](#).

- **Logging:** Log events describe what's happening as an app runs. A baseline set is enabled for .NET Aspire integrations by default and more extensive logging can be enabled on-demand to diagnose particular problems.
- **Tracing:** Traces correlate log events that are part of the same logical activity (e.g. the handling of a single request), even if they're spread across multiple machines or processes.
- **Metrics:** Metrics expose the performance and health characteristics of an app as simple numerical values. As a result, they have low performance overhead and many services configure them as always-on telemetry. This also makes them suitable for triggering alerts when potential problems are detected.

Together, these types of telemetry allow you to gain insights into your application's behavior and performance using various monitoring and analysis tools. Depending on the backing service, some integrations may only support some of these features.

.NET Aspire OpenTelemetry integration

The [.NET OpenTelemetry SDK](#) includes features for gathering data from several .NET APIs, including [ILogger](#), [Activity](#), [Meter](#), and [Instrument<T>](#). These APIs correspond to telemetry features like logging, tracing, and metrics. .NET Aspire projects define OpenTelemetry SDK configurations in the *ServiceDefaults* project. For more information, see [.NET Aspire service defaults](#).

By default, the `ConfigureOpenTelemetry` method enables logging, tracing, and metrics for the app. It also adds exporters for these data points so they can be collected by other monitoring tools.

Export OpenTelemetry data for monitoring

The .NET OpenTelemetry SDK facilitates the export of this telemetry data to a data store or reporting tool. The telemetry export mechanism relies on the [OpenTelemetry](#)

protocol (OTLP) [🔗](#), which serves as a standardized approach for transmitting telemetry data through REST or gRPC. The `ConfigureOpenTelemetry` method also registers exporters to provide your telemetry data to other monitoring tools, such as Prometheus or Azure Monitor. For more information, see [OpenTelemetry configuration](#).

OpenTelemetry environment variables

OpenTelemetry has a [list of known environment variables](#) [🔗](#) that configure the most important behavior for collecting and exporting telemetry. OpenTelemetry SDKs, including the .NET SDK, support reading these variables.

.NET Aspire projects launch with environment variables that configure the name and ID of the app in exported telemetry and set the address endpoint of the OTLP server to export data. For example:

- `OTEL_SERVICE_NAME` = myfrontend
- `OTEL_RESOURCE_ATTRIBUTES` = service.instance.id=1a5f9c1e-e5ba-451b-95ee-ced1ee89c168
- `OTEL_EXPORTER_OTLP_ENDPOINT` = `http://localhost:4318`

The environment variables are automatically set in local development.

.NET Aspire local development

When you create a .NET Aspire project, the .NET Aspire dashboard provides a UI for viewing app telemetry by default. Telemetry data is sent to the dashboard using OTLP, and the dashboard implements an OTLP server to receive telemetry data and store it in memory. The .NET Aspire debugging workflow is as follows:

- Developer starts the .NET Aspire project with debugging, presses `F5`.
- .NET Aspire dashboard and developer control plane (DCP) start.
- App configuration is run in the *AppHost* project.
 - OpenTelemetry environment variables are automatically added to .NET projects during app configuration.
 - DCP provides the name (`OTEL_SERVICE_NAME`) and ID (`OTEL_RESOURCE_ATTRIBUTES`) of the app in exported telemetry.
 - The OTLP endpoint is an HTTP/2 port started by the dashboard. This endpoint is set in the `OTEL_EXPORTER_OTLP_ENDPOINT` environment variable on each project. That tells projects to export telemetry back to the dashboard.
 - Small export intervals (`OTEL_BSP_SCHEDULE_DELAY`, `OTEL_BLRP_SCHEDULE_DELAY`, `OTEL_METRIC_EXPORT_INTERVAL`) so data is quickly available in the dashboard.

Small values are used in local development to prioritize dashboard responsiveness over efficiency.

- The DCP starts configured projects, containers, and executables.
- Once started, apps send telemetry to the dashboard.
- Dashboard displays near real-time telemetry of all .NET Aspire projects.

All of these steps happen internally, so in most cases the developer simply needs to run the app to see this process in action.

.NET Aspire deployment

.NET Aspire deployment environments should configure OpenTelemetry environment variables that make sense for their environment. For example,

`OTEL_EXPORTER_OTLP_ENDPOINT` should be configured to the environment's local OTLP collector or monitoring service.

.NET Aspire telemetry works best in environments that support OTLP. OTLP exporting is disabled if `OTEL_EXPORTER_OTLP_ENDPOINT` isn't configured.

For more information, see [.NET Aspire deployments](#).

.NET Aspire integrations overview

Article • 09/25/2024

.NET Aspire integrations are a curated suite of NuGet packages selected to facilitate the integration of cloud-native applications with prominent services and platforms, such as Redis and PostgreSQL. Each integration furnishes essential cloud-native functionalities through either automatic provisioning or standardized configuration patterns.

💡 Tip

Always strive to use the latest version of .NET Aspire integrations to take advantage of the latest features, improvements, and security updates.

Integration responsibilities

Most .NET Aspire integrations are made up of two separate libraries, each with a different responsibility. One type represents resources within the [app host](#) project—known as [hosting integrations](#). The other type of integration represents client libraries that connect to the resources modeled by hosting integrations, and they're known as [client integrations](#).

Hosting integrations

Hosting integrations configure applications by provisioning resources (like containers or cloud resources) or pointing to existing instances (such as a local SQL server). These packages model various services, platforms, or capabilities, including caches, databases, logging, storage, and messaging systems.

Hosting integrations extend the [IDistributedApplicationBuilder](#) interface, enabling the [app host](#) project to express resources within its [app model](#). The official [hosting integration NuGet packages](#) are tagged with `aspire`, `integration`, and `hosting`.

For information on creating a custom *hosting integration*, see [Create custom .NET Aspire hosting integration](#).

Client integrations

Client integrations wire up client libraries to dependency injection (DI), define configuration schema, and add health checks, resiliency, and telemetry where applicable.

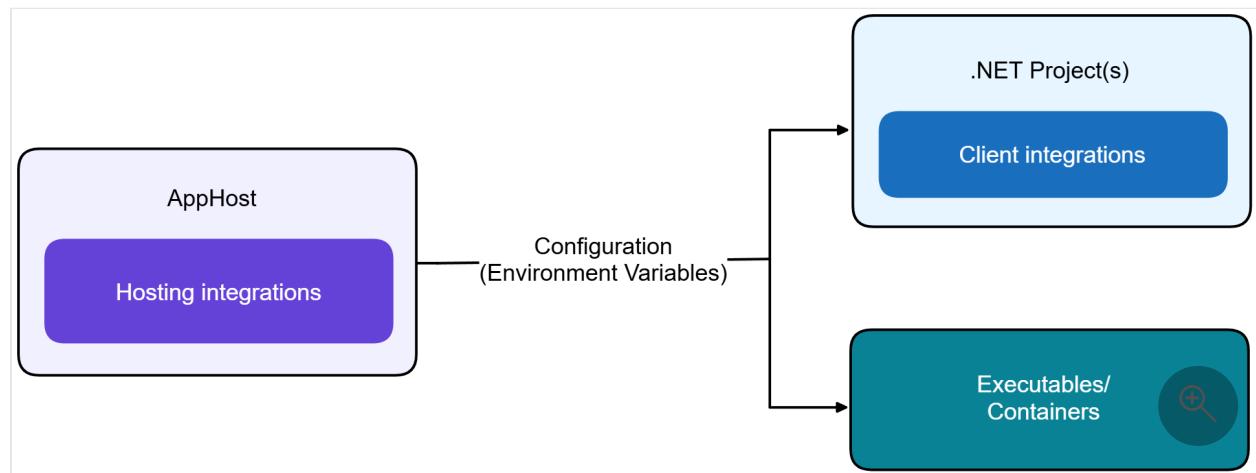
These packages configure existing client libraries to connect to hosting integrations. They extend the [IHostApplicationBuilder](#) interface allowing client-consuming projects, such as your web app or API, to use the connected resource. The official [client integration NuGet packages](#) are tagged with `aspire`, `integration`, and `client`.

For more information on creating a custom client integration, see [Create custom .NET Aspire client integrations](#).

Relationship between hosting and client integrations

Hosting and client integrations are best when used together, but are **not** coupled and can be used separately. Some hosting integrations don't have a corresponding client integration. Configuration is what makes the hosting integration work with the client integration.

Consider the following diagram that depicts the relationship between hosting and client integrations:



The app host project is where hosting integrations are used. Configuration, specifically environment variables, is injected into projects, executables, and containers, allowing client integrations to connect to the hosting integrations.

Integration features

When you add a client integration to a project within your .NET Aspire solution, [service defaults](#) are automatically applied to that project; meaning the Service Defaults project is referenced and the `AddServiceDefaults` extension method is called. These defaults are designed to work well in most scenarios and can be customized as needed. The following service defaults are applied:

- **Observability and telemetry:** Automatically sets up logging, tracing, and metrics configurations:
 - **Logging:** A technique where code is instrumented to produce logs of interesting events that occurred while the program was running.
 - **Tracing:** A specialized form of logging that helps you localize failures and performance issues within applications distributed across multiple machines or processes.
 - **Metrics:** Numerical measurements recorded over time to monitor application performance and health. Metrics are often used to generate alerts when potential problems are detected.
- **Health checks:** Exposes HTTP endpoints to provide basic availability and state information about an app. Health checks are used to influence decisions made by container orchestrators, load balancers, API gateways, and other management services.
- **Resiliency:** The ability of your system to react to failure and still remain functional. Resiliency extends beyond preventing failures to include recovering and reconstructing your cloud-native environment back to a healthy state.

Official integrations

.NET Aspire provides many integrations to help you build cloud-native applications. These integrations are designed to work seamlessly with the .NET Aspire app host and client libraries. The following sections detail cloud-agnostic, Azure-specific, and Amazon Web Services (AWS) integrations.

Cloud-agnostic integrations

The following section details cloud-agnostic .NET Aspire integrations with links to their respective docs and NuGet packages, and provides a brief description of each integration.

 Expand table

Integration docs and NuGet packages	Description
<ul style="list-style-type: none"> - Learn more:  Apache Kafka - Hosting:  Aspire.Hosting.Kafka - Client:  Aspire.Confluent.Kafka 	A library for producing and consuming messages from an Apache Kafka broker.

Integration docs and NuGet packages	Description
<ul style="list-style-type: none"> - Learn more:  Dapr - Hosting:  Aspire.Hosting.Dapr - Client: N/A 	A library for modeling Dapr as a .NET Aspire resource.
<ul style="list-style-type: none"> - Learn more:  Elasticsearch - Hosting:  Aspire.Hosting.Elasticsearch - Client:  Aspire.Elastic.Clients.Elasticsearch 	A library for accessing Elasticsearch databases.
<ul style="list-style-type: none"> - Learn more:  Keycloak - Hosting:  Aspire.Hosting.Keycloak - Client:  Aspire.Keycloak.Authentication 	A library for accessing Keycloak authentication.
<ul style="list-style-type: none"> - Learn more:  Milvus - Hosting:  Aspire.Hosting.Milvus - Client:  Aspire.Milvus.Client 	A library for accessing Milvus databases.
<ul style="list-style-type: none"> - Learn more:  MongoDB Driver - Hosting:  Aspire.Hosting.MongoDB - Client:  Aspire.MongoDB.Driver 	A library for accessing MongoDB databases.
<ul style="list-style-type: none"> - Learn more:  MySqlConnector - Hosting:  Aspire.Hosting.MySql - Client:  Aspire.MySqlConnector 	A library for accessing MySqlConnector databases.
<ul style="list-style-type: none"> - Learn more:  NATS - Hosting:  Aspire.Hosting.Nats - Client:  Aspire.NATS.Net 	A library for accessing NATS messaging.
<ul style="list-style-type: none"> - Learn more:  Oracle - EF Core - Hosting:  Aspire.Hosting.Oracle - Client:  Aspire.Oracle.EntityFrameworkCore 	A library for accessing Oracle databases with Entity Framework Core .
<ul style="list-style-type: none"> - Learn more:  Orleans - Hosting:  Aspire.Hosting.Orleans - Client: N/A 	A library for modeling Orleans as a .NET Aspire resource.
<ul style="list-style-type: none"> - Learn more:  Pomelo MySQL - EF Core - Hosting:  Aspire.Hosting.MySql - Client:  Aspire.Pomelo.EntityFrameworkCore.MySql 	A library for accessing MySql databases with Entity Framework Core .
<ul style="list-style-type: none"> - Learn more:  PostgreSQL - EF Core - Hosting:  Aspire.Hosting.PostgreSQL - Client:  Aspire.Npgsql.EntityFrameworkCore.PostgreSQL 	A library for accessing PostgreSQL databases using Entity Framework Core .
<ul style="list-style-type: none"> - Learn more:  PostgreSQL - Hosting:  Aspire.Hosting.PostgreSQL - Client:  Aspire.Npgsql 	A library for accessing PostgreSQL databases.

Integration docs and NuGet packages	Description
<ul style="list-style-type: none"> - Learn more: Qdrant - Hosting: Aspire.Hosting.Qdrant - Client: Aspire.Qdrant.Client 	A library for accessing Qdrant databases.
<ul style="list-style-type: none"> - Learn more: RabbitMQ - Hosting: Aspire.Hosting.RabbitMQ - Client: Aspire.RabbitMQ.Client 	A library for accessing RabbitMQ .
<ul style="list-style-type: none"> - Learn more: Redis Distributed Caching - Hosting: Aspire.Hosting.Redis, Aspire.Hosting.Garnet, or Aspire.Hosting.Valkey - Client: Aspire.StackExchange.Redis.DistributedCaching 	A library for accessing Redis caches for distributed caching .
<ul style="list-style-type: none"> - Learn more: Redis Output Caching - Hosting: Aspire.Hosting.Redis, Aspire.Hosting.Garnet, or Aspire.Hosting.Valkey - Client: Aspire.StackExchange.Redis.OutputCaching 	A library for accessing Redis caches for output caching .
<ul style="list-style-type: none"> - Learn more: Redis - Hosting: Aspire.Hosting.Redis, Aspire.Hosting.Garnet, or Aspire.Hosting.Valkey - Client: Aspire.StackExchange.Redis 	A library for accessing Redis caches.
<ul style="list-style-type: none"> - Learn more: Seq - Hosting: Aspire.Hosting.Seq - Client: Aspire.Seq 	A library for logging to Seq .
<ul style="list-style-type: none"> - Learn more: SQL Server - EF Core - Hosting: Aspire.Hosting.SqlServer - Client: Aspire.Microsoft.EntityFrameworkCore.SqlServer 	A library for accessing SQL Server databases using EF Core .
<ul style="list-style-type: none"> - Learn more: SQL Server - Hosting: Aspire.Hosting.SqlServer - Client: Aspire.Microsoft.Data.SqlClient 	A library for accessing SQL Server databases.

For more information on working with .NET Aspire integrations in Visual Studio, see [Visual Studio tooling](#).

Azure integrations

Azure integrations configure applications to use Azure resources. These hosting integrations are available in the `Aspire.Hosting.Azure.*` NuGet packages, while their client integrations are available in the `Aspire.*` NuGet packages:

Integration docs and NuGet packages	Description
<ul style="list-style-type: none"> - Learn more: Azure App Configuration - Hosting:  Aspire.Hosting.Azure.AppConfiguration - Client: N/A 	A library for interacting with Azure App Configuration .
<ul style="list-style-type: none"> - Learn more: Azure Application Insights - Hosting:  Aspire.Hosting.Azure.ApplicationInsights - Client: N/A 	A library for interacting with Azure Application Insights .
<ul style="list-style-type: none"> - Learn more: Azure Cosmos DB - EF Core - Hosting:  Aspire.Hosting.Azure.CosmosDB - Client:  Aspire.Microsoft.EntityFrameworkCore.Cosmos 	A library for accessing Azure Cosmos DB databases with Entity Framework Core .
<ul style="list-style-type: none"> - Learn more: Azure Cosmos DB - Hosting:  Aspire.Hosting.Azure.CosmosDB - Client:  Aspire.Microsoft.Azure.Cosmos 	A library for accessing Azure Cosmos DB databases.
<ul style="list-style-type: none"> - Learn more: Azure Event Hubs - Hosting:  Aspire.Hosting.Azure.EventHubs - Client:  Aspire.Azure.Messaging.EventHubs 	A library for accessing Azure Event Hubs .
<ul style="list-style-type: none"> - Learn more: Azure Key Vault - Hosting:  Aspire.Hosting.Azure.KeyVault - Client:  Aspire.Azure.Security.KeyVault 	A library for accessing Azure Key Vault .
<ul style="list-style-type: none"> - Learn more: Azure Operational Insights - Hosting:  Aspire.Hosting.Azure.OperationalInsights - Client: N/A 	A library for interacting with Azure Operational Insights .
<ul style="list-style-type: none"> - Learn more: Azure AI OpenAI - Hosting:  Aspire.Hosting.Azure.CognitiveServices - Client:  Aspire.Azure.AI.OpenAI 	A library for accessing Azure AI OpenAI or OpenAI functionality.
<ul style="list-style-type: none"> - Learn more: Azure PostgreSQL - Hosting:  Aspire.Hosting.Azure.PostgreSQL - Client: N/A 	A library for interacting with Azure Database for PostgreSQL .
<ul style="list-style-type: none"> - Learn more: Azure AI Search - Hosting:  Aspire.Hosting.Azure.Search - Client:  Aspire.Azure.Search.Documents 	A library for accessing Azure AI Search functionality.
<ul style="list-style-type: none"> - Learn more: Azure Service Bus - Hosting:  Aspire.Hosting.Azure.ServiceBus - Client:  Aspire.Azure.Messaging.ServiceBus 	A library for accessing Azure Service Bus .

Integration docs and NuGet packages	Description
<ul style="list-style-type: none"> - Learn more: Azure SignalR Service - Hosting: Aspire.Hosting.Azure.SignalR - Client: Microsoft.Azure.SignalR 	A library for accessing Azure SignalR Service .
<ul style="list-style-type: none"> - Learn more: Azure Blob Storage - Hosting: Aspire.Hosting.Azure.Storage - Client: Aspire.Azure.Storage.Blobs 	A library for accessing Azure Blob Storage .
<ul style="list-style-type: none"> - Learn more: Azure Storage Queues - Hosting: Aspire.Hosting.Azure.Storage - Client: Aspire.Azure.Storage.Queues 	A library for accessing Azure Storage Queues .
<ul style="list-style-type: none"> - Learn more: Azure Table Storage - Hosting: Aspire.Hosting.Azure.Storage - Client: Aspire.Azure.Data.Tables 	A library for accessing the Azure Table service .
<ul style="list-style-type: none"> - Learn more: Azure Web PubSub - Hosting: Aspire.Hosting.Azure.WebPubSub - Client: Aspire.Azure.Messaging.WebPubSub 	A library for accessing the Azure Web PubSub service .

Amazon Web Services (AWS) hosting integrations

[Expand table](#)

Integration docs and NuGet packages	Description
<ul style="list-style-type: none"> - Learn more: AWS Hosting - Hosting: Aspire.Hosting.AWS - Client: N/A 	A library for modeling AWS resources .

For more information, see [GitHub: Aspire.Hosting.AWS library](#).

Tutorial: Implement caching with .NET Aspire integrations

Article • 08/29/2024

Cloud-native apps often require various types of scalable caching solutions to improve performance. .NET Aspire integrations simplify the process of connecting to popular caching services such as Redis. In this article, you'll learn how to:

- ✓ Create a basic ASP.NET core app that is set up to use .NET Aspire.
- ✓ Add .NET Aspire integrations to connect to Redis and implement caching.
- ✓ Configure the .NET Aspire integrations to meet specific requirements.

This article explores how to use two different types of ASP.NET Core caching using .NET Aspire and Redis:

- **Output caching**: A configurable, extensible caching method for storing entire HTTP responses for future requests.
- **Distributed caching**: A cache shared by multiple app servers that allows you to cache specific pieces of data. A distributed cache is typically maintained as an external service to the app servers that access it and can improve the performance and scalability of an ASP.NET Core app.

Prerequisites

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

- [.NET 8.0](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or [the .NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

Create the project

1. At the top of Visual Studio, navigate to **File > New > Project....**
2. In the dialog window, enter **.NET Aspire** into the project template search box and select **.NET Aspire Starter Application**. Choose **Next**.
3. On the **Configure your new project** screen:

- Enter a **Project name** of **AspireRedis**.
- Leave the rest of the values at their defaults and select **Next**.

4. On the **Additional information** screen:

- Make sure **.NET 8.0** is selected.
- Uncheck **Use Redis for caching**. You will implement your own caching setup.
- Select **Create**.

Visual Studio creates a new .NET Aspire solution that consists of the following projects:

- **AspireRedis.Web** - A Blazor UI project with default .NET Aspire configurations.
- **AspireRedis.ApiService** - A Minimal API with default .NET Aspire configurations that provides the frontend with data.
- **AspireRedis.AppHost** - An orchestrator project designed to connect and configure the different projects and services of your app.
- **AspireRedis.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**.

Configure the App Host project

1. Add the [.NET Aspire Hosting Redis](#) package to the `AspireRedis.AppHost` project:

The screenshot shows the .NET CLI interface. A dropdown menu is open, with the option ".NET CLI" highlighted. Below the menu, a command-line input field contains the text: `dotnet add package Aspire.Hosting.Redis`.

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

2. Update the `Program.cs` file of the `AspireRedis.AppHost` project to match the following code:

C#

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

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

var apiservice = builder.AddProject<Projects.AspireRedis_ApiService>("apiservice")
    .WithReference(redis);

builder.AddProject<Projects.AspireRedis_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(apiservice)
    .WithReference(redis);

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

The preceding code creates a local Redis container instance and configures the UI and API to use the instance automatically for both output and distributed caching. The code also configures communication between the frontend UI and the backend API using service discovery. With .NET Aspire's implicit service discovery, setting up and managing service connections is streamlined for developer productivity. In the context of this tutorial, the feature simplifies how you connect to Redis.

Traditionally, you'd manually specify the Redis connection string in each project's *appsettings.json* file:

JSON

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

Configuring connection string with this method, while functional, requires duplicating the connection string across multiple projects, which can be cumbersome and error-prone.

Configure the UI with output caching

1. Add the [.NET Aspire Stack Exchange Redis output caching](#) integration packages to your `AspireRedis.Web` app:

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.OutputCaching
```

2. In the `Program.cs` file of the `AspireRedis.Web` Blazor project, immediately after the line `var builder = WebApplication.CreateBuilder(args);`, add a call to the [AddRedisOutputCache](#) extension method:

C#

```
builder.AddRedisOutputCache("cache");
```

This method accomplishes the following tasks:

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

3. Replace the contents of the `Home.razor` file of the `AspireRedis.Web` Blazor project with the following:

razor

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

<PageTitle>Home</PageTitle>

<h1>Hello, world!</h1>

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

The integration include the `[OutputCache]` attribute, which caches the entire rendered response. The page also include a call to `@DateTime.Now` to help verify that the response is cached.

Configure the API with distributed caching

1. Add the [.NET Aspire Stack Exchange Redis distributed caching](#) integration packages to your `AspireRedis.ApiService` app:

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.DistributedCaching
```

2. Towards the top of the `Program.cs` file, add a call to [AddRedisDistributedCache](#):

C#

```
builder.AddRedisDistributedCache("cache");
```

3. In the *Program.cs* file, replace the existing `/weatherforecast` endpoint code with the following:

C#

```
app.MapGet("/weatherforecast", async (IDistributedCache cache) =>
{
    var cachedForecast = await cache.GetAsync("forecast");

    if (cachedForecast is null)
    {
        var summaries = new[] { "Freezing", "Bracing", "Chilly",
" Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching" };
        var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        (
            DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            Random.Shared.Next(-20, 55),
            summaries[Random.Shared.Next(summaries.Length)])
        ))
        .ToArray();

        await cache.SetAsync("forecast",
Encoding.UTF8.GetBytes(JsonSerializer.Serialize(forecast)), new ()
        {
            AbsoluteExpiration = DateTime.Now.AddSeconds(10)
        });
    }

    return forecast;
}

return JsonSerializer.Deserialize<IEnumerable<WeatherForecast>>(cachedForecast);
})
.WithName("GetWeatherForecast");
```

Test the app locally

Test the caching behavior of your app using the following steps:

1. Run the app using Visual Studio by pressing `F5`.
2. If the **Start Docker Desktop** dialog appears, select **Yes** to start the service.
3. The .NET Aspire Dashboard loads in the browser and lists the UI and API projects.

Test the output cache:

1. On the projects page, in the **webfrontend** row, click the `localhost` link in the **Endpoints** column to open the UI of your app.
2. The application will display the current time on the home page.
3. Refresh the browser every few seconds to see the same page returned by output caching. After 10 seconds the cache expires and the page reloads with an updated time.

Test the distributed cache:

1. Navigate to the **Weather** page on the Blazor UI to load a table of randomized weather data.
2. Refresh the browser every few seconds to see the same weather data returned by output caching. After 10 seconds the cache expires and the page reloads with updated weather data.

Congratulations! You configured a ASP.NET Core app to use output and distributed caching with .NET Aspire.

Tutorial: Connect an ASP.NET Core app to SQL Server using .NET Aspire and Entity Framework Core

Article • 09/10/2024

In this tutorial, you create an ASP.NET Core app that uses a .NET Aspire Entity Framework Core SQL Server integration to connect to SQL Server to read and write support ticket data. [Entity Framework Core](#) is a lightweight, extensible, open source object-relational mapper that enables .NET developers to work with databases using .NET objects. You'll learn how to:

- ✓ Create a basic .NET app that is set up to use .NET Aspire integrations
- ✓ Add a .NET Aspire integration to connect to SQL Server
- ✓ Configure and use .NET Aspire Component features to read and write from the database

Prerequisites

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

- [.NET 8.0](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or [the .NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

Create the sample solution

1. At the top of Visual Studio, navigate to **File > New > Project**.
2. In the dialog window, search for *Blazor* and select **Blazor Web App**. Choose **Next**.
3. On the **Configure your new project** screen:
 - Enter a **Project Name** of **AspireSQLEFCore**.

- Leave the rest of the values at their defaults and select **Next**.

4. On the **Additional information** screen:

- Make sure **.NET 8.0** is selected.
- Ensure the **Interactive render mode** is set to **None**.
- Check the **Enlist in .NET Aspire orchestration** option and select **Create**.

Visual Studio creates a new ASP.NET Core solution that is structured to use .NET Aspire. The solution consists of the following projects:

- **AspireSQLEFCore**: A Blazor project that depends on service defaults.
- **AspireSQLEFCore.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.
- **AspireSQLEFCore.ServiceDefaults**: A shared class library to hold configurations that can be reused across the projects in your solution.

Create the database model and context classes

To represent a user submitted support request, add the following `SupportTicket` model class at the root of the *AspireSQLEFCore* project.

```
C#  
  
using System.ComponentModel.DataAnnotations;  
  
namespace AspireSQLEFCore;  
  
public sealed class SupportTicket  
{  
    public int Id { get; set; }  
    [Required]  
    public string Title { get; set; } = string.Empty;  
    [Required]  
    public string Description { get; set; } = string.Empty;  
}
```

Add the following `TicketDbContext` data context class at the root of the *AspireSQLEFCore* project. The class inherits `System.Data.Entity.DbContext` to work with Entity Framework and represent your database.

```
C#  
  
using Microsoft.EntityFrameworkCore;  
using System.Reflection.Metadata;
```

```
namespace AspireSQLEFCore;

public class TicketContext(DbContextOptions options) : DbContext(options)
{
    public DbSet<SupportTicket> Tickets => Set<SupportTicket>();
}
```

Add the .NET Aspire integration to the Blazor app

Add the [.NET Aspire Entity Framework Core Sql Server library package](#) to your *AspireSQLEFCore* project:

.NET CLI

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

Your *AspireSQLEFCore* project is now set up to use .NET Aspire integrations. Here's the updated *AspireSQLEFCore.csproj* file:

XML

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

    <PropertyGroup>
        <TargetFramework>net8.0</TargetFramework>
        <Nullable>enable</Nullable>
        <ImplicitUsings>enable</ImplicitUsings>
    </PropertyGroup>

    <ItemGroup>
        <PackageReference
Include="Aspire.Microsoft.EntityFrameworkCore.SqlServer" Version="8.2.0" />
    </ItemGroup>

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

</Project>
```

Configure the .NET Aspire integration

In the `Program.cs` file of the `AspireSQLEFCore` project, add a call to the `AddSqlServerDbContext` extension method after the creation of the `builder` but before the call to `AddServiceDefaults`. For more information, see [.NET Aspire service defaults](#). Provide the name of your connection string as a parameter.

C#

```
using AspireSQLEFCore;
using AspireSQLEFCore.Components;

var builder = WebApplication.CreateBuilder(args);
builder.AddSqlServerDbContext<TicketContext>("sqldata");

builder.AddServiceDefaults();

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

var app = builder.Build();

app.MapDefaultEndpoints();
```

This method accomplishes the following tasks:

- Registers a `TicketContext` with the DI container for connecting to the containerized Azure SQL Database.
- Automatically enable corresponding health checks, logging, and telemetry.

Create the database

While developing locally, you need to create a database inside the SQL Server container. Update the `Program.cs` file with the following code:

C#

```
using AspireSQLEFCore;
using AspireSQLEFCore.Components;

var builder = WebApplication.CreateBuilder(args);
builder.AddSqlServerDbContext<TicketContext>("sqldata");

builder.AddServiceDefaults();

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

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

```
app.MapDefaultEndpoints();

if (app.Environment.IsDevelopment())
{
    using (var scope = app.Services.CreateScope())
    {
        var context =
scope.ServiceProvider.GetRequiredService<TicketContext>();
        context.Database.EnsureCreated();
    }
}
else
{
    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();
}
```

The preceding code:

- Checks if the app is running in a development environment.
- If it is, it retrieves the `TicketContext` service from the DI container and calls `Database.EnsureCreated()` to create the database if it doesn't already exist.

!**Note**

Note that `EnsureCreated()` is not suitable for production environments, and it only creates the database as defined in the context. It doesn't apply any migrations. For more information on Entity Framework Core migrations in .NET Aspire, see [Apply Entity Framework Core migrations in .NET Aspire](#).

Create the form

The app requires a form for the user to be able to submit support ticket information and save the entry to the database.

Use the following Razor markup to create a basic form, replacing the contents of the `Home.razor` file in the `AspireSQLEFCore/Components/Pages` directory:

razor

```
@page "/"
@inject TicketContext context
```

```

<div class="row">
    <div class="col-md-6">
        <div>
            <h1 class="display-4">Request Support</h1>
        </div>
        <EditForm Model="@Ticket" FormName="Tickets" method="post"
                  OnValidSubmit="@HandleValidSubmit" class="mb-4">
            <DataAnnotationsValidator />
            <div class="mb-4">
                <label>Issue Title</label>
                <InputText class="form-control" @bind-Value="@Ticket.Title" />
            </div>
            <ValidationMessage For="() => Ticket.Title" />
            <div class="mb-4">
                <label>Issue Description</label>
                <InputText class="form-control" @bind-Value="@Ticket.Description" />
            </div>
            <ValidationMessage For="() => Ticket.Description" />
            <button class="btn btn-primary" type="submit">Submit</button>
            <button class="btn btn-danger mx-2" type="reset" @onclick=@ClearForm>Clear</button>
        </EditForm>

        <table class="table table-striped">
            @foreach (var ticket in Tickets)
            {
                <tr>
                    <td>@ticket.Id</td>
                    <td>@ticket.Title</td>
                    <td>@ticket.Description</td>
                </tr>
            }
        </table>
    </div>
</div>

@code {
    [SupplyParameterFromForm]
    private SupportTicket Ticket { get; set; } = new();
    private List<SupportTicket> Tickets = [];

    private void ClearForm() => Ticket = new();

    protected override async Task OnInitializedAsync()
    {
        Tickets = await context.Tickets.ToListAsync();
    }

    private async Task HandleValidSubmit()
    {
        context.Tickets.Add(Ticket);
        await context.SaveChangesAsync();
        Tickets = await context.Tickets.ToListAsync();
    }
}

```

```
    }  
}
```

For more information about creating forms in Blazor, see [ASP.NET Core Blazor forms overview](#).

Configure the AppHost

The *AspireSQUEFCore.AppHost* project is the orchestrator for your app. It's responsible for connecting and configuring the different projects and services of your app. The orchestrator should be set as the startup project.

Add the [.NET Aspire Hosting Sql Server](#) NuGet package to your *AspireStorage.AppHost* project:

.NET CLI

```
dotnet add package Aspire.Hosting.SqlServer
```

Replace the contents of the *Program.cs* file in the *AspireSQUEFCore.AppHost* project with the following code:

C#

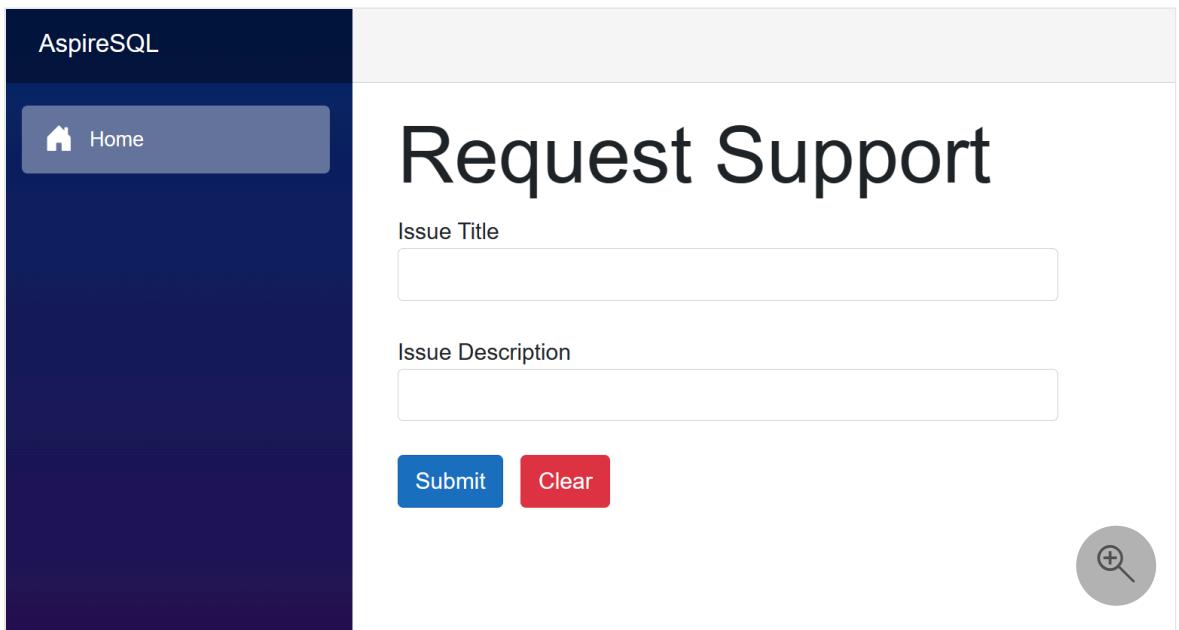
```
var builder = DistributedApplication.CreateBuilder(args);  
  
var sql = builder.AddSqlServer("sql")  
    .AddDatabase("sqldata");  
  
builder.AddProject<Projects.AspireSQUEFCore>("aspiresql")  
    .WithReference(sql);  
  
builder.Build().Run();
```

The preceding code adds a SQL Server Container resource to your app and configures a connection to a database called `sql`. The Entity Framework classes you configured earlier will automatically use this connection when migrating and connecting to the database.

Run and test the app locally

The sample app is now ready for testing. Verify that the submitted form data is persisted to the database by completing the following steps:

1. Select the run button at the top of Visual Studio (or **F5**) to launch your .NET Aspire project dashboard in the browser.
2. On the projects page, in the **AspireSQLEFCore** row, click the link in the **Endpoints** column to open the UI of your app.



3. Enter sample data into the **Title** and **Description** form fields.
4. Select the **Submit** button, and the form submits the support ticket for processing — (then select **Clear** to clear the form).
5. The data you submitted displays in the table at the bottom of the page when the page reloads.

See also

- [.NET Aspire with SQL Database deployment](#)
- [.NET Aspire deployment via Azure Container Apps](#)
- [Deploy a .NET Aspire project using GitHub Actions](#)

Tutorial: Connect an ASP.NET Core app to .NET Aspire storage integrations

Article • 08/29/2024

Cloud-native apps often require scalable storage solutions that provide capabilities like blob storage, queues, or semi-structured NoSQL databases. .NET Aspire integrations simplify connections to various storage services, such as Azure Blob Storage. In this tutorial, you'll create an ASP.NET Core app that uses .NET Aspire integrations to connect to Azure Blob Storage and Azure Queue Storage to submit support tickets. The app sends the tickets to a queue for processing and uploads an attachment to storage. You'll learn how to:

- ✓ Create a basic .NET app that is set up to use .NET Aspire integrations
- ✓ Add .NET Aspire integrations to connect to multiple storage services
- ✓ Configure and use .NET Aspire Component features to send and receive data

Prerequisites

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

- [.NET 8.0](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or [the .NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

Explore the completed sample app

A completed version of the sample app from this tutorial is available on GitHub. The project is also structured as a template for the [Azure Developer CLI](#), meaning you can use the `azd up` command to automate Azure resource provisioning if you have the tool [installed](#).

Bash

```
git clone https://github.com/Azure-Samples/dotnet-aspire-connect-storage.git
```

Set up the Azure Storage resources

For this article, you'll need to create a blob container and storage queue resource in your local development environment using an emulator. To do so, use Azurite. Azurite is a free, open source, cross-platform Azure Storage API compatible server (emulator) that runs in a Docker container.

To use the emulator you need to [install Azurite](#).

Create the sample solution

Create a .NET Aspire project using either Visual Studio or the .NET CLI.

Visual Studio

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 Application**. Choose **Next**.
3. On the **Configure your new project** screen:
 - Enter a **Solution Name of AspireStorage** and select **Next**.
4. On the **Additional information** screen:
 - Uncheck **Use Redis for caching** (not required for this tutorial).
 - Select **Create**.

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

The solution consists of the following projects:

- **AspireStorage.ApiService** - An API project with default .NET Aspire service configurations.
- **AspireStorage.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.

- **AspireStorage.ServiceDefaults** - A shared class library to hold code that can be reused across the projects in your solution.
- **AspireStorage.Web** - A Blazor Server project that serves as the front end of your app.

Add the Worker Service project

Next, add a Worker Service project to the solution to retrieve and process messages as they are added to the Azure Storage queue.

Visual Studio

1. In the solution explorer, right click on the top level *AspireStorage* solution node and select **Add > New project**.
2. Search for and select the **Worker Service** template and choose **Next**.
3. For the **Project name**, enter *AspireStorage.WorkerService* and select **Next**.
4. On the **Additional information** screen:
 - Make sure **.NET 8.0** is selected.
 - Make sure **Enlist in .NET Aspire orchestration** is checked and select **Create**.

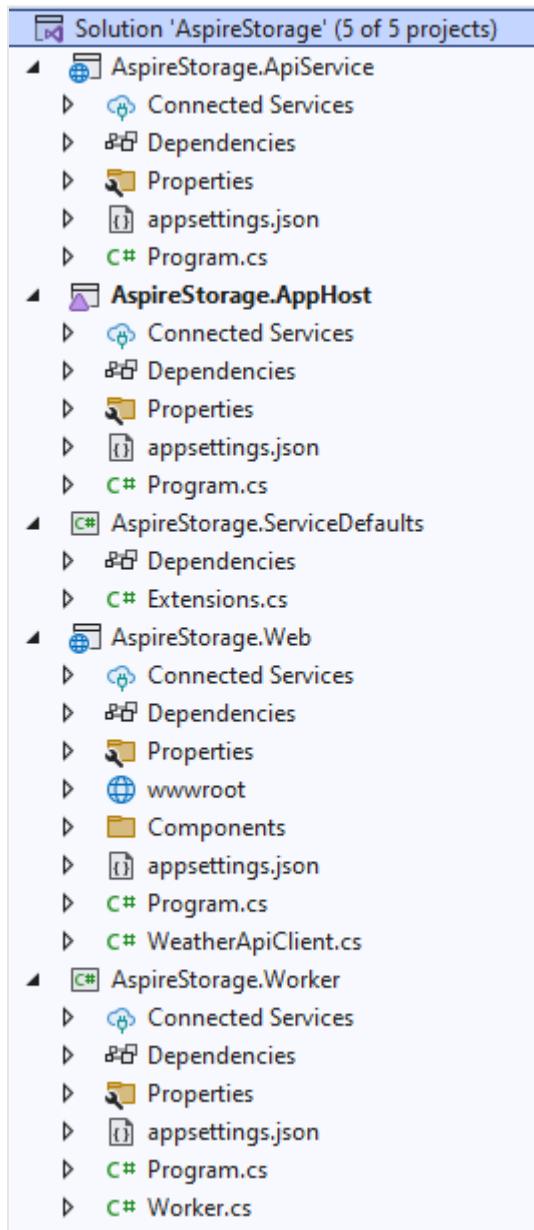
Visual Studio adds the project to your solution and updates the *Program.cs* file of the *AspireStorage.AppHost* project with a new line of code:

C#

```
builder.AddProject<Projects.AspireStorage_WorkerService>(  
    "aspirestorage-workerservice");
```

Visual Studio tooling added this line of code to register your new project with the **IDistributedApplicationBuilder** object, which enables orchestration features. For more information, see [.NET Aspire orchestration overview](#).

The completed solution structure should resemble the following:



Add the .NET Aspire integrations to the Blazor app

Add the [.NET Aspire Azure Blob Storage integration](#) and [.NET Aspire Azure Queue Storage integration](#) packages to your *AspireStorage.Web* project:

.NET CLI

```
dotnet add package Aspire.Azure.Storage.Blobs  
dotnet add package Aspire.Azure.Storage.Queues
```

Your *AspireStorage.Web* project is now set up to use .NET Aspire integrations. Here's the updated *AspireStorage.Web.csproj* file:

XML

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

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

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

  <ItemGroup>
    <PackageReference Include="Aspire.Azure.Storage.Blobs" Version="8.2.0"
    />
    <PackageReference Include="Aspire.Azure.Storage.Queues" Version="8.2.0"
    />
  </ItemGroup>

</Project>
```

The next step is to add the integrations to the app.

In the *Program.cs* file of the *AspireStorage.Web* project, add calls to the [AddAzureBlobClient](#) and [AddAzureQueueClient](#) extension methods after the creation of the `builder` but before the call to `AddServiceDefaults`. For more information, see [.NET Aspire service defaults](#). Provide the name of your connection string as a parameter.

C#

```
using AspireStorage.Web;
using AspireStorage.Web.Components;

using Azure.Storage.Blobs;
using Azure.Storage.Queues;

var builder = WebApplication.CreateBuilder(args);

builder.AddAzureBlobClient("BlobConnection");
builder.AddAzureQueueClient("QueueConnection");

// Add service defaults & Aspire components.
builder.AddServiceDefaults();

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

builder.Services.AddOutputCache();
```

```

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();
}
else
{
    // In development, create the blob container and queue if they don't
    // exist.
    var blobService = app.Services.GetRequiredService<BlobServiceClient>();
    var docsContainer = blobService.GetBlobContainerClient("fileuploads");

    await docsContainer.CreateIfNotExistsAsync();

    var queueService = app.Services.GetRequiredService<QueueServiceClient>();
    var queueClient = queueService.GetQueueClient("tickets");

    await queueClient.CreateIfNotExistsAsync();
}

app.UseHttpsRedirection();

app.UseStaticFiles();
app.UseAntiforgery();

app.UseOutputCache();

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

app.MapDefaultEndpoints();

app.Run();

```

With the additional `using` statements, these methods accomplish the following tasks:

- Register a `Azure.Storage.Blobs.BlobServiceClient` and a `Azure.Storage.Queues.QueueServiceClient` with the DI container for connecting to

Azure Storage.

- Automatically enable corresponding health checks, logging, and telemetry for the respective services.

When the *AspireStorage.Web* project starts, it will create a `fileuploads` container in Azurite Blob Storage and a `tickets` queue in Azurite Queue Storage. This is conditional when the app is running in a development environment. When the app is running in a production environment, the container and queue are assumed to have already been created.

Add the .NET Aspire integration to the Worker Service

The worker service handles pulling messages off of the Azure Storage queue for processing. Add the [.NET Aspire Azure Queue Storage integration](#) integration package to your *AspireStorage.WorkerService* app:

.NET CLI

```
dotnet add package Aspire.Azure.Storage.Queues
```

In the *Program.cs* file of the *AspireStorage.WorkerService* project, add a call to the [AddAzureQueueClient](#) extension method after the creation of the `builder` but before the call to `AddServiceDefaults`:

C#

```
using AspireStorage.WorkerService;

var builder = Host.CreateApplicationBuilder(args);

builder.AddAzureQueueClient("QueueConnection");

builder.AddServiceDefaults();
builder.Services.AddHostedService<WorkerService>();

var host = builder.Build();
host.Run();
```

This method handles the following tasks:

- Register a [QueueServiceClient](#) with the DI container for connecting to Azure Storage Queues.

- Automatically enable corresponding health checks, logging, and telemetry for the respective services.

Create the form

The app requires a form for the user to be able to submit support ticket information and upload an attachment. The app uploads the attached file on the `Document` (`IFormFile`) property to Azure Blob Storage using the injected `BlobServiceClient`. The `QueueServiceClient` sends a message composed of the `Title` and `Description` to the Azure Storage Queue.

Use the following Razor markup to create a basic form, replacing the contents of the `Home.razor` file in the `AspireStorage.Web/Components/Pages` directory:

razor

```
@page "/"

@using System.ComponentModel.DataAnnotations
@using Azure.Storage.Blobs
@using Azure.Storage.Queues

@inject BlobServiceClient BlobClient
@inject QueueServiceClient QueueServiceClient

<PageTitle>Home</PageTitle>

<div class="text-center">
    <h1 class="display-4">Request Support</h1>
</div>

<EditForm Model="@Ticket" FormName="Tickets" method="post"
          OnValidSubmit="@HandleValidSubmit" enctype="multipart/form-data">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <div class="mb-4">
        <label>Issue Title</label>
        <InputText class="form-control" @bind-Value="@Ticket.Title" />
        <ValidationMessage For="() => Ticket.Title" />
    </div>
    <div class="mb-4">
        <label>Issue Description</label>
        <InputText class="form-control" @bind-Value="@Ticket.Description" />
        <ValidationMessage For="() => Ticket.Description" />
    </div>
    <div class="mb-4">
        <label>Attachment</label>
        <InputFile class="form-control" name="Ticket.Document" />
        <ValidationMessage For="() => Ticket.Document" />
    </div>
</EditForm>
```

```

        </div>
        <button class="btn btn-primary" type="submit">Submit</button>
        <button class="btn btn-danger mx-2" type="reset">
@onclick=@ClearForm>Clear</button>
</EditForm>

@code {
    [SupplyParameterFromForm(FormName = "Tickets")]
    private SupportTicket Ticket { get; set; } = new();

    private async Task HandleValidSubmit()
    {
        var docsContainer =
BlobClient.GetBlobContainerClient("fileuploads");

        // Upload file to blob storage
        await docsContainer.UploadBlobAsync(
            Ticket.Document.FileName,
            Ticket.Document.OpenReadStream());

        // Send message to queue
        var queueClient = QueueServiceClient.GetQueueClient("tickets");

        await queueClient.SendMessageAsync(
            $"{Ticket.Title} - {Ticket.Description}");

        ClearForm();
    }

    private void ClearForm() => Ticket = new();

    private class SupportTicket()
    {
        [Required] public string Title { get; set; } = default!;
        [Required] public string Description { get; set; } = default!;
        [Required] public IFormFile Document { get; set; } = default!;
    }
}

```

For more information about creating forms in Blazor, see [ASP.NET Core Blazor forms overview](#).

Update the AppHost

The *AspireStorage.AppHost* project is the orchestrator for your app. It's responsible for connecting and configuring the different projects and services of your app. The orchestrator should be set as the startup project.

To add Azure Storage hosting support to your [IDistributedApplicationBuilder](#), install the [Aspire.Hosting.Azure.Storage](#) NuGet package.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Storage
```

Replace the contents of the *Program.cs* file in the *AspireStorage.AppHost* project with the following code:

C#

```
using Microsoft.Extensions.Hosting;

var builder = DistributedApplication.CreateBuilder(args);

var storage = builder.AddAzureStorage("Storage");

if (builder.Environment.IsDevelopment())
{
    storage.RunAsEmulator();
}

var blobs = storage.AddBlobs("BlobConnection");
var queues = storage.AddQueues("QueueConnection");

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

builder.AddProject<Projects.AspireStorage_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(apiService)
    .WithReference(blobs)
    .WithReference(queues);

builder.AddProject<Projects.AspireStorage_WorkerService>("aspirestorage-
workerservice")
    .WithReference(queues);

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

The preceding code adds Azure storage, blobs, and queues, and when in development mode, it uses the emulator. Each project defines references for these resources that they depend on.

Process the items in the queue

When a new message is placed on the `tickets` queue, the worker service should retrieve, process, and delete the message. Update the `Worker.cs` class, replacing the contents with the following code:

C#

```
using Azure.Storage.Queues;
using Azure.Storage.Queues.Models;

namespace AspireStorage.WorkerService;

public sealed class WorkerService(
    QueueServiceClient client,
    ILogger<WorkerService> logger) : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        var queueClient = client.GetQueueClient("tickets");
        await queueClient.CreateIfNotExistsAsync(cancellationToken:
stoppingToken);

        while (!stoppingToken.IsCancellationRequested)
        {
            QueueMessage[] messages =
                await queueClient.ReceiveMessagesAsync(
                    maxMessages: 25, cancellationToken: stoppingToken);

            foreach (var message in messages)
            {
                logger.LogInformation(
                    "Message from queue: {Message}", message.MessageText);

                await queueClient.DeleteMessageAsync(
                    message.MessageId,
                    message.PopReceipt,
                    cancellationToken: stoppingToken);
            }

            // TODO: Determine an appropriate time to wait
            // before checking for more messages.
            await Task.Delay(TimeSpan.FromSeconds(15), stoppingToken);
        }
    }
}
```

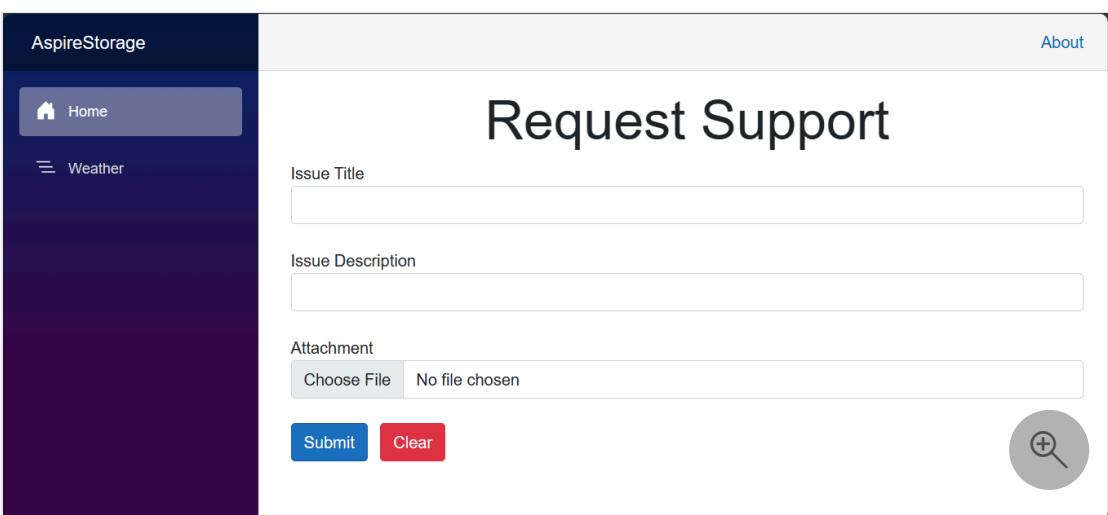
Before the worker service can process messages, it needs to be able to connect to the Azure Storage queue. With Azurite, you need to ensure that the queue is available before the worker service starts executing message queue processing.

The worker service processes message in the queue and deletes them when they've been processed.

Run and test the app locally

The sample app is now ready for testing. Verify that the submitted form data is sent to Azure Blob Storage and Azure Queue Storage by completing the following steps:

Visual Studio



The screenshot shows a web application interface titled "AspireStorage". On the left, there's a sidebar with "Home" and "Weather" options. The main content area has a title "Request Support". It contains three input fields: "Issue Title", "Issue Description", and "Attachment" (with a "Choose File" button). Below these are two buttons: "Submit" (blue) and "Clear" (red). In the bottom right corner of the main area, there's a circular icon with a magnifying glass and a plus sign.

1. Press the run button at the top of Visual Studio to launch your .NET Aspire project dashboard in the browser.
2. On the resources page, in the **aspirestorage.web** row, click the link in the **Endpoints** column to open the UI of your app.
3. Enter sample data into the **Title** and **Description** form fields and select a simple file to upload.
4. Select the **Submit** button, and the form submits the support ticket for processing — and clears the form.
5. In a separate browser tab, use the Azure portal to navigate to the **Storage browser** in your Azure Storage Account.
6. Select **Containers** and then navigate into the **Documents** container to see the uploaded file.
7. You can verify the message on the queue was processed by looking at the **Project logs** of the [.NET Aspire dashboard](#), and selecting the **aspirestorage.workerservice** from the dropdown.

```
150     x-ms-return-client-request-id:true
151     User-Agent:azsdk-net-Storage.Queues/12.17.1 (.NET 8.0.5; Microsoft Windows 10.0.22631)
152     x-ms-date:Wed, 22 May 2024 23:08:18 GMT
153     Authorization:REDACTED
154     client assembly: Azure.Storage.Queues
155     2024-05-22T18:08:18.237003Z info: Azure.Core[5]
156     Response [e09e3c30-0f9c-47ef-8e4c-adef240a4440] 200 OK (00.0s)
157     Server:Azurite-Queue/3.29.0
158     x-ms-client-request-id:e09e3c30-0f9c-47ef-8e4c-adef240a4440
159     x-ms-request-id:3bae1286-b828-4526-bb37-9ed1bcaa2001
160     x-ms-version:2024-02-04
161     Date:Wed, 22 May 2024 23:08:18 GMT
162     Connection:keep-alive
163     Keep-Alive:REDACTED
164     Transfer-Encoding:chunked
165     Content-Type:application/xml
166
167 2024-05-22T18:08:18.237003Z info: AspireStorage.WorkerService[0]
168     Message from queue: I need help! - Everything is broken and nothing works!
169 2024-05-22T18:08:18.265108Z info: Azure.Core[1]
170     Request [9f9a3bdf-553e-43b6-a9a0-fdc85cc3be3e] DELETE http://127.0.0.1:64355/devstoreaccount1/tickets/messages/2d0b6f31-f9c8-44af-8fd5-1987c7b76903?popreceipt=MjJ
NYXkyMDI0MjM0MDQg0Htgz0DBL
171     x-ms-version:2018-11-09
172     Accept:application/xml
173     x-ms-client-request-id:9f9a3bdf-553e-43b6-a9a0-fdc85cc3be3e
174     x-ms-return-client-request-id:true
175     User-Agent:azsdk-net-Storage.Queues/12.17.1 (.NET 8.0.5; Microsoft Windows 10.0.22631)
176     x-ms-date:Wed, 22 May 2024 23:08:18 GMT
177     Authorization:REDACTED
178     client assembly: Azure.Storage.Queues
179 2024-05-22T18:08:18.284352Z info: Azure.Core[5]
180     Response [9f9a3bdf-553e-43b6-a9a0-fdc85cc3be3e] 204 No Content (00.0s)
181     Server:Azurite-Queue/3.29.0
```

Summary

The example app that you built demonstrates persisting blobs from an ASP.NET Core Blazor Web App and processing queues in a [.NET Worker Service](#). Your app connects to Azure Storage using .NET Aspire integrations. The app sends the support tickets to a queue for processing and uploads an attachment to storage.

Since you choose to use Azurite, there's no need to clean up these resources when you're done testing them, as you created them locally in the context of an emulator. The emulator enabled you to test your app locally without incurring any costs, as no Azure resources were provisioned or created.

Apply Entity Framework Core migrations in .NET Aspire

Article • 08/02/2024

Since .NET Aspire projects use a containerized architecture, databases are ephemeral and can be recreated at any time. Entity Framework Core (EF Core) uses a feature called [migrations](#) to create and update database schemas. Since databases are recreated when the app starts, you need to apply migrations to initialize the database schema each time your app starts. This is accomplished by registering a migration service project in your app that runs migrations during startup.

In this tutorial, you learn how to configure .NET Aspire projects to run EF Core migrations during app startup.

Prerequisites

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

- [.NET 8.0](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or [the .NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

Obtain the starter app

This tutorial uses a sample app that demonstrates how to apply EF Core migrations in .NET Aspire. Use Visual Studio to clone [the sample app from GitHub](#) or use the following command:

Bash

```
git clone https://github.com/MicrosoftDocs/aspire-docs-samples/
```

The sample app is in the *SupportTicketApi* folder. Open the solution in Visual Studio or VS Code and take a moment to review the sample app and make sure it runs before proceeding. The sample app is a rudimentary support ticket API, and it contains the following projects:

- **SupportTicketApi.Api**: The ASP.NET Core project that hosts the API.
- **SupportTicketApi.Data**: Contains the EF Core contexts and models.
- **SupportTicketApi.AppHost**: Contains the .NET Aspire app host and configuration.
- **SupportTicketApi.ServiceDefaults**: Contains the default service configurations.

Run the app to ensure it works as expected. From the .NET Aspire dashboard, select the **https** Swagger endpoint and test the API's **GET /api/SupportTickets** endpoint by expanding the operation and selecting **Try it out**. Select **Execute** to send the request and view the response:

```
JSON

[  
  {  
    "id": 1,  
    "title": "Initial Ticket",  
    "description": "Test ticket, please ignore."  
  }  
]
```

Create migrations

Start by creating some migrations to apply.

1. Open a terminal (**ctrl**+**Shift**+**T** in Visual Studio).
2. Set *SupportTicketApi*/*SupportTicketApi.Api* as the current directory.
3. Use the **dotnet ef command-line tool** to create a new migration to capture the initial state of the database schema:

```
.NET CLI

dotnet ef migrations add InitialCreate --project
..\SupportTicketApi.Data\SupportTicketApi.Data.csproj
```

The proceeding command:

- Runs EF Core migration command-line tool in the *SupportTicketApi.Api* directory. **dotnet ef** is run in this location because the API service is where

- the DB context is used.
- Creates a migration named *InitialCreate*.
 - Creates the migration in the in the *Migrations* folder in the *SupportTicketApi.Data* project.

4. Modify the model so that it includes a new property. Open *SupportTicketApi.DataModelsSupportTicket.cs* and add a new property to the `SupportTicket` class:

C#

```
public sealed class SupportTicket
{
    public int Id { get; set; }
    [Required]
    public string Title { get; set; } = string.Empty;
    [Required]
    public string Description { get; set; } = string.Empty;
    public bool Completed { get; set; }
}
```

5. Create another new migration to capture the changes to the model:

.NET CLI

```
dotnet ef migrations add AddCompleted --project
..\SupportTicketApi.Data\SupportTicketApi.Data.csproj
```

Now you've got some migrations to apply. Next, you'll create a migration service that applies these migrations during app startup.

Create the migration service

To run the migrations at startup, you need to create a service that applies the migrations.

1. Add a new Worker Service project to the solution. If using Visual Studio, right-click the solution in Solution Explorer and select **Add > New Project**. Select **Worker Service** and name the project *SupportTicketApi.MigrationService*. If using the command line, use the following commands from the solution directory:

.NET CLI

```
dotnet new worker -n SupportTicketApi.MigrationService
```

```
dotnet sln add SupportTicketApi.MigrationService
```

2. Add the *SupportTicketApi.Data* and *SupportTicketApi.ServiceDefaults* project references to the *SupportTicketApi.MigrationService* project using Visual Studio or the command line:

.NET CLI

```
dotnet add SupportTicketApi.MigrationService reference  
SupportTicketApi.Data  
dotnet add SupportTicketApi.MigrationService reference  
SupportTicketApi.ServiceDefaults
```

3. Add the [Aspire.Microsoft.EntityFrameworkCore.SqlServer](#) NuGet package reference to the *SupportTicketApi.MigrationService* project using Visual Studio or the command line:

.NET CLI

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

4. Add the highlighted lines to the *Program.cs* file in the *SupportTicketApi.MigrationService* project:

C#

```
using SupportTicketApi.Data.Contexts;  
using SupportTicketApi.MigrationService;  
  
var builder = Host.CreateApplicationBuilder(args);  
  
builder.AddServiceDefaults();  
builder.Services.AddHostedService<Worker>();  
  
builder.Services.AddOpenTelemetry()  
    .WithTracing(tracing =>  
    tracing.AddSource(Worker.ActivitySourceName));  
  
builder.AddSqlServerDbContext<TicketContext>("sqldata");  
  
var host = builder.Build();  
host.Run();
```

In the preceding code:

- The `AddServiceDefaults` extension method adds service defaults functionality.

- The `AddOpenTelemetry` extension method [configures OpenTelemetry functionality](#).
- The `AddSqlServerDbContext` extension method adds the `TicketContext` service to the service collection. This service is used to run migrations and seed the database.

5. Replace the contents of the `Worker.cs` file in the `SupportTicketApi.MigrationService` project with the following code:

C#

```
using System.Diagnostics;

using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.Storage;

using OpenTelemetry.Trace;

using SupportTicketApi.Data.Contexts;
using SupportTicketApi.Data.Models;

namespace SupportTicketApi.MigrationService;

public class Worker(
    IServiceProvider serviceProvider,
    IHostApplicationLifetime hostApplicationLifetime) :
BackgroundService
{
    public const string ActivitySourceName = "Migrations";
    private static readonly ActivitySource s_activitySource =
new(ActivitySourceName);

    protected override async Task ExecuteAsync(CancellationToken cancellationToken)
    {
        using var activity = s_activitySource.StartActivity("Migrating
database", ActivityKind.Client);

        try
        {
            using var scope = serviceProvider.CreateScope();
            var dbContext =
scope.ServiceProvider.GetRequiredService<TicketContext>();

            await EnsureDatabaseAsync(dbContext, cancellationToken);
            await RunMigrationAsync(dbContext, cancellationToken);
            await SeedDataAsync(dbContext, cancellationToken);
        }
        catch (Exception ex)
        {
            activity?.RecordException(ex);
        }
    }
}
```

```
        throw;
    }

    hostApplicationLifetime.StopApplication();
}

private static async Task EnsureDatabaseAsync(TicketContext dbContext, CancellationToken cancellationToken)
{
    var dbCreator =
dbContext.GetService<IRelationalDatabaseCreator>();

    var strategy = dbContext.Database.CreateExecutionStrategy();
    await strategy.ExecuteAsync(async () =>
{
    // Create the database if it does not exist.
    // Do this first so there is then a database to start a
transaction against.
    if (!await dbCreator.ExistsAsync(cancellationToken))
    {
        await dbCreator.CreateAsync(cancellationToken);
    }
});
}

private static async Task RunMigrationAsync(TicketContext dbContext, CancellationToken cancellationToken)
{
    var strategy = dbContext.Database.CreateExecutionStrategy();
    await strategy.ExecuteAsync(async () =>
{
    // Run migration in a transaction to avoid partial
migration if it fails.
    await using var transaction = await
dbContext.Database.BeginTransactionAsync(cancellationToken);
    await dbContext.Database.MigrateAsync(cancellationToken);
    await transaction.CommitAsync(cancellationToken);
});
}

private static async Task SeedDataAsync(TicketContext dbContext,
CancellationToken cancellationToken)
{
    SupportTicket firstTicket = new()
    {
        Title = "Test Ticket",
        Description = "Default ticket, please ignore!",
        Completed = true
    };

    var strategy = dbContext.Database.CreateExecutionStrategy();
    await strategy.ExecuteAsync(async () =>
{
    // Seed the database
    await using var transaction = await
```

```
dbContext.Database.BeginTransactionAsync(cancellationToken);
    await dbContext.Tickets.AddAsync(firstTicket,
cancellationToken);
    await dbContext.SaveChangesAsync(cancellationToken);
    await transaction.CommitAsync(cancellationToken);
}
}
```

In the preceding code:

- The `ExecuteAsync` method is called when the worker starts. It in turn performs the following steps:
 - a. Gets a reference to the `TicketContext` service from the service provider.
 - b. Calls `EnsureDatabaseAsync` to create the database if it doesn't exist.
 - c. Calls `RunMigrationAsync` to apply any pending migrations.
 - d. Calls `SeedDataAsync` to seed the database with initial data.
 - e. Stops the worker with `StopApplication`.
- The `EnsureDatabaseAsync`, `RunMigrationAsync`, and `SeedDataAsync` methods all encapsulate their respective database operations using execution strategies to handle transient errors that may occur when interacting with the database. To learn more about execution strategies, see [Connection Resiliency](#).

Add the migration service to the orchestrator

The migration service is created, but it needs to be added to the .NET Aspire app host so that it runs when the app starts.

1. In the `SupportTicketApi.AppHost` project, open the `Program.cs` file.
2. Add the following highlighted code to the `ConfigureServices` method:

C#

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

var sql = builder.AddSqlServer("sql")
    .AddDatabase("sqldata");

builder.AddProject<Projects.SupportTicketApi_Api>("api")
    .WithReference(sql);

builder.AddProject<Projects.SupportTicketApi_MigrationService>
("migrations")
    .WithReference(sql);
```

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

This enlists the *SupportTicketApi.MigrationService* project as a service in the .NET Aspire app host.

ⓘ Important

If you are using Visual Studio, and you selected the **Enlist in Aspire orchestration** option when creating the Worker Service project, similar code is added automatically with the service name `supportticketapi-migrationservice`. Replace that code with the preceding code.

Remove existing seeding code

Since the migration service seeds the database, you should remove the existing data seeding code from the API project.

1. In the *SupportTicketApi.Api* project, open the *Program.cs* file.
2. Delete the highlighted lines.

C#

```
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();

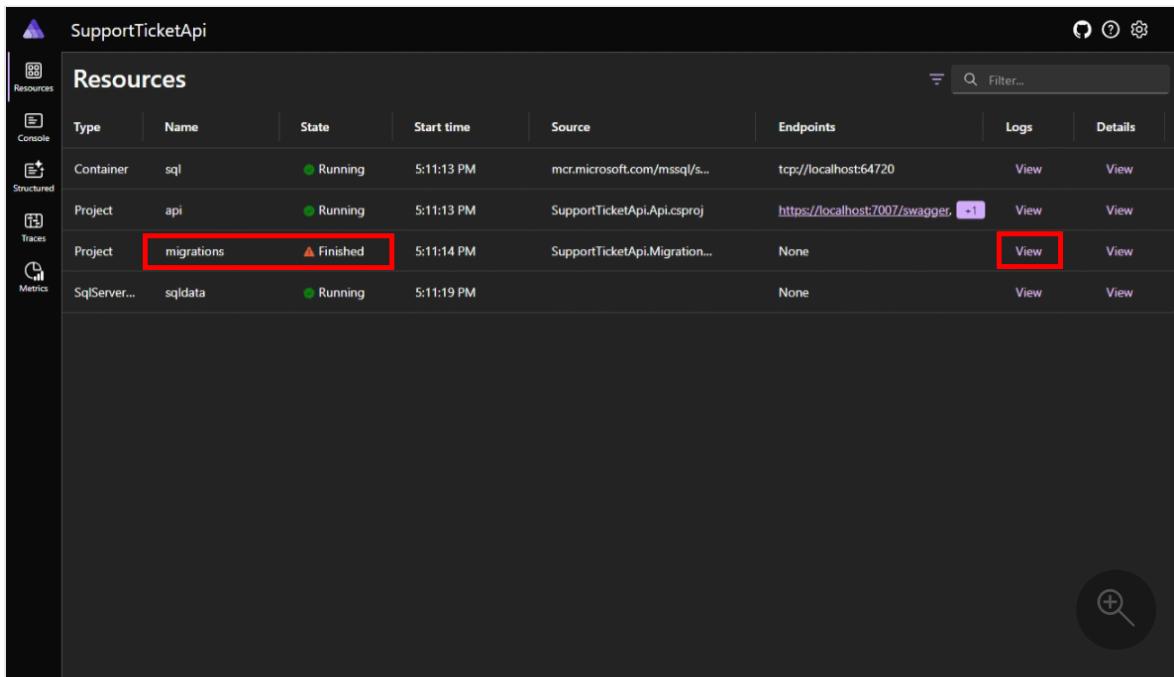
    using (var scope = app.Services.CreateScope())
    {
        var context =
            scope.ServiceProvider.GetRequiredService<TicketContext>();
        context.Database.EnsureCreated();

        if (!context.Tickets.Any())
        {
            context.Tickets.Add(new SupportTicket { Title = "Initial
Ticket", Description = "Test ticket, please ignore." });
            context.SaveChanges();
        }
    }
}
```

Test the migration service

Now that the migration service is configured, run the app to test the migrations.

1. Run the app and observe the SupportTicketApi dashboard.
2. After a short wait, the `migrations` service state will display **Finished**.



Type	Name	State	Start time	Source	Endpoints	Logs	Details
Container	sql	Running	5:11:13 PM	mcr.microsoft.com/mssql/s...	tcp://localhost:64720	View	View
Project	api	Running	5:11:13 PM	SupportTicketApi.Api.csproj	https://localhost:7007/swagger	View	View
Project	migrations	⚠️ Finished	5:11:14 PM	SupportTicketApi.Migration...	None	View	View
SqlServer...	sqldata	Running	5:11:19 PM		None	View	View

3. Select the **View** link on the migration service to investigate the logs showing the SQL commands that were executed.

Get the code

You can find the [completed sample app on GitHub](#).

More sample code

The [Aspire Shop](#) sample app uses this approach to apply migrations. See the `AspireShop.CatalogDbManager` project for the migration service implementation.

Seed data in a database using .NET Aspire

Article • 08/12/2024

In this article, you learn how to configure .NET Aspire projects to seed data in a database during app startup. .NET Aspire enables you to seed data using database scripts or Entity Framework Core for common platforms such as SQL Server, PostgreSQL and MySQL.

When to seed data

Seeding data pre-populates database tables with rows of data so they're ready for testing via your app. You may want to seed data for the following scenarios:

- Manually develop and test different features of your app against a meaningful set of data, such as a product catalog or list of customers.
- Run test suites to verify that features behave a specific way with a given set of data.

Manually seeding data is tedious and time consuming, so you should automate the process when possible. Use volumes to run database scripts for .NET Aspire projects during startup. You can also seed your database using tools like Entity Framework Core, which handles many underlying concerns for you.

Understand containerized databases

By default, .NET Aspire database integrations rely on containerized databases, which create the following challenges when trying to seed data:

- .NET Aspire destroys and recreates containers every time the app restarts, which means by default you have to re-seed your database every time the app restarts.
- Depending on your selected database technology, the new container instance may or may not create a default database, which means you might also have to create the database itself.
- Even if a default database exists, it most likely will not have the desired name or schema for your specific app.

.NET Aspire enables you to resolve these challenges using volumes and a few configurations to seed data effectively.

Seed data using volumes and SQL scripts

Volumes are the recommended way to automatically seed containerized databases when using SQL scripts. Volumes can store data for multiple containers at a time, offer high performance and are easy to back up or migrate. With .NET Aspire, you configure a volume for each resource container using the [ContainerResourceBuilderExtensions.WithBindMount](#) method, which accepts three parameters:

- **Source:** The source path of the volume mount, which is the physical location on your host.
- **Target:** The target path in the container of the data you want to persist.

Consider the following volume configuration code from a *Program.cs* file in a sample **AppHost** project:

C#

```
var todosDbName = "Todos";
var todosDb = builder.AddPostgres("postgres")
    .WithEnvironment("POSTGRES_DB", todosDbName)
    .WithBindMount(
        "../DatabaseContainers.ApiService/data/postgres",
        "/docker-entrypoint-initdb.d")
    .AddDatabase(todosDbName);
```

In this example, the `.WithBindMount` method parameters configure the following:

- `../DatabaseContainers.ApiService/data/postgres` sets a path to the SQL script in your local project that you want to run in the container to seed data.
- `/docker-entrypoint-initdb.d` sets the path to an entry point in the container so your script will be run during container startup.

The referenced SQL script located at `../DatabaseContainers.ApiService/data/postgres` creates and seeds a `Todos` table:

SQL

```
-- Postgres init script

-- Create the Todos table
CREATE TABLE IF NOT EXISTS Todos
(
    Id SERIAL PRIMARY KEY,
    Title text UNIQUE NOT NULL,
    IsComplete boolean NOT NULL DEFAULT false
```

```
);

-- Insert some sample data into the Todos table
INSERT INTO Todos (Title, IsComplete)
VALUES
    ('Give the dog a bath', false),
    ('Wash the dishes', false),
    ('Do the groceries', false)
ON CONFLICT DO NOTHING;
```

The script runs during startup every time a new container instance is created.

Database seeding examples

The following examples demonstrate how to seed data using SQL scripts and configurations applied using the `.WithBindMount` method for different database technologies:

! Note

Visit the [Database Container Sample App](#) to view the full project and file structure for each database option.

SQL Server

The configuration code in the `.AppHost Program.cs` file mounts the required database files and folders and configures an endpoint so that they run during startup.

C#

```
// SQL Server container is configured with an auto-generated password by
default
// but doesn't support any auto-creation of databases or running scripts
on startup so we have to do it manually.
var addressBookDb = builder.AddSqlServer("sqlserver")
    // Mount the init scripts directory into the container.
    .WithBindMount("./sqlserverconfig", "/usr/config")
    // Mount the SQL scripts directory into the container so that the
init scripts run.
    .WithBindMount("../DatabaseContainers.ApiService/data/sqlserver",
"/docker-entrypoint-initdb.d")
    // Run the custom endpoint script on startup.
    .WithEntrypoint("/usr/config/entrypoint.sh")
    // Add the database to the application model so that it can be
```

```
referenced by other resources.  
.AddDatabase("AddressBook");
```

The *entrypoint.sh* script lives in the mounted `./sqlserverconfig` project folder and runs when the container starts. The script launches SQL Server and checks that it's running.

shell

```
#!/bin/bash  
  
# Adapted from: https://github.com/microsoft/mssql-docker/blob/80e2a51d0eb1693f2de014fb26d4a414f5a5add5/linux/preview/examples/mssql-customize/entrypoint.sh  
  
# Start the script to create the DB and user  
/usr/config/configure-db.sh &  
  
# Start SQL Server  
/opt/mssql/bin/sqlservr
```

The *init.sql* SQL script that lives in the mounted `../DatabaseContainers.ApiService/data/sqlserver` project folder creates the database and tables.

SQL

```
-- SQL Server init script  
  
-- Create the AddressBook database  
IF NOT EXISTS (SELECT * FROM sys.databases WHERE name = N'AddressBook')  
BEGIN  
    CREATE DATABASE AddressBook;  
END;  
GO  
  
USE AddressBook;  
GO  
  
-- Create the Contacts table  
IF OBJECT_ID(N'Contacts', N'U') IS NULL  
BEGIN  
    CREATE TABLE Contacts  
    (  
        Id      INT PRIMARY KEY IDENTITY(1,1) ,  
        FirstName VARCHAR(255) NOT NULL,  
        LastName  VARCHAR(255) NOT NULL,  
        Email    VARCHAR(255) NULL,  
        Phone    VARCHAR(255) NULL  
    );
```

```

END;
GO

-- Ensure that either the Email or Phone column is populated
IF OBJECT_ID(N'chk_Contacts_Email_Phone', N'C') IS NULL
BEGIN
    ALTER TABLE Contacts
    ADD CONSTRAINT chk_Contacts_Email_Phone CHECK
    (
        Email IS NOT NULL OR Phone IS NOT NULL
    );
END;
GO

-- Insert some sample data into the Contacts table
IF (SELECT COUNT(*) FROM Contacts) = 0
BEGIN
    INSERT INTO Contacts (FirstName, LastName, Email, Phone)
    VALUES
        ('John', 'Doe', 'john.doe@example.com', '555-123-4567'),
        ('Jane', 'Doe', 'jane.doe@example.com', '555-234-5678');
END;
GO

```

Seed data using Entity Framework Core

You can also seed data in .NET Aspire projects using Entity Framework Core by explicitly running migrations during startup. Entity Framework Core handles underlying database connections and schema creation for you, which eliminates the need to use volumes or run SQL scripts during container startup.

ⓘ Important

These types of configurations should only be done during development, so make sure to add a conditional that checks your current environment context.

Add the following code to the *Program.cs* file of your API Service project.

SQL Server

C#

```

// Register DbContext class
builder.AddSqlServerDbContext<TicketContext>("sqldata");

var app = builder.Build();

```

```
app.MapDefaultEndpoints();

if (app.Environment.IsDevelopment())
{
    // Retrieve an instance of the DbContext class and manually run
    // migrations during startup
    using (var scope = app.Services.CreateScope())
    {
        var context =
scope.ServiceProvider.GetRequiredService<TicketContext>();
        context.Database.Migrate();
    }
}
```

Next steps

Database seeding is useful in a variety of app development scenarios. Try combining these techniques with the resource implementations demonstrated in the following tutorials:

- [Tutorial: Connect an ASP.NET Core app to SQL Server using .NET Aspire and Entity Framework Core](#)
- [Tutorial: Connect an ASP.NET Core app to .NET Aspire storage integrations](#)
- [.NET Aspire orchestration overview](#)

Tutorial: Use .NET Aspire messaging integrations in ASP.NET Core

Article • 08/29/2024

Cloud-native apps often require scalable messaging solutions that provide capabilities such as messaging queues and topics and subscriptions. .NET Aspire integrations simplify the process of connecting to various messaging providers, such as Azure Service Bus. In this tutorial, you'll create an ASP.NET Core app that uses .NET Aspire integrations to connect to Azure Service Bus to create a notification system. Submitted messages will be sent to a Service Bus topic for consumption by subscribers. You'll learn how to:

- ✓ Create a basic .NET app that is set up to use .NET Aspire integrations
- ✓ Add an .NET Aspire integration to connect to Azure Service Bus
- ✓ Configure and use .NET Aspire integration features to send and receive data

Prerequisites

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

- [.NET 8.0](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or [the .NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

In addition to the preceding prerequisites, you also need to install the Azure CLI. To install the Azure CLI, follow the instructions in the [Azure CLI installation guide](#).

Set up the Azure Service Bus account

For this tutorial, you'll need access to an Azure Service Bus namespace with a topic and subscription configured. Use one of the following options to set up the required resources:

- Azure portal: [Create a service bus account with a topic and subscription](#).

Alternatively:

- Azure CLI: Run the following commands in the Azure CLI or CloudShell to set up the required Azure Service Bus resources:

Azure CLI

```
az group create -n <your-resource-group-name> --location eastus
az servicebus namespace create -g <your-resource-group-name> --name
<your-namespace-name> --location eastus
az servicebus topic create -g <your-resource-group-name> --namespace-
name <your-namespace-name> --name notifications
az servicebus topic subscription create -g <your-resource-group-name> -
--namespace-name <your-namespace-name> --topic-name notifications --name
mobile
```

ⓘ Note

Replace the **your-resource-group-name** and **your-namespace-name** placeholders with your own values. Service Bus namespace names must be globally unique across Azure.

Azure authentication

This quickstart can be completed using either passwordless authentication or a connection string. Passwordless connections use Azure Active Directory and role-based access control (RBAC) to connect to a Service Bus namespace. You don't need to worry about having hard-coded connection string in your code, a configuration file, or in secure storage such as Azure Key Vault.

You can also use a connection string to connect to a Service Bus namespace, but the passwordless approach is recommended for real-world applications and production environments. For more information, read about [Authentication and authorization](#) or visit the passwordless [overview page](#).

Passwordless (Recommended)

On your Service Bus namespace, assign the following role to the user account you logged into Visual Studio or the Azure CLI with:

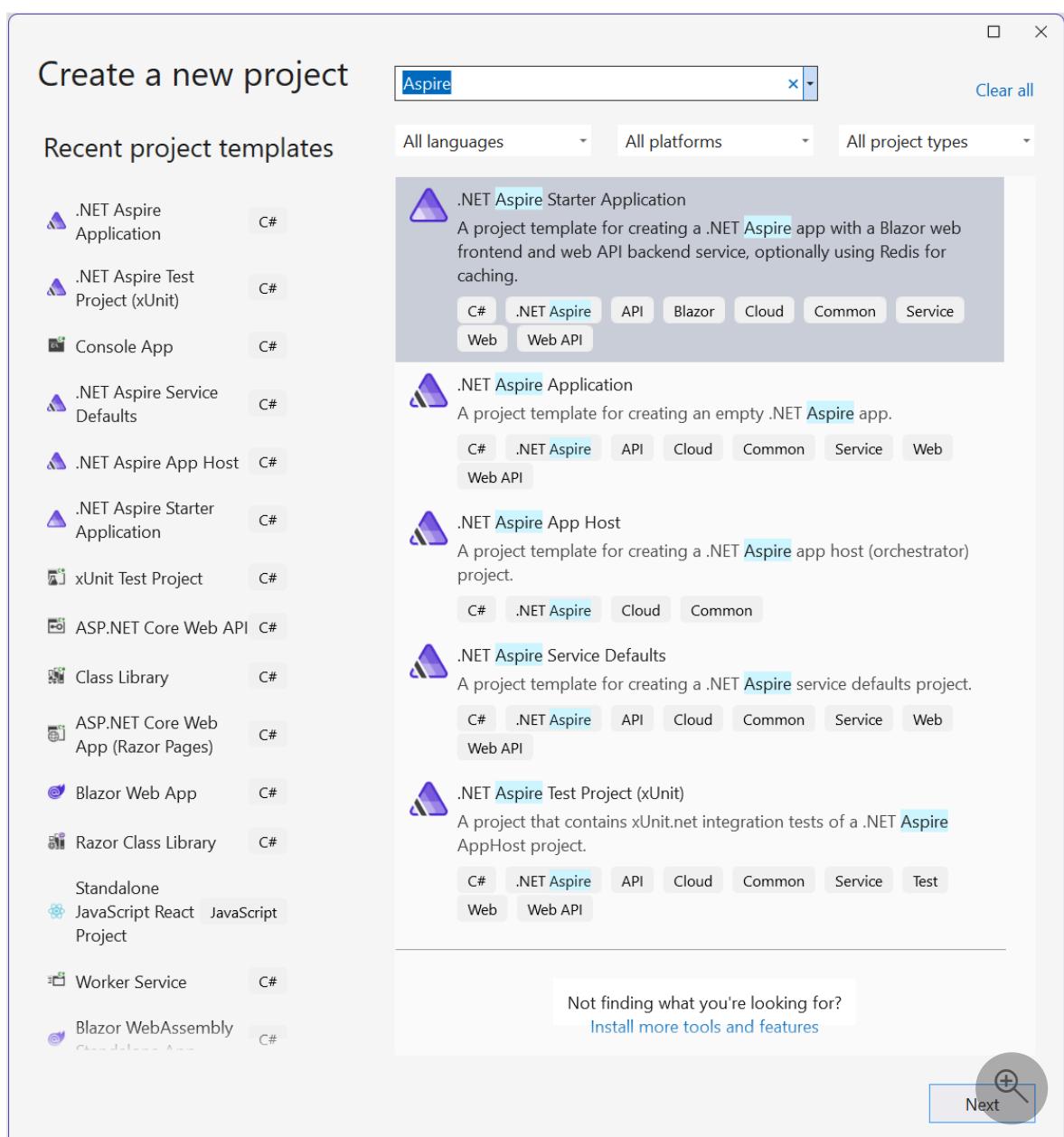
- Service Bus Data Owner: [Assign an Azure RBAC role](#)

Create the sample solution

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 project 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 Application**. 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 8.0 (Long 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 [Testing .NET Aspire projects](#).

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

Add the Worker Service project

Next, add a Worker Service project to the solution to retrieve and process messages to and from Azure Service Bus.

1. In the solution explorer, right click on the top level `AspireSample` solution node and select **Add > New project**.
2. Search for and select the **Worker Service** template and choose **Next**.
3. For the **Project name**, enter *AspireSample.WorkerService* and select **Next**.
4. On the **Additional information** screen:

- Make sure **.NET 8.0** is selected.
- Make sure **Enlist in .NET Aspire orchestration** is checked and select **Create**.

Visual Studio adds the project to your solution and updates the *Program.cs* file of the `AspireSample.AppHost` project with a new line of code:

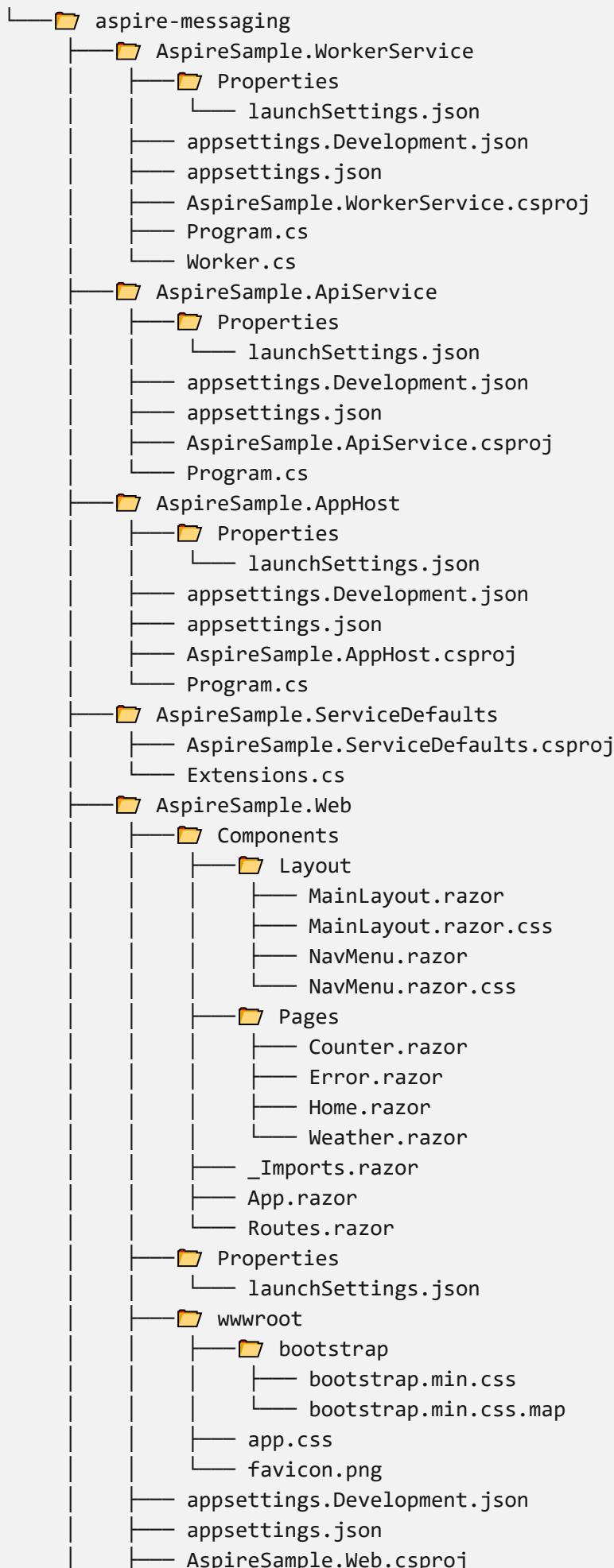
C#

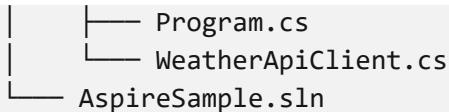
```
builder.AddProject<Projects.AspireSample_WorkerService>(  
    "aspiresample-workerservice");
```

Visual Studio tooling added this line of code to register your new project with the [IDistributedApplicationBuilder](#) object, which enables orchestration features you'll explore later.

The completed solution structure should resemble the following, assuming the top-level directory is named *aspire-messaging*:

Directory





Add the .NET Aspire integration to the API

Add the [.NET Aspire Azure Service Bus](#) integration to your **AspireSample.ApiService** app:

1. In the **Solution Explorer**, double-click the **AspireSample.ApiService.csproj** file to open its XML file.
2. Add the following `<PackageReference>` item to the `<ItemGroup>` element:

XML

```
<ItemGroup>
  <PackageReference Include="Aspire.Azure.Messaging.ServiceBus"
    Version="8.0.1" />
</ItemGroup>
```

In the *Program.cs* file of the **AspireSample.ApiService** project, add a call to the `AddAzureServiceBusClient` extension method—replacing the existing call to `AddServiceDefaults`:

C#

```
// Add service defaults & Aspire integrations.
builder.AddServiceDefaults();
builder.AddAzureServiceBusClient("serviceBusConnection");
```

For more information, see [AddAzureServiceBusClient](#).

This method accomplishes the following tasks:

- Registers a `ServiceBusClient` with the DI container for connecting to Azure Service Bus.
- Automatically enables corresponding health checks, logging, and telemetry for the respective services.

In the *appsettings.json* file of the same project, add the corresponding connection information:

JSON

```
{  
    // Existing configuration is omitted for brevity.  
    "ConnectionStrings": {  
        "serviceBusConnection": "{your_namespace}.servicebus.windows.net"  
    }  
}
```

⚠ Note

Make sure to replace `{your_namespace}` in the service URIs with the name of your own Service Bus namespace.

Create the API endpoint

The API must provide an endpoint to receive data and publish it to the Service Bus topic and broadcast to subscribers. Add the following endpoint to the **AspireSample.ApiService** project to send a message to the Service Bus topic. Replace all of the contents of the *Program.cs* file with the following C# code:

C#

```
using Azure.Messaging.ServiceBus;  
  
var builder = WebApplication.CreateBuilder(args);  
  
// Add service defaults & Aspire integrations.  
builder.AddServiceDefaults();  
builder.AddAzureServiceBusClient("serviceBusConnection");  
  
// Add services to the container.  
builder.Services.AddProblemDetails();  
  
var app = builder.Build();  
  
// Configure the HTTP request pipeline.  
app.UseExceptionHandler();  
  
app.MapPost("/notify", static async (ServiceBusClient client, string  
message) =>  
{  
    var sender = client.CreateSender("notifications");  
  
    // Create a batch  
    using ServiceBusMessageBatch messageBatch =
```

```

        await sender.CreateMessageBatchAsync();

        if (messageBatch.TryAddMessage(
            new ServiceBusMessage($"Message {message}")) is false)
    {
        // If it's too large for the batch.
        throw new Exception(
            $"The message {message} is too large to fit in the batch.");
    }

    // Use the producer client to send the batch of
    // messages to the Service Bus topic.
    await sender.SendMessagesAsync(messageBatch);

    Console.WriteLine($"A message has been published to the topic.");
});

app.MapDefaultEndpoints();

app.Run();

```

Add the .NET Aspire integration to the Worker Service

Add the [.NET Aspire Azure Service Bus](#) integration to your `AspireSample.WorkerService` project. Follow the same steps as you did before when you added the [Aspire.Azure.Messaging.ServiceBus](#) NuGet package to the `AspireSample.ApiService` project. Once it's been added, you can configure the worker service to process messages from the Service Bus topic.

In the `Program.cs` file of the `AspireSample.WorkerService` project, replace the existing code with the following:

C#

```

using AspireSample.WorkerService;

var builder = Host.CreateApplicationBuilder(args);

builder.AddAzureServiceBusClient("serviceBusConnection");

builder.Services.AddHostedService<Worker>();

var host = builder.Build();
host.Run();

```

The `AddAzureServiceBusClient` method accomplishes the following tasks:

- Registers a `ServiceBusClient` with the DI container for connecting to Azure Service Bus.
- Automatically enables corresponding health checks, logging, and telemetry for the respective services.

In the `appsettings.json` file of the `AspireSample.WorkerService` project, add the corresponding connection information:

```
 Passwordless (Recommended) JSON

{
    // Existing configuration is omitted for brevity.
    "ConnectionStrings": {
        "serviceBusConnection": "{your_namespace}.servicebus.windows.net"
    }
}
```

(!) Note

Make sure to replace `{your_namespace}` in the Service URIs with the name of your own Service Bus namespace.

Process the message from the subscriber

When a new message is placed on the `messages` queue, the worker service should retrieve, process, and delete the message. Update the `Worker.cs` class to match the following code:

```
C# using Azure.Messaging.ServiceBus;

namespace AspireSample.WorkerService;

public sealed class Worker(
    ILogger<Worker> logger,
    ServiceBusClient client) : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
```

```
    {
        var processor = client.CreateProcessor(
            "notifications",
            "mobile",
            new ServiceBusProcessorOptions());

        // Add handler to process messages
        processor.ProcessMessageAsync += MessageHandler;

        // Add handler to process any errors
        processor.ProcessErrorAsync += ErrorHandler;

        // Start processing
        await processor.StartProcessingAsync();

        logger.LogInformation(""""
            Wait for a minute and then press any key to end the
processing
        """);

        Console.ReadKey();

        // Stop processing
        logger.LogInformation(""""

            Stopping the receiver...
        """);

        await processor.StopProcessingAsync();

        logger.LogInformation("Stopped receiving messages");
    }
}

async Task MessageHandler(ProcessMessageEventArgs args)
{
    string body = args.Message.Body.ToString();

    logger.LogInformation("Received: {Body} from subscription.", body);

    // Complete the message. messages is deleted from the subscription.
    await args.CompleteMessageAsync(args.Message);
}

// Handle any errors when receiving messages
Task ErrorHandler(ProcessErrorEventArgs args)
{
    logger.LogError(args.Exception, "{Error}", args.Exception.Message);

    return Task.CompletedTask;
}
}
```

Run and test the app locally

The sample app is now ready for testing. Verify that the data submitted to the API is sent to the Azure Service Bus topic and consumed by the subscriber worker service:

1. Launch the .NET Aspire project by selecting the **Start** debugging button, or by pressing **F5**. The .NET Aspire dashboard app should open in the browser.
2. On the resources page, in the **apiservice** row, find the link in the **Endpoints** that opens the `weatherforecast` endpoint. Note the HTTPS port number.
3. On the .NET Aspire dashboard, navigate to the logs for the **aspiresample-workerservice** project.
4. In a terminal window, use the `curl` command to send a test message to the API:

Bash

```
curl -X POST -H "Content-Type: application/json" https://localhost:  
{port}/notify?message=hello%20aspire
```

Be sure to replace `{port}` with the port number from earlier.

5. Switch back to the **aspiresample-workerservice** logs. You should see the test message printed in the output logs.

Congratulations! You created and configured an ASP.NET Core API that connects to Azure Service Bus using Aspire integrations.

Clean up resources

Run the following Azure CLI command to delete the resource group when you no longer need the Azure resources you created. Deleting the resource group also deletes the resources contained inside of it.

Azure CLI

```
az group delete --name <your-resource-group-name>
```

For more information, see [Clean up resources in Azure](#).

.NET Aspire Apache Kafka integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Apache Kafka client message-broker. The `Aspire.Confluent.Kafka` library registers an [IProducer<TKey, TValue>](#) and an [IConsumer<TKey, TValue>](#) in the dependency injection (DI) container for connecting to a Apache Kafka server. It enables corresponding health check, logging and telemetry.

Get started

To get started with the .NET Aspire Apache Kafka integration, install the [Aspire.Confluent.Kafka](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Apache Kafka client.

```
.NET CLI
.NET CLI
dotnet add package Aspire.Confluent.Kafka
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the [AddKafkaProducer](#) extension method to register an `IProducer<TKey, TValue>` for use via the dependency injection container. The method takes two generic parameters corresponding to the type of the key and the type of the message to send to the broker. These generic parameters will be used to new an instance of `ProducerBuilder<TKey, TValue>`. This method also take connection name parameter.

```
C#
builder.AddKafkaProducer<string, string>("messaging");
```

You can then retrieve the `IProducer<TKey, TValue>` instance using dependency injection. For example, to retrieve the producer from an `IHostedService`:

C#

```
internal sealed class MyWorker(IProducer<string, string> producer) :  
BackgroundService  
{  
    // Use producer...  
}
```

App host usage

To model the Kafka resource in the app host, install the [Aspire.Hosting.Kafka](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Kafka
```

In your app host project, register a Kafka container and consume the connection using the following methods:

C#

```
var builder = DistributedApplication.CreateBuilder(args);  
  
var messaging = builder.AddKafka("messaging")  
    .WithKafkaUI();  
  
var myService = builder.AddProject<Projects.MyService>()  
    .WithReference(messaging);
```

The `WithKafkaUI()` extension method which provides a web-based interface to view the state of the Kafka container instance. The `WithReference` method configures a connection in the `MyService` project named `messaging`. In the `Program.cs` file of `MyService`, the Apache Kafka broker connection can be consumed using:

C#

```
builder.AddKafkaProducer<string, string>("messaging");
```

or

C#

```
builder.AddKafkaConsumer<string, string>("messaging");
```

Configuration

The .NET Aspire Apache Kafka integration provides multiple options to configure the connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddKafkaProducer()` or `builder.AddKafkaConsumer()`:

C#

```
builder.AddKafkaProducer<string, string>("myConnection");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "myConnection": "broker:9092"
  }
}
```

The value provided as connection string will be set to the `BootstrapServers` property of the produced `IProducer<TKey, TValue>` or `IConsumer<TKey, TValue>` instance. Refer to [BootstrapServers](#) for more information.

Use configuration providers

The .NET Aspire Apache Kafka integration supports [Microsoft.Extensions.Configuration](#). It loads the `KafkaProducerSettings` or `KafkaConsumerSettings` from configuration by respectively using the `Aspire:Confluent:Kafka:Producer` and `Aspire.Confluent.Kafka.Consumer` keys. This example `appsettings.json` configures some of the options:

JSON

```
{  
    "Aspire": {  
        "Confluent": {  
            "Kafka": {  
                "Producer": {  
                    "DisableHealthChecks": false,  
                    "Config": {  
                        "Acks": "All"  
                    }  
                }  
            }  
        }  
    }  
}
```

The `Config` properties of both `Aspire:Confluent:Kafka:Producer` and `Aspire.Confluent:Kafka:Consumer` configuration sections respectively bind to instances of [ProducerConfig ↗](#) and [ConsumerConfig ↗](#).

`Confluent.Kafka.Consumer< TKey, TValue >` requires the `ClientId` property to be set to let the broker track consumed message offsets.

Use inline delegates

Configuring `KafkaProducerSettings` and `KafkaConsumerSettings`

You can pass the `Action< KafkaProducerSettings > configureSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddKafkaProducer< string, string >("messaging", settings =>  
    settings.DisableHealthChecks = true);
```

You can configure inline a consumer from code:

C#

```
builder.AddKafkaConsumer< string, string >("messaging", settings =>  
    settings.DisableHealthChecks = true);
```

Configuring `ProducerBuilder<TKey, TValue>` and `ConsumerBuilder<TKey, TValue>`

To configure `Confluent.Kafka` builders, pass an `Action<ProducerBuilder<TKey, TValue>>` (or `Action<ConsumerBuilder<TKey, TValue>>`):

C#

```
builder.AddKafkaProducer<string, MyMessage>("messaging", producerBuilder =>
{
    producerBuilder.SetValueSerializer(new MyMessageSerializer());
```

Refer to [ProducerBuilder<TKey, TValue>](#) and [ConsumerBuilder<TKey, TValue>](#) API documentation for more information.

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Apache Kafka integration handles the following:

- Adds the `Aspire.Confluent.Kafka.Producer` health check when `KafkaProducerSettings.DisableHealthChecks` is `false`.
- Adds the `Aspire.Confluent.Kafka.Consumer` health check when `KafkaConsumerSettings.DisableHealthChecks` is `false`.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Apache Kafka integration uses the following log categories:

- `Aspire.Confluent.Kafka`

Tracing

The .NET Aspire Apache Kafka integration will emit the following tracing activities using OpenTelemetry:

- `Aspire.Confluent.Kafka`

Metrics

The .NET Aspire Apache Kafka integration will emit the following metrics using OpenTelemetry:

- `messaging.kafka.network.tx`
- `messaging.kafka.network.transmitted`
- `messaging.kafka.network.rx`
- `messaging.kafka.network.received`
- `messaging.publish.messages`
- `messaging.kafka.message.transmitted`
- `messaging.receive.messages`
- `messaging.kafka.message.received`

See also

- [Apache Kafka ↗](#)
- [Confluent ↗](#)
- [Confluent Kafka .NET client docs ↗](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Azure OpenAI integration

Article • 09/28/2024

In this article, you learn how to use the .NET Aspire Azure OpenAI client. The `Aspire.Azure.AI.OpenAI` library is used to register an `OpenAIClient` in the dependency injection (DI) container for consuming Azure OpenAI or OpenAI functionality. It enables corresponding logging and telemetry.

For more information on using the `OpenAIClient`, see [Quickstart: Get started generating text using Azure OpenAI Service](#).

Get started

- Azure subscription: [create one for free ↗](#).
- Azure OpenAI or OpenAI account: [create an Azure OpenAI Service resource](#).

To get started with the .NET Aspire Azure OpenAI integration, install the `Aspire.Azure.AI.OpenAI` NuGet package in the client-consuming project, i.e., the project for the application that uses the Azure OpenAI client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Azure.AI.OpenAI
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the extension method to register an `OpenAIClient` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddAzureOpenAIClient("openAiConnectionName");
```

In the preceding code, the `AddAzureOpenAI` method adds an `OpenAIClient` to the DI container. The `openAiConnectionName` parameter is the name of the connection string in the configuration. You can then retrieve the `OpenAIClient` instance using dependency injection. For example, to retrieve the connection from an example service:

```
C#  
  
public class ExampleService(OpenAIClient client)  
{  
    // Use client...  
}
```

App host usage

To add Azure hosting support to your `IDistributedApplicationBuilder`, install the [Aspire.Hosting.Azure.CognitiveServices](#) NuGet package in the app host project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.CognitiveServices
```

In your app host project, register an Azure OpenAI resource using the following methods, such as `AddAzureOpenAI`:

C#

```
var builder = DistributedApplication.CreateBuilder(args);  
  
var openai = builder.ExecutionContext.IsPublishMode  
    ? builder.AddAzureOpenAI("openAiConnectionName")  
    : builder.AddConnectionString("openAiConnectionName");  
  
builder.AddProject<Projects.ExampleProject>()  
    .WithReference(openai);
```

The `AddAzureAIOpenAI` method will read connection information from the app host's configuration (for example, from "user secrets") under the `ConnectionStrings:openAiConnectionName` config key. The `WithReference` method passes that connection information into a connection string named `openAiConnectionName` in

the `ExampleProject` project. In the `Program.cs` file of `ExampleProject`, the connection can be consumed using:

```
C#  
  
builder.AddAzureAIOpenAI("openAiConnectionName");
```

Configuration

The .NET Aspire Azure OpenAI integration provides multiple options to configure the connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddAzureAIOpenAI`:

```
C#  
  
builder.AddAzureAIOpenAI("openAiConnectionName");
```

The connection string is retrieved from the `ConnectionStrings` configuration section, and there are two supported formats, either the account endpoint used in conjunction with the default Azure credential or a connection string with the account key.

Account endpoint

The recommended approach is to use an `Endpoint`, which works with the `AzureOpenAISettings.Credential` property to establish a connection. If no credential is configured, the `DefaultAzureCredential` is used.

```
JSON  
  
{  
  "ConnectionStrings": {  
    "openAiConnectionName": "https://{{account_name}}.openai.azure.com/"  
  }  
}
```

For more information, see [Use Azure OpenAI without keys](#).

Connection string

Alternatively, a custom connection string can be used.

JSON

```
{  
  "ConnectionStrings": {  
    "openAiConnectionString":  
      "Endpoint=https://{{account_name}}.openai.azure.com/;Key={{account_key}};"  
  }  
}
```

In order to connect to the non-Azure OpenAI service, drop the `Endpoint` property and only set the `Key` property to set the [API key](#).

Use configuration providers

The .NET Aspire Azure OpenAI integration supports [Microsoft.Extensions.Configuration](#). It loads the `AzureOpenAISettings` from configuration by using the `Aspire:Azure:AI:OpenAI` key. Example `appsettings.json` that configures some of the options:

JSON

```
{  
  "Aspire": {  
    "Azure": {  
      "AI": {  
        "OpenAI": {  
          "DisableTracing": false,  
        }  
      }  
    }  
  }  
}
```

Use inline delegates

Also you can pass the `Action<AzureOpenAISettings> configureSettings` delegate to set up some or all the options inline, for example to disable tracing from code:

C#

```
builder.AddAzureAIOpenAI(  
  "openAiConnectionString",
```

```
    static settings => settings.DisableTracing = true);
```

You can also setup the `OpenAIOptions` using the optional `Action<IAzureClientBuilder<OpenAIOptions>>` `configureClientBuilder` parameter of the `AddAzureAIOpenAI` method. For example, to set the client ID for this client:

C#

```
builder.AddAzureAIOpenAI(  
    "openAiConnectionName",  
    configureClientBuilder: builder => builder.ConfigureOptions(  
        options => options.Diagnostics.ApplicationId = "CLIENT_ID"));
```

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Azure OpenAI integration uses the following log categories:

- `Azure`
- `Azure.Core`
- `Azure.Identity`

See also

- [Azure OpenAI docs](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Azure AI Search Documents integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Azure AI Search Documents client. The `Aspire.Azure.Search.Documents` library is used to register an `SearchIndexClient` in the dependency injection (DI) container for connecting to Azure Search. It enables corresponding health checks and logging.

For more information on using the `SearchIndexClient`, see [How to use Azure.Search.Documents in a C# .NET Application](#).

Get started

- Azure subscription: [create one for free](#).
- Azure Search service: [create an Azure AI Search service resource](#).

To get started with the .NET Aspire Azure AI Search Documents integration, install the `Aspire.Azure.Search.Documents` NuGet package in the client-consuming project, i.e., the project for the application that uses the Azure AI Search Documents client.

```
.NET CLI
.NET CLI
dotnet add package Aspire.Azure.Search.Documents
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the extension method to register an `SearchIndexClient` for use via the dependency injection container. The `AddAzureSearchClient` method takes a connection name parameter.

```
C#
```

```
builder.AddAzureSearchClient("searchConnectionName");
```

You can then retrieve the `SearchIndexClient` instance using dependency injection. For example, to retrieve the client from an example service:

C#

```
public class ExampleService(SearchIndexClient indexClient)
{
    // Use indexClient
}
```

You can also retrieve a `SearchClient` which can be used for querying, by calling the `SearchIndexClient.GetSearchClient` method as follows:

C#

```
public class ExampleService(SearchIndexClient indexClient)
{
    public async Task<long> GetDocumentCountAsync(
        string indexName,
        CancellationToken cancellationToken)
    {
        var searchClient = indexClient.GetSearchClient(indexName);

        var documentCountResponse = await
searchClient.GetDocumentCountAsync(
            cancellationToken);

        return documentCountResponse.Value;
    }
}
```

For more information, see the [Azure AI Search client library for .NET](#) for examples on using the `SearchIndexClient`.

App host usage

To add Azure AI hosting support to your `IDistributedApplicationBuilder`, install the [Aspire.Hosting.Azure.Search](#) NuGet package in the `app host` project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Search
```

In the `Program.cs` file of `AppHost`, add an Azure Search service and consume the connection using the following methods:

C#

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

var search = builder.ExecutionContext.IsPublishMode
    ? builder.AddAzureSearch("search")
    : builder.AddConnectionString("search");

var myService = builder.AddProject<Projects.MyService>()
    .WithReference(search);
```

The `AddAzureSearch` method will read connection information from the `AppHost`'s configuration (for example, from "user secrets") under the `ConnectionStrings:search` config key. The `WithReference` method passes that connection information into a connection string named `search` in the `MyService` project. In the `Program.cs` file of `MyService`, the connection can be consumed using:

C#

```
builder.AddAzureSearch("search");
```

Configuration

The .NET Aspire Azure Search library provides multiple options to configure the Azure Search Service based on the requirements and conventions of your project. Note that either an `Endpoint` or a `ConnectionString` is required to be supplied.

Use a connection string

A connection can be constructed from the **Keys and Endpoint** tab with the format `Endpoint={endpoint};Key={key};`. You can provide the name of the connection string when calling `builder.AddAzureSearch()`:

C#

```
builder.AddAzureSearch("searchConnectionName");
```

And then the connection string will be retrieved from the `.ConnectionStrings` configuration section. Two connection formats are supported:

Account endpoint

The recommended approach is to use an `Endpoint`, which works with the `AzureSearchSettings.Credential` property to establish a connection. If no credential is configured, the `DefaultAzureCredential` is used.

JSON

```
{
  "ConnectionStrings": {
    "searchConnectionName": "https://{{search_service}}.search.windows.net/"
  }
}
```

Connection string

Alternatively, a custom connection string can be used.

JSON

```
{
  "ConnectionStrings": {
    "searchConnectionName":
      "Endpoint=https://{{search_service}}.search.windows.net/;Key={{account_key}};"
  }
}
```

Use configuration providers

The .NET Aspire Azure AI Search library supports `Microsoft.Extensions.Configuration`. It loads the `AzureSearchSettings` and `SearchClientOptions` from configuration by using the `Aspire:Azure:Search:Documents` key. Example `appsettings.json` that configures some of the options:

JSON

```
{  
  "Aspire": {  
    "Azure": {  
      "Search": {  
        "Documents": {  
          "DisableTracing": false,  
        }  
      }  
    }  
  }  
}
```

Use inline delegates

You can also pass the `Action<AzureSearchSettings> configureSettings` delegate to set up some or all the options inline, for example to disable tracing from code:

C#

```
builder.AddAzureSearch(  
  "searchConnectionName",  
  static settings => settings.DisableTracing = true);
```

You can also setup the `SearchClientOptions` using the optional `Action<IAzureClientBuilder<SearchIndexClient, SearchClientOptions>> configureClientBuilder` parameter of the `AddAzureSearch` method. For example, to set the client ID for this client:

C#

```
builder.AddAzureSearch(  
  "searchConnectionName",  
  configureClientBuilder: builder => builder.ConfigureOptions(  
    static options => options.Diagnostics.ApplicationId = "CLIENT_ID"));
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Azure AI Search Documents integration implements a single health check, that calls the `GetServiceStatisticsAsync` method on the `SearchIndexClient` to verify that the service is available.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Azure AI Search Documents integration uses the following log categories:

- `Azure`
- `Azure.Core`
- `Azure.Identity`

See also

- [Azure AI OpenAI docs](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Azure Blob Storage integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Azure Blob Storage integration. The `Aspire.Azure.Storage.Blobs` library is used to register a `BlobServiceClient` in the DI container for connecting to Azure Blob Storage. It also enables corresponding health checks, logging and telemetry.

Get started

To get started with the .NET Aspire Azure Blob Storage integration, install the [Aspire.Azure.Storage.Blobs](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Azure Blob Storage client.

```
.NET CLI
.NET CLI
dotnet add package Aspire.Azure.Storage.Blobs
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the `AddAzureBlobClient` extension to register a `BlobServiceClient` for use via the dependency injection container.

```
C#
builder.AddAzureBlobClient("blobs");
```

You can then retrieve the `BlobServiceClient` instance using dependency injection. For example, to retrieve the client from a service:

```
C#
```

```
public class ExampleService(BlobServiceClient client)
{
    // Use client...
}
```

App host usage

To add Azure Storage hosting support to your [IDistributedApplicationBuilder](#), install the [Aspire.Hosting.Azure.Storage](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Storage
```

In your app host project, register the Azure Blob Storage integration and consume the service using the following methods, such as [AddAzureStorage](#):

C#

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

var blobs = builder.AddAzureStorage("storage")
    .AddBlobs("blobs");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
    .WithReference(blobs);
```

The [AddBlobs](#) method will read connection information from the AppHost's configuration (for example, from "user secrets") under the `ConnectionStrings:blobs` config key. The [WithReference](#) method passes that connection information into a connection string named blobs in the `ExampleProject` project. In the `Program.cs` file of `ExampleProject`, the connection can be consumed using:

C#

```
builder.AddAzureBlobClient("blobs");
```

Configuration

The .NET Aspire Azure Blob Storage integration provides multiple options to configure the `BlobServiceClient` based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddAzureBlobClient`:

C#

```
builder.AddAzureBlobClient("blobs");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section. Two connection formats are supported:

Service URI

The recommended approach is to use a `ServiceUri`, which works with the `AzureStorageBlobsSettings.Credential` property to establish a connection. If no credential is configured, the `Azure.Identity.DefaultAzureCredential` is used.

JSON

```
{
  "ConnectionStrings": {
    "blobsConnectionString": "https://[account_name].blob.core.windows.net/"
  }
}
```

Connection string

Alternatively, an [Azure Storage connection string](#) can be used.

JSON

```
{
  "ConnectionStrings": {
    "blobsConnectionString": "AccountName=myaccount;AccountKey=myaccountkey"
  }
}
```

Use configuration providers

The .NET Aspire Azure Blob Storage integration supports [Microsoft.Extensions.Configuration](#). It loads the `AzureStorageBlobsSettings` and `BlobClientOptions` from configuration by using the `Aspire:Azure:Storage:Blobs` key. Example `appsettings.json` that configures some of the options:

JSON

```
{  
  "Aspire": {  
    "Azure": {  
      "Storage": {  
        "Blobs": {  
          "DisableHealthChecks": true,  
          "DisableTracing": false,  
          "ClientOptions": {  
            "Diagnostics": {  
              "ApplicationId": "myapp"  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

Use inline delegates

You can also pass the `Action<AzureStorageBlobsSettings> configureSettings` delegate to set up some or all the options inline, for example to configure health checks:

C#

```
builder.AddAzureBlobClient(  
  "blobs",  
  static settings => settings.DisableHealthChecks = true);
```

You can also set up the `BlobClientOptions` using

`Action<IAzureClientBuilder<BlobServiceClient, BlobClientOptions>> configureClientBuilder` delegate, the second parameter of the `AddAzureBlobClient` method. For example, to set the first part of user-agent headers for all requests issued by this client:

C#

```
builder.AddAzureBlobClient(
    "blobs",
    static configureClientBuilder: clientBuilder =>
        clientBuilder.ConfigureOptions(
            static options => options.Diagnostics.ApplicationId = "myapp"));
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Azure Blob Storage integration handles the following:

- Adds the `AzureBlobStorageHealthCheck` health check, which attempts to connect to and query blob storage
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Azure Blob Storage integration uses the following log categories:

- `Azure.Core`
- `Azure.Identity`

Tracing

The .NET Aspire Azure Blob Storage integration will emit the following tracing activities using OpenTelemetry:

- "Azure.Storage.Blobs.BlobContainerClient"

Metrics

The .NET Aspire Azure Blob Storage integration currently does not support metrics by default due to limitations with the Azure SDK.

See also

- [Azure Blob Storage docs](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Microsoft Entity Framework Core Cosmos DB integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Microsoft Entity Framework Core Cosmos DB integration. The `Aspire.Microsoft.EntityFrameworkCore.Cosmos` library is used to register a `System.Data.Entity.DbContext` as a singleton in the DI container for connecting to Azure Cosmos DB. It also enables corresponding health checks, logging and telemetry.

Get started

To get started with the .NET Aspire Microsoft Entity Framework Core Cosmos DB integration, install the [Aspire.Microsoft.EntityFrameworkCore.Cosmos](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Microsoft Entity Framework Core Cosmos DB client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Microsoft.EntityFrameworkCore.Cosmos
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the `AddCosmosDbContext` extension to register a `System.Data.Entity.DbContext` for use via the dependency injection container.

C#

```
builder.AddCosmosDbContext<MyDbContext>("cosmosdb");
```

You can then retrieve the `DbContext` instance using dependency injection. For example, to retrieve the client from a service:

C#

```
public class ExampleService(MyDbContext context)
{
    // Use context...
}
```

For more information on using Entity Framework Core with Azure Cosmos DB, see the [Examples for Azure Cosmos DB for NoSQL SDK for .NET](#).

App host usage

To add Azure Cosmos DB hosting support to your `IDistributedApplicationBuilder`, install the [Aspire.Hosting.Azure.CosmosDB](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.CosmosDB
```

In your app host project, register the .NET Aspire Microsoft Entity Framework Core Cosmos DB integration and consume the service using the following methods:

C#

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

var cosmos = builder.AddAzureCosmosDB("cosmos");
var cosmosdb = cosmos.AddDatabase("cosmosdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
    .WithReference(cosmosdb);
```

💡 Tip

To use the Azure Cosmos DB emulator, chain a call to the [AddAzureCosmosDB](#) method.

C#

```
cosmosdb.RunAsEmulator();
```

Configuration

The .NET Aspire Microsoft Entity Framework Core Cosmos DB integration provides multiple options to configure the Azure Cosmos DB connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

```
builder.AddCosmosDbContext :
```

C#

```
builder.AddCosmosDbContext<MyDbContext>("CosmosConnection");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "CosmosConnection": {
      "AccountEndpoint": "https://{{account_name}}.documents.azure.com:443/",
      "AccountKey": "{{account_key}}"
    }
}
```

For more information, see the [ConnectionString documentation](#).

Use configuration providers

The .NET Aspire Microsoft Entity Framework Core Cosmos DB integration supports [Microsoft.Extensions.Configuration](#). It loads the `EntityFrameworkCoreCosmosSettings` from `appsettings.json` or other configuration files using

`Aspire:Microsoft:EntityFrameworkCore:Cosmos` key. Example `appsettings.json` that configures some of the options:

JSON

```
{  
    "Aspire": {  
        "Microsoft": {  
            "EntityFrameworkCore": {  
                "Cosmos": {  
                    "DisableTracing": true  
                }  
            }  
        }  
    }  
}
```

Use inline delegates

You can also pass the `Action<EntityFrameworkCoreCosmosSettings> configureSettings` delegate to set up some or all the `EntityFrameworkCoreCosmosSettings` options inline, for example to disable tracing from code:

C#

```
builder.AddCosmosDbContext<MyDbContext>(  
    "cosmosdb",  
    settings => settings.DisableTracing = true);
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Microsoft Entity Framework Core Cosmos DB integration currently doesn't implement health checks, though this may change in future releases.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Microsoft Entity Framework Core Cosmos DB integration uses the following log categories:

- Azure-Cosmos-Operation-Request-Diagnostics
- Microsoft.EntityFrameworkCore.ChangeTracking
- Microsoft.EntityFrameworkCore.Database.Command
- Microsoft.EntityFrameworkCore.Infrastructure
- Microsoft.EntityFrameworkCore.Query

Tracing

The .NET Aspire Microsoft Entity Framework Core Cosmos DB integration will emit the following tracing activities using OpenTelemetry:

- Azure.Cosmos.Operation
- OpenTelemetry.Instrumentation.EntityFrameworkCore

Metrics

The .NET Aspire Microsoft Entity Framework Core Cosmos DB integration currently supports the following metrics:

- Microsoft.EntityFrameworkCore"
 - ec_Microsoft_EntityFrameworkCore_active_db_contexts
 - ec_Microsoft_EntityFrameworkCore_total_queries
 - ec_Microsoft_EntityFrameworkCore_queries_per_second
 - ec_Microsoft_EntityFrameworkCore_total_save_changes
 - ec_Microsoft_EntityFrameworkCore_save_changes_per_second
 - ec_Microsoft_EntityFrameworkCore_compiled_query_cache_hit_rate
 - ec_Microsoft_Entity_total_execution_strategy_operation_failures
 - ec_Microsoft_E_execution_strategy_operation_failures_per_second
 - ec_Microsoft_EntityFramew_total_optimistic_concurrency_failures
 - ec_Microsoft_EntityF_optimistic_concurrency_failures_per_second

See also

- [Azure Cosmos DB docs](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Azure Cosmos DB integration

Article • 09/28/2024

In this article, you learn how to use the .NET Aspire Azure Cosmos DB integration. The `Aspire.Microsoft.Azure.Cosmos` library is used to register a `CosmosClient` as a singleton in the DI container for connecting to Azure Cosmos DB. It also enables corresponding health checks, logging and telemetry.

Get started

To get started with the .NET Aspire Azure Cosmos DB integration, install the [Aspire.Microsoft.Azure.Cosmos](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Azure Cosmos DB client.

```
.NET CLI
.NET CLI
dotnet add package Aspire.Microsoft.Azure.Cosmos
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the `AddAzureCosmosClient` extension to register a `Microsoft.Azure.Cosmos.CosmosClient` for use via the dependency injection container.

```
C#
builder.AddAzureCosmosClient("cosmosConnectionName");
```

You can then retrieve the `CosmosClient` instance using dependency injection. For example, to retrieve the client from a service:

```
C#
```

```
public class ExampleService(CosmosClient client)
{
    // Use client...
}
```

For more information on using the [CosmosClient](#), see the [Examples for Azure Cosmos DB for NoSQL SDK for .NET](#).

App host usage

To add Azure Cosmos DB hosting support to your [IDistributedApplicationBuilder](#), install the [Aspire.Hosting.Azure.CosmosDB](#) NuGet package in the [app host](#) project. This is helpful if you want Aspire to provision a new Azure Cosmos DB account for you, or if you want to use the Azure Cosmos DB emulator. If you want to use an Azure Cosmos DB account that is already provisioned, there's no need to add it to the app host project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.CosmosDB
```

In your app host project, register the .NET Aspire Azure Cosmos DB integration and consume the service using the following methods:

C#

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

var cosmos = builder.AddAzureCosmosDB("myNewCosmosAccountName");
var cosmosdb = cosmos.AddDatabase("myCosmosDatabaseName");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
    .WithReference(cosmosdb);
```

💡 Tip

To use the Azure Cosmos DB emulator, chain a call to the [AddAzureCosmosDB](#) method.

C#

```
cosmosdb.RunAsEmulator();
```

Configuration

The .NET Aspire Azure Cosmos DB library provides multiple options to configure the `CosmosClient` connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddAzureCosmosClient`:

C#

```
builder.AddAzureCosmosClient("cosmosConnectionString");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "cosmosConnectionString":
      "https://{{account_name}}.documents.azure.com:443/"
  }
}
```

The recommended connection approach is to use an account endpoint, which works with the `Microsoft.Azure.CosmosSettings.Credential` property to establish a connection. If no credential is configured, the `DefaultAzureCredential` is used:

JSON

```
{
  "ConnectionStrings": {
    "cosmosConnectionString":
      "https://{{account_name}}.documents.azure.com:443/"
  }
}
```

Alternatively, an Azure Cosmos DB connection string can be used:

JSON

```
{  
    "ConnectionStrings": {  
        "cosmosConnectionString":  
            "AccountEndpoint=https://{{account_name}}.documents.azure.com:443/;AccountKey={{account_key}};"  
    }  
}
```

Use configuration providers

The .NET Aspire Azure Cosmos DB integration supports

[Microsoft.Extensions.Configuration](#). It loads the [MicrosoftAzureCosmosSettings](#) from *appsettings.json* or other configuration files using `Aspire:Microsoft:Azure:Cosmos` key. Example *appsettings.json* that configures some of the options:

JSON

```
{  
    "Aspire": {  
        "Microsoft": {  
            "Azure": {  
                "Cosmos": {  
                    "DisableTracing": false,  
                }  
            }  
        }  
    }  
}
```

Use inline delegates

You can also pass the `Action<MicrosoftAzureCosmosSettings>` delegate to set up some or all the options inline, for example to disable tracing from code:

C#

```
builder.AddAzureCosmosClient(  
    "cosmosConnectionString",  
    static settings => settings.DisableTracing = true);
```

You can also set up the [Microsoft.Azure.Cosmos.CosmosClientOptions](#) using the optional `Action<CosmosClientOptions> configureClientOptions` parameter of the `AddAzureCosmosClient` method. For example to set the `CosmosClientOptions.ApplicationName` user-agent header suffix for all requests issued by this client:

C#

```
builder.AddAzureCosmosClient(  
    "cosmosConnectionName",  
    configureClientOptions:  
        clientOptions => clientOptions.ApplicationName = "myapp");
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Azure Cosmos DB integration currently doesn't implement health checks, though this may change in future releases.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Azure Cosmos DB integration uses the following log categories:

- Azure-Cosmos-Operation-Request-Diagnostics

In addition to getting Azure Cosmos DB request diagnostics for failed requests, you can configure latency thresholds to determine which successful Azure Cosmos DB request diagnostics will be logged. The default values are 100 ms for point operations and 500 ms for non point operations.

C#

```
builder.AddAzureCosmosClient(
    "cosmosConnectionName",
    configureClientOptions:
        clientOptions => {
            clientOptions.CosmosClientTelemetryOptions = new()
            {
                CosmosThresholdOptions = new()
                {
                    PointOperationLatencyThreshold =
TimeSpan.FromMilliseconds(50),
                    NonPointOperationLatencyThreshold =
TimeSpan.FromMilliseconds(300)
                }
            };
        });
});
```

Tracing

The .NET Aspire Azure Cosmos DB integration will emit the following tracing activities using OpenTelemetry:

- Azure.Cosmos.Operation

Azure Cosmos DB tracing is currently in preview, so you must set the experimental switch to ensure traces are emitted. [Learn more about tracing in Azure Cosmos DB.](#)

C#

```
ApplicationContext.SetSwitch("Azure.Experimental.EnableActivitySource", true);
```

Metrics

The .NET Aspire Azure Cosmos DB integration currently doesn't support metrics by default due to limitations with the Azure SDK.

See also

- [Azure Cosmos DB docs](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Azure Event Hubs integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Azure Event Hubs integration. The `Aspire.Azure.Messaging.EventHubs` library offers options for registering the following types:

- `EventHubProducerClient`
- `EventHubBufferedProducerClient`
- `EventHubConsumerClient`
- `EventProcessorClient`
- `PartitionReceiver`

These type are registered in the DI container for connecting to [Azure Event Hubs](#).

Prerequisites

- Azure subscription: [create one for free](#).
- Azure Event Hubs namespace: for more information, see [add an Event Hubs namespace](#). Alternatively, you can use a connection string, which isn't recommended in production environments.

Get started

To get started with the .NET Aspire Azure Event Hubs integration, install the `Aspire.Azure.Messaging.EventHubs` NuGet package in the client-consuming project, i.e., the project for the application that uses the Azure Event Hubs client.

```
.NET CLI  
.NET CLI  
dotnet add package Aspire.Azure.Messaging.EventHubs
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Supported clients with options classes

The following clients are supported by the library, along with their corresponding options and settings classes:

 [Expand table](#)

Azure Client type	Azure Options class	.NET Aspire Settings class
<code>EventHubProducerClient</code>	<code>EventHubProducerClientOptions</code>	<code>AzureMessagingEventHubsProducerSettings</code>
<code>EventHubBufferedProducerClient</code>	<code>EventHubBufferedProducerClientOptions</code>	<code>AzureMessagingEventHubsBufferedProducerSettings</code>
<code>EventHubConsumerClient</code>	<code>EventHubConsumerClientOptions</code>	<code>AzureMessagingEventHubsConsumerSettings</code>

Azure Client type	Azure Options class	.NET Aspire Settings class
EventProcessorClient	EventProcessorClientOptions	AzureMessagingEventHubsProcessorSettings
PartitionReceiver	PartitionReceiverOptions	AzureMessagingEventHubsPartitionReceiverSettings

The client types are from the Azure SDK for .NET, as are the corresponding options classes. The settings classes are provided by the .NET Aspire Azure Event Hubs integration library.

Example usage

The following example assumes that you have an Azure Event Hubs namespace and an Event Hub created and wish to configure an `EventHubProducerClient` to send events to the Event Hub. The `EventHubBufferedProducerClient`, `EventHubConsumerClient`, `EventProcessorClient`, and `PartitionReceiver` are configured in a similar manner.

In the `Program.cs` file of your client-consuming project, call the `AddAzureEventHubProducerClient` extension to register a `EventHubProducerClient` for use via the dependency injection container.

```
C#
builder.AddAzureEventHubProducerClient("eventHubsConnectionName");
```

You can then retrieve the `EventHubProducerClient` instance using dependency injection. For example, to retrieve the client from a service:

```
C#
public class ExampleService(EventHubProducerClient client)
{
    // Use client...
}
```

For more information, see the [Azure.Messaging.EventHubs documentation](#) for examples on using the `EventHubProducerClient`.

App host usage

To add Azure Event Hub hosting support to your `IDistributedApplicationBuilder`, install the `Aspire.Hosting.Azure.EventHubs` NuGet package in the `app host` project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.EventHubs
```

In your app host project, add an Event Hubs connection and an Event Hub resource and consume the connection using the following methods:

C#

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

var eventHubs = builder.AddAzureEventHubs("eventHubsConnectionString")
    .AddEventHub("MyHub");

var exampleService = builder.AddProject<Projects.ExampleService>()
    .WithReference(eventHubs);
```

The `AddAzureEventHubs` method will read connection information from the AppHost's configuration (for example, from "user secrets") under the `ConnectionStrings:eventHubsConnectionString` config key. The `WithReference` method passes that connection information into a connection string named `eventHubsConnectionString` in the `ExampleService` project.

As of .NET Aspire 8.1, the Azure EventHubs extension for .NET Aspire supports launching a local emulator for EventHubs. You can use the emulator by applying the `RunAsEmulator()` extension method as follows:

C#

```
var eventHubs = builder.AddAzureEventHubs("eventHubsConnectionString")
    .RunAsEmulator()
    .AddEventHub("MyHub");
```

The emulator for Azure EventHubs results in two container resources being launched inside .NET Aspire derived from the name of the Event Hubs resource name.

ⓘ Important

Even though we are creating an Event Hub using the `AddEventHub` at the same time as the namespace, as of .NET Aspire version `preview-5`, the connection string will not include the `EntityPath` property, so the `EventHubName` property must be set in the settings callback for the preferred client. Future versions of Aspire will include the `EntityPath` property in the connection string and will not require the `EventHubName` property to be set in this scenario.

In the `Program.cs` file of `ExampleService`, the connection can be consumed using by calling of the supported Event Hubs client extension methods:

C#

```
builder.AddAzureEventProcessorClient(
    "eventHubsConnectionString",
    static settings =>
{
    settings.EventHubName = "MyHub";
});
```

Configuration

The .NET Aspire Azure Event Hubs library provides multiple options to configure the Azure Event Hubs connection based on the requirements and conventions of your project. Either a `FullyQualifiedNamespace` or

a `ConnectionString` is a required to be supplied.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, provide the name of the connection string when calling `builder.AddAzureEventHubProducerClient()` and other supported Event Hubs clients. In this example, the connection string does not include the `EntityPath` property, so the `EventHubName` property must be set in the settings callback:

```
C#  
  
builder.AddAzureEventHubProducerClient(  
    "eventHubsConnectionString",  
    static settings =>  
    {  
        settings.EventHubName = "MyHub";  
    });
```

And then the connection information will be retrieved from the `ConnectionStrings` configuration section. Two connection formats are supported:

Fully Qualified Namespace (FQN)

The recommended approach is to use a fully qualified namespace, which works with the `AzureMessagingEventHubsSettings.Credential` property to establish a connection. If no credential is configured, the `DefaultAzureCredential` is used.

```
JSON  
  
{  
    "ConnectionStrings": {  
        "eventHubsConnectionString": "{your_namespace}.servicebus.windows.net"  
    }  
}
```

Connection string

Alternatively, use a connection string:

```
JSON  
  
{  
    "ConnectionStrings": {  
        "eventHubsConnectionString":  
            "Endpoint=sb://mynamespace.servicebus.windows.net/;SharedAccessKeyName=accesskeyname;SharedAccessKey=accesskey;EntityPath=MyHub"  
    }  
}
```

Use configuration providers

The .NET Aspire Azure Event Hubs library supports [Microsoft.Extensions.Configuration](#). It loads the `AzureMessagingEventHubsSettings` and the associated Options, e.g. `EventProcessorClientOptions`, from configuration by using the `Aspire:Azure:Messaging:EventHubs:` key prefix, followed by the name of the specific client in use. For example, consider the `appsettings.json` that configures some of the options for an `EventProcessorClient`:

JSON

```
{  
  "Aspire": {  
    "Azure": {  
      "Messaging": {  
        "EventHubs": {  
          "EventProcessorClient": {  
            "EventHubName": "MyHub",  
            "ClientOptions": {  
              "Identifier": "PROCESSOR_ID"  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

You can also setup the Options type using the optional `Action<IAzureClientBuilder<EventProcessorClient, EventProcessorClientOptions>> configureClientBuilder` parameter of the `AddAzureEventProcessorClient` method. For example, to set the processor's client ID for this client:

C#

```
builder.AddAzureEventProcessorClient(  
  "eventHubsConnectionName",  
  configureClientBuilder: clientBuilder => clientBuilder.ConfigureOptions(  
    options => options.Identifier = "PROCESSOR_ID"));
```

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Azure Event Hubs integration uses the following log categories:

- `Azure.Core`
- `Azure.Identity`

Tracing

The .NET Aspire Azure Event Hubs integration will emit the following tracing activities using OpenTelemetry:

- "Azure.Messaging.EventHubs.*"

Metrics

The .NET Aspire Azure Event Hubs integration currently doesn't support metrics by default due to limitations with the Azure SDK for .NET. If that changes in the future, this section will be updated to reflect those changes.

See also

- [Azure Event Hubs](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Azure Key Vault integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Azure Key Vault integration. The `Aspire.Azure.Key.Vault` integration library is used to register a `SecretClient` in the DI container for connecting to Azure Key Vault. It also enables corresponding health checks, logging and telemetry.

Get started

To get started with the .NET Aspire Azure Key Vault integration, install the [Aspire.Azure.Security.KeyVault](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Azure Key Vault client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Azure.Security.KeyVault
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

The following sections describe various example usages.

Add secrets to configuration

In the `Program.cs` file of your client-consuming project, call the `AddAzureKeyVaultSecrets` extension to add the secrets in the Azure Key Vault to the application's Configuration. The method takes a connection name parameter.

C#

```
builder.Configuration.AddAzureKeyVaultSecrets("secrets");
```

You can then retrieve a secret through normal `IConfiguration` APIs. For example, to retrieve a secret from a service:

C#

```
public class ExampleService(IConfiguration configuration)
{
    string secretValue = configuration["secretKey"];
    // Use secretValue ...
}
```

Use SecretClient

Alternatively, you can use a `SecretClient` to retrieve the secrets on demand. In the `Program.cs` file of your client-consuming project, call the `AddAzureKeyVaultClient` extension to register a `SecretClient` for use via the dependency injection container.

C#

```
builder.AddAzureKeyVaultClient("secrets");
```

You can then retrieve the `SecretClient` instance using dependency injection. For example, to retrieve the client from a service:

C#

```
public class ExampleService(SecretClient client)
{
    // Use client...
}
```

App host usage

To add Azure Key Vault hosting support to your `IDistributedApplicationBuilder`, install the [Aspire.Hosting.Azure.KeyVault](#) NuGet package in the `app host` project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.KeyVault
```

In your app host project, register the Azure Key Vault integration and consume the service using the following methods:

C#

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

var secrets = builder.ExecutionContext.IsPublishMode
    ? builder.AddAzureKeyVault("secrets")
    : builder.AddConnectionString("secrets");

builder.AddProject<Projects.ExampleProject>()
    .WithReference(secrets)
```

The preceding code conditionally adds the Azure Key Vault resource to the project based on the execution context. If the app host is executing in publish mode, the resource is added otherwise the connection string to an existing resource is added.

Configuration

The .NET Aspire Azure Key Vault integration provides multiple options to configure the `SecretClient` based on the requirements and conventions of your project.

Use configuration providers

The .NET Aspire Azure Key Vault integration supports `Microsoft.Extensions.Configuration`. It loads the `AzureSecurityKeyVaultSettings` from `appsettings.json` or other configuration files using `Aspire:Azure:Security:KeyVault` key.

JSON

```
{
  "Aspire": {
    "Azure": {
      "Security": {
        "KeyVault": {
          "VaultUri": "YOUR_VAULT_URI",
          "DisableHealthChecks": false,
          "DisableTracing": true,
          "ClientOptions": {
            "DisableChallengeResourceVerification": true
          }
        }
      }
    }
  }
}
```

If you have set up your configurations in the `Aspire:Azure:Security:KeyVault` section of your `appsettings.json` file you can just call the method `AddAzureKeyVaultSecrets` without passing any parameters.

Use inline delegates

You can also pass the `Action<AzureSecurityKeyVaultSettings>` delegate to set up some or all the options inline, for example to set the `VaultUri`:

```
C#  
  
builder.AddAzureKeyVaultSecrets(  
    "secrets",  
    static settings => settings.VaultUri = new Uri("YOUR_VAULTURI"));
```

💡 Tip

The `AddAzureKeyVaultSecrets` API name has caused a bit of confusion. The method is used to configure the `SecretClient` and not to add secrets to the configuration.

You can also set up the `SecretClientOptions` using

`Action<IAzureClientBuilder<SecretClient, SecretClientOptions>>` delegate, the second parameter of the `AddAzureKeyVaultSecrets` method. For example to set the `KeyClientOptions.DisableChallengeResourceVerification` ID to identify the client:

```
C#  
  
builder.AddAzureKeyVaultSecrets(  
    "secrets",  
    static clientBuilder =>  
        clientBuilder.ConfigureOptions(  
            static options => options.DisableChallengeResourceVerification =  
true))
```

Named instances

If you want to add more than one `SecretClient` you can use named instances. Load the named configuration section from the json config by calling the `AddAzureKeyVaultSecrets` method and passing in the `INSTANCE_NAME`.

```
C#
```

```
builder.AddAzureKeyVaultSecrets("INSTANCE_NAME");
```

The corresponding configuration JSON is defined as follows:

JSON

```
{  
    "Aspire": {  
        "Azure": {  
            "Security": {  
                "KeyVault": {  
                    "INSTANCE_NAME": {  
                        "VaultUri": "YOUR_VAULT_URI",  
                        "DisableHealthChecks": false,  
                        "DisableTracing": true,  
                        "ClientOptions": {  
                            "DisableChallengeResourceVerification": true  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

Configuration options

The following configurable options are exposed through the [AzureSecurityKeyVaultSettings](#) class:

[] [Expand table](#)

Name	Description
VaultUri	A URI to the vault on which the client operates. Appears as "DNS Name" in the Azure portal.
Credential	The credential used to authenticate to the Azure Key Vault.
DisableHealthChecks	A boolean value that indicates whether the Key Vault health check is disabled or not.
DisableTracing	A boolean value that indicates whether the OpenTelemetry tracing is disabled or not.

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Azure Key Vault integration includes the following health checks:

- Adds the `AzureKeyVaultSecretsHealthCheck` health check, which attempts to connect to and query the Key Vault
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Azure Key Vault integration uses the following log categories:

- `Azure.Core`
- `Azure.Identity`

Tracing

The .NET Aspire Azure Key Vault integration will emit the following tracing activities using OpenTelemetry:

- "Azure.Security.KeyVault.Secrets.SecretClient"

Metrics

The .NET Aspire Azure Key Vault integration currently does not support metrics by default due to limitations with the Azure SDK.

See also

- [Azure Key Vault docs](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire support for Azure SignalR Service

Article • 06/13/2024

In this article, you learn how to use .NET Aspire to express an Azure SignalR Service resource. Demonstrating how to write a SignalR app is beyond the scope of this article. Instead, you explore an app that's already been written and how it's wired up with .NET Aspire. Like other Azure resources within the .NET Aspire [app model](#), you benefit from simple provisioning and deployment with the Azure Developer CLI (`azd`). For more information, see [Deploy a .NET Aspire project to Azure Container Apps using the azd \(in-depth guide\)](#).

Hub host

The hub host project is where you host your SignalR hub, the project that calls `AddSignalR()` and `MapHub` for example.

Install the NuGet package

You need to install the [Microsoft.Azure.SignalR](#) NuGet package.

```
.NET CLI
dotnet add package Microsoft.Azure.SignalR
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Express the resource

Whichever project you're using to host your `Hub` is where you'll wire up your Azure SignalR Service resource. The following example demonstrates how to use the `AddNamedAzureSignalR` extension method which is chained on the `AddSignalR` method:

C#

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

builder.AddServiceDefaults();

builder.Services.AddProblemDetails();

builder.Services.AddSignalR()
    .AddNamedAzureSignalR("signalr");

var app = builder.Build();

app.UseExceptionHandler();

app.MapHub<ChatHub>(HubEndpoints.ChatHub);

app.MapDefaultEndpoints();

app.Run();
```

Calling `AddNamedAzureSignalR` adds Azure SignalR with the specified name, the connection string will be read from `ConnectionStrings_{name}`, the settings are loaded from `Azure:SignalR:{name}` section.

App host

In the [app host project](#), you express an `AzureSignalRResource` with the `AddAzureSignalR` method. The following example demonstrates how the resource is referenced by the consuming project, in this case the `Hub` host project:

C#

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

var signalr = builder.ExecutionContext.IsPublishMode
    ? builder.AddAzureSignalR("signalr")
    : builder.AddConnectionString("signalr");

var apiService = builder.AddProject<Projects.SignalR_ApiService>
    ("apiservice")
        .WithReference(signalr);

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

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

In the preceding code:

- The `builder` has its execution context checked to see if it's in publish mode.
- When publishing the `AddAzureSignalR` method is called to express the `AzureSignalRResource`.
- When not publishing, the `AddConnectionString` method is called to express an `IResourceWithConnectionString` to an existing resource.
- The `signalr` resource is referenced by the `Hub` host project, in this case known as `apiService`.
- The `apiService` project resource is referenced by the `SignalR_Web` project.

See also

- [Azure SignalR Service](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET Aspire Azure Service Bus integration

Article • 08/29/2024

Cloud-native apps often require communication with messaging services such as [Azure Service Bus](#). Messaging services help decouple applications and enable scenarios that rely on features such as queues, topics and subscriptions, atomic transactions, load balancing, and more. The .NET Aspire Service Bus integration handles the following concerns to connect your app to Azure Service Bus:

- A `ServiceBusClient` is registered in the DI container for connecting to Azure Service Bus.
- Applies `ServiceBusClient` configurations either inline through code or through configuration file settings.

Prerequisites

- Azure subscription - [create one for free ↗](#)
- Azure Service Bus namespace, learn more about how to [add a Service Bus namespace](#). Alternatively, you can use a connection string, which is not recommended in production environments.

Get started

To get started with the .NET Aspire Azure Service Bus integration, install the [Aspire.Azure.Messaging.ServiceBus ↗](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Azure Service Bus client.

.NET CLI

```
.NET CLI
dotnet add package Aspire.Azure.Messaging.ServiceBus
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the `AddAzureServiceBusClient` extension to register a `ServiceBusClient` for use via the dependency injection container.

```
C#
```

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

To retrieve the configured `ServiceBusClient` instance using dependency injection, require it as a constructor parameter. For example, to retrieve the client from an example service:

```
C#
```

```
public class ExampleService(ServiceBusClient client)
{
    // ...
}
```

App host usage

To add Azure Service Bus hosting support to your `IDistributedApplicationBuilder`, install the [Aspire.Hosting.Azure.ServiceBus](#) NuGet package in the `app host` project.

```
.NET CLI
```

```
.NET CLI
```

```
dotnet add package Aspire.Hosting.Azure.ServiceBus
```

In your app host project, register the Service Bus integration and consume the service using the following methods:

```
C#
```

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

var serviceBus = builder.ExecutionContext.IsPublishMode
    ? builder.AddAzureServiceBus("messaging")
    : builder.AddConnectionString("messaging");
```

```
builder.AddProject<Projects.ExampleProject>()
    .WithReference(serviceBus)
```

Configuration

The .NET Aspire Service Bus integration provides multiple options to configure the `ServiceBusClient` based on the requirements and conventions of your project.

Use configuration providers

The Service Bus integration supports [Microsoft.Extensions.Configuration](#). It loads the `AzureMessagingServiceBusSettings` from `appsettings.json` or other configuration files using `Aspire:Azure:Messaging:ServiceBus` key.

JSON

```
{
  "Aspire": {
    "Azure": {
      "Messaging": {
        "ServiceBus": {
          "DisableHealthChecks": true,
          "DisableTracing": false,
          "ClientOptions": {
            "Identifier": "CLIENT_ID"
          }
        }
      }
    }
  }
}
```

If you have set up your configurations in the `Aspire:Azure:Messaging:ServiceBus` section of your `appsettings.json` file you can just call the method `AddAzureServiceBus` without passing any parameters.

Use inline delegates

You can also pass the `Action<AzureMessagingServiceBusSettings>` delegate to set up some or all the options inline, for example to set the `FullyQualifiedNamespace`:

C#

```
builder.AddAzureServiceBus(  
    "messaging",  
    static settings => settings.FullyQualifiedNamespace =  
    "YOUR_SERVICE_BUS_NAMESPACE");
```

You can also set up the [ServiceBusClientOptions](#) using

`Action<IAzureClientBuilder<ServiceBusClient, ServiceBusClientOptions>>` delegate, the second parameter of the `AddAzureServiceBus` method. For example to set the `ServiceBusClient` ID to identify the client:

C#

```
builder.AddAzureServiceBus(  
    "messaging",  
    static clientBuilder =>  
        clientBuilder.ConfigureOptions(  
            static options => options.Identifier = "CLIENT_ID"));
```

Named instances

If you want to add more than one `ServiceBusClient` you can use named instances. Load the named configuration section from the JSON config by calling the `AddAzureServiceBus` method and passing in the `INSTANCE_NAME`.

C#

```
builder.AddAzureServiceBus("INSTANCE_NAME");
```

The corresponding configuration JSON is defined as follows:

JSON

```
{  
    "Aspire": {  
        "Azure": {  
            "Messaging": {  
                "INSTANCE_NAME": {  
                    "FullyQualifiedNamespace": "YOUR_SERVICE_BUS_NAMESPACE",  
                    "ClientOptions": {  
                        "Identifier": "CLIENT_ID"  
                    }  
                }  
            }  
        }  
    }  
}
```

```
    }  
}
```

Configuration options

The following configurable options are exposed through the [AzureMessagingServiceBusSettings](#) class:

[+] [Expand table](#)

Name	Description
<code>ConnectionString</code>	The connection string used to connect to the Service Bus namespace.
<code>Credential</code>	The credential used to authenticate to the Service Bus namespace.
<code>FullyQualifiedNamespace</code>	The fully qualified Service Bus namespace.
<code>DisableTracing</code>	Disables tracing for the Service Bus client.
[†] <code>HealthCheckQueueName</code>	The name of the queue used for health checks.
[†] <code>HealthCheckTopicName</code>	The name of the topic used for health checks.

[†] At least one of the name options are mandatory when enabling health checks.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Azure Service Bus integration uses the following log categories:

- `Azure.Core`
- `Azure.Identity`
- `Azure-Messaging-ServiceBus`

Tracing

① Note

Service Bus `ActivitySource` support in the Azure SDK for .NET is experimental, and the shape of activities may change in the future without notice.

You can enable tracing in several ways:

- Setting the `Azure.Experimental.EnableActivitySource` runtime configuration setting to `true`. Which can be done with either:
 - Call `AppContext.SetSwitch("Azure.Experimental.EnableActivitySource", true);`
 - Add the `RuntimeHostConfigurationOption` setting to your project file:

XML

```
<ItemGroup>
    <RuntimeHostConfigurationOption
        Include="Azure.Experimental.EnableActivitySource"
        Value="true" />
</ItemGroup>
```

- Set the `AZURE_EXPERIMENTAL_ENABLE_ACTIVITY_SOURCE` environment variable to "true".
 - Can be achieved by chaining a call to
`WithEnvironment("AZURE_EXPERIMENTAL_ENABLE_ACTIVITY_SOURCE", "true")`

When enabled, the .NET Aspire Azure Service Bus integration will emit the following tracing activities using OpenTelemetry:

- `Message`
- `ServiceBusSender.Send`
- `ServiceBusSender.Schedule`
- `ServiceBusSender.Cancel`
- `ServiceBusReceiver.Receive`
- `ServiceBusReceiver.ReceiveDeferred`
- `ServiceBusReceiver.Peek`
- `ServiceBusReceiver.Abandon`
- `ServiceBusReceiver.Complete`
- `ServiceBusReceiver.DeadLetter`

- `ServiceBusReceiver.Defer`
- `ServiceBusReceiver.RenewMessageLock`
- `ServiceBusSessionReceiver.RenewSessionLock`
- `ServiceBusSessionReceiver.GetSessionState`
- `ServiceBusSessionReceiver.SetSessionState`
- `ServiceBusProcessor.ProcessMessage`
- `ServiceBusSessionProcessor.ProcessSessionMessage`
- `ServiceBusRuleManager.CreateRule`
- `ServiceBusRuleManager.DeleteRule`
- `ServiceBusRuleManager.GetRules`

For more information, see:

- [Azure SDK for .NET: Distributed tracing and the Service Bus client ↗](#).
- [Azure SDK for .NET: OpenTelemetry configuration ↗](#).
- [Azure SDK for .NET: Enabling experimental tracing features ↗](#).

Metrics

The .NET Aspire Azure Service Bus integration currently doesn't support metrics by default due to limitations with the Azure SDK for .NET. If that changes in the future, this section will be updated to reflect those changes.

See also

- [Azure Service Bus](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Azure Queue Storage integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Azure Queue Storage integration. The `Aspire.Azure.Storage.Queues` library is used to register a `QueueServiceClient` in the DI container for connecting to Azure Queue Storage. It also enables corresponding health checks, logging and telemetry.

Get started

To get started with the .NET Aspire Azure Queue Storage integration, install the [Aspire.Azure.Storage.Queues](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Azure Queue Storage client.

```
.NET CLI
.NET CLI
dotnet add package Aspire.Azure.Storage.Queues
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the `AddAzureQueueClient` extension to register a `QueueServiceClient` for use via the dependency injection container.

```
C#
builder.AddAzureQueueClient("queue");
```

You can then retrieve the `QueueServiceClient` instance using dependency injection. For example, to retrieve the client from an example service:

```
C#
```

```
public class ExampleService(QueueServiceClient client)
{
    // Use client...
}
```

App host usage

To add Azure Storage hosting support to your [IDistributedApplicationBuilder](#), install the [Aspire.Hosting.Azure.Storage](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Storage
```

In your app host project, add a Storage Queue connection and consume the connection using the following methods, such as [AddAzureStorage](#):

C#

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

var queues = builder.AddAzureStorage("storage")
    .AddQueues("queues");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
    .WithReference(queues);
```

The [AddQueues](#) method will read connection information from the AppHost's configuration (for example, from "user secrets") under the `ConnectionStrings:queue` config key. The [WithReference](#) method passes that connection information into a connection string named `queue` in the `ExampleProject` project. In the `Program.cs` file of `ExampleProject`, the connection can be consumed using:

C#

```
builder.AddAzureQueueClient("queue");
```

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

```
builder.AddAzureQueueClient:
```

C#

```
builder.AddAzureQueueClient("queueConnectionString");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section. Two connection formats are supported:

Service URI

The recommended approach is to use a `ServiceUri`, which works with the `AzureStorageQueuesSettings.Credential` property to establish a connection. If no credential is configured, the `Azure.Identity.DefaultAzureCredential` is used.

JSON

```
{
  "ConnectionStrings": {
    "queueConnectionString": "https://{{account_name}}.queue.core.windows.net/"
  }
}
```

Connection string

Alternatively, an [Azure Storage connection string](#) can be used.

JSON

```
{
  "ConnectionStrings": {
    "queueConnectionString": "AccountName=myaccount;AccountKey=myaccountkey"
  }
}
```

Configuration

The .NET Aspire Azure Queue Storage integration provides multiple options to configure the `QueueServiceClient` based on the requirements and conventions of your project.

Use configuration providers

The .NET Aspire Azure Queue Storage integration supports [Microsoft.Extensions.Configuration](#). It loads the [AzureStorageQueuesSettings](#) and [QueueClientOptions](#) from configuration by using the `Aspire:Azure:Storage:Queues` key. Example `appsettings.json` that configures some of the options:

JSON

```
{  
  "Aspire": {  
    "Azure": {  
      "Storage": {  
        "Queues": {  
          "DisableHealthChecks": true,  
          "DisableTracing": false,  
          "ClientOptions": {  
            "Diagnostics": {  
              "ApplicationId": "myapp"  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

Use inline delegates

You can also pass the `Action<AzureStorageQueuesSettings> configureSettings` delegate to set up some or all the options inline, for example to disable the health check:

C#

```
builder.AddAzureQueueClient(  
  "queue",  
  static settings => settings.DisableHealthChecks = true);
```

You can also set up the `QueueClientOptions` using

`Action<IAzureClientBuilder<QueueServiceClient, QueueClientOptions>> configureClientBuilder` delegate, the second parameter of the `AddAzureQueueClient` method. For example, to set the first part of user-agent headers for all requests issued by this client:

C#

```
builder.AddAzureQueueClient(
    "queue",
    configureClientBuilder:
        static clientBuilder => clientBuilder.ConfigureOptions(
            static options =>
                options.Diagnostics.ApplicationId = "myapp"));
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Azure Queue Storage integration handles the following:

- Adds the `AzureQueueStorageHealthCheck` health check, which attempts to connect to and query the storage queue
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Azure Queue Storage integration uses the following log categories:

- `Azure.Core`
- `Azure.Identity`

Tracing

The .NET Aspire Azure Queue Storage integration will emit the following tracing activities using OpenTelemetry:

- "Azure.Storage.Queues.QueueClient"

Metrics

The .NET Aspire Azure Queue Storage integration currently does not support metrics by default due to limitations with the Azure SDK.

See also

- [Azure Queues Storage docs](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Azure Data Tables integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Azure Data Tables integration. The `Aspire.Azure.Data.Tables` library is used to:

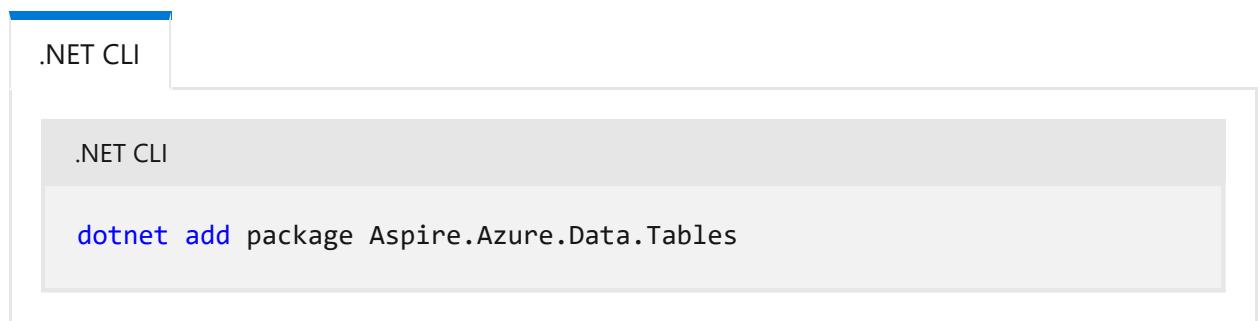
- Registers a `TableServiceClient` as a singleton in the DI container for connecting to Azure Table storage.
- Enables corresponding health checks, logging and telemetry.

Prerequisites

- Azure subscription - [create one for free ↗](#)
- An Azure storage account or Azure Cosmos DB database with Azure Table API specified. - [create a storage account](#)

Get started

To get started with the .NET Aspire Azure Data Tables integration, install the `Aspire.Azure.Data.Tables` NuGet package in the client-consuming project, i.e., the project for the application that uses the Azure Data Tables client.



For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your integration-consuming project, call the `AddAzureTableClient` extension to register a `TableServiceClient` for use via the dependency injection container.

C#

```
builder.AddAzureTableClient("tables");
```

To retrieve the `TableServiceClient` instance using dependency injection, define it as a constructor parameter. Consider the following example service:

C#

```
public class ExampleService(TableServiceClient client)
{
    // Use client...
}
```

App host usage

To add Azure Storage hosting support to your `IDistributedApplicationBuilder`, install the [Aspire.Hosting.Azure.Storage](#) NuGet package in the `app host` project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Storage
```

In your app host project, register the Azure Table Storage integration and consume the service using the following methods:

C#

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

var tables = builder.AddAzureStorage("storage")
    .AddTables("tables");

Builder.AddProject<MyApp.ExampleProject>()
    .WithReference(tables)
```

For more information, see [WithReference](#).

Configuration

The .NET Aspire Azure Table Storage integration provides multiple options to configure the `TableServiceClient` based on the requirements and conventions of your project.

Use configuration providers

The .NET Aspire Azure Table Storage integration supports `Microsoft.Extensions.Configuration`. It loads the `AzureDataTablesSettings` from `appsettings.json` or other configuration files using `Aspire:Azure:Data:Tables` key.

JSON

```
{  
  "Aspire": {  
    "Azure": {  
      "Data": {  
        "Tables": {  
          "ServiceUri": "YOUR_URI",  
          "DisableHealthChecks": true,  
          "DisableTracing": false,  
          "ClientOptions": {  
            "EnableTenantDiscovery": true  
          }  
        }  
      }  
    }  
  }  
}
```

If you have set up your configurations in the `Aspire:Azure:Data:Tables` section of your `appsettings.json` file you can just call the method `AddAzureTableClient` without passing any parameters.

Use inline delegates

You can also pass the `Action<AzureDataTablesSettings>` delegate to set up some or all the options inline, for example to set the `ServiceUri`:

C#

```
builder.AddAzureTableClient(  
  "tables",  
  static settings => settings.ServiceUri = new Uri("YOUR_SERVICEURI"));
```

You can also set up the `TableClientOptions` using

`Action<IAzureClientBuilder<TableServiceClient, TableClientOptions>>` delegate, the

second parameter of the `AddAzureTableClient` method. For example to set the `TableServiceClient` ID to identify the client:

```
C#  
  
builder.AddAzureTableClient(  
    "tables",  
    static clientBuilder =>  
        clientBuilder.ConfigureOptions(  
            static options => options.EnableTenantDiscovery = true));
```

Named instances

If you want to add more than one `TableServiceClient` you can use named instances. Load the named configuration section from the json config by calling the `AddAzureTableClient` method and passing in the `INSTANCE_NAME`.

```
C#  
  
builder.AddAzureTableClient("INSTANCE_NAME");
```

The corresponding configuration JSON is defined as follows:

```
JSON  
  
{  
    "Aspire": {  
        "Azure": {  
            "Data": {  
                "Tables": {  
                    "INSTANCE_NAME": {  
                        "ServiceUri": "YOUR_URI",  
                        "DisableHealthChecks": true,  
                        "ClientOptions": {  
                            "EnableTenantDiscovery": true  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

Configuration options

The following configurable options are exposed through the [AzureDataTablesSettings](#) class:

 [Expand table](#)

Name	Description
ServiceUri	A "Uri" referencing the Table service.
Credential	The credential used to authenticate to the Table Storage.
DisableHealthChecks	A boolean value that indicates whether the Table Storage health check is disabled or not.
DisableTracing	A boolean value that indicates whether the OpenTelemetry tracing is disabled or not.

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

By default, The .NET Aspire Azure Data Tables integration handles the following:

- Adds the `AzureTableStorageHealthCheck` health check, which attempts to connect to and query table storage
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Azure Data Tables integration uses the following log categories:

- `Azure.Core`
- `Azure.Identity`

Tracing

The .NET Aspire Azure Data Tables integration will emit the following tracing activities using OpenTelemetry:

- "Azure.Data.Tables.TableServiceClient"

Metrics

The .NET Aspire Azure Data Tables integration currently does not support metrics by default due to limitations with the Azure SDK.

See also

- [Azure Table Storage docs](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Azure Web PubSub integration

Article • 08/29/2024

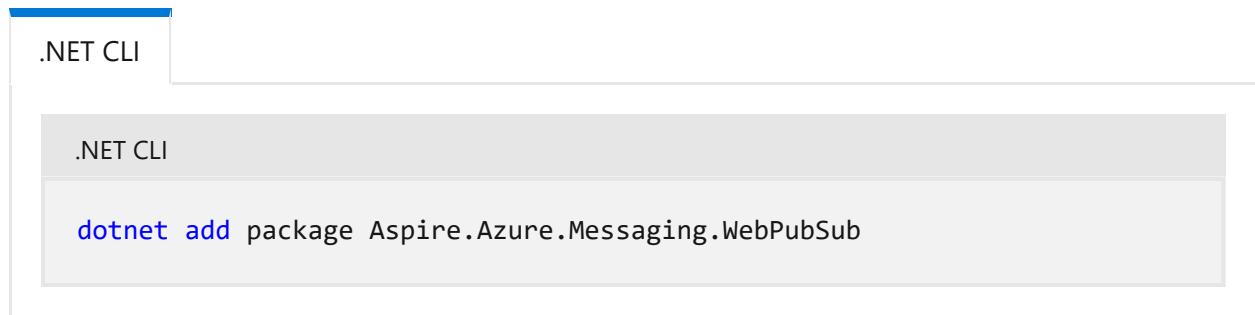
In this article, you learn how to use the .NET Aspire Azure Web PubSub integration. The `Aspire.Azure.Messaging.WebPubSub` library offers options for registering an `WebPubSubServiceClient` in the DI container for connecting to [Azure Web PubSub](#).

Prerequisites

- Azure subscription: [create one for free](#).
- An existing Azure Web PubSub service instance. For more information, see [Create a Web PubSub resource](#). Alternatively, you can use a connection string, which isn't recommended in production environments.

Get started

To get started with the .NET Aspire Azure Web PubSub integration, install the `Aspire.Azure.Messaging.WebPubSub` NuGet package in the client-consuming project, i.e., the project for the application that uses the Azure Web PubSub client.



For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your project, call the `AddAzureWebPubSubHub` extension method to register a `WebPubSubServiceClient` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddAzureWebPubSubServiceClient("wps");
```

You can then retrieve the `WebPubSubServiceClient` instance using dependency injection. For example, to retrieve the client from a service:

C#

```
public class ExampleService(WebPubSubServiceClient client)
{
    // Use client...
}
```

For more information, see the [Azure.Messaging.WebPubSub documentation](#).

App host usage

To add Azure Web PubSub hosting support to your `IDistributedApplicationBuilder`, install the [Aspire.Hosting.Azure.WebPubSub](#) NuGet package in the `app host` project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.WebPubSub
```

In your app host project, add a Web PubSub connection and consume the connection using the following methods:

C#

```
var webPubSub = builder.AddAzureWebPubSub("wps");

var exampleService = builder.AddProject<Projects.ExampleService>()
    .WithReference(webPubSub);
```

The `AddAzureWebPubSubHub` method reads connection information from the app host's configuration (for example, from "user secrets") under the `ConnectionStrings:wps` configuration key. The `WithReference` method passes that connection information into a connection string named `wps` in the `ExampleService` project. In the `Program.cs` file of `ExampleService`, the connection can be consumed using:

C#

```
builder.AddAzureWebPubSubServiceClient("wps");
```

Configuration

The .NET Aspire Azure Web PubSub library provides multiple options to configure the Azure Web PubSub connection based on the requirements and conventions of your project. Note that either a `Endpoint` or a `ConnectionString` is required to be supplied.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

```
builder.AddAzureWebPubSubHub();
```

C#

```
builder.AddAzureWebPubSubServiceClient(
    "WebPubSubConnectionString",
    "your_hub_name");
```

And then the connection information will be retrieved from the `ConnectionStrings` configuration section. Two connection formats are supported:

Use the service endpoint

The recommended approach is to use the service endpoint, which works with the `AzureMessagingWebPubSubSettings.Credential` property to establish a connection. If no credential is configured, the `DefaultAzureCredential` is used.

JSON

```
{
  "ConnectionStrings": {
    "WebPubSubConnectionString": "https://xxx.webpubsub.azure.com"
  }
}
```

Connection string

Alternatively, a connection string can be used.

JSON

```
{  
  "ConnectionStrings": {  
    "WebPubSubConnectionString":  
      "Endpoint=https://xxx.webpubsub.azure.com;AccessKey==xxxxxxxx"  
  }  
}
```

Use configuration providers

The .NET Aspire Azure Web PubSub library supports [Microsoft.Extensions.Configuration](#). It loads the `AzureMessagingWebPubSubSettings` and `WebPubSubServiceClientOptions` from configuration by using the `Aspire:Azure:Messaging:WebPubSub` key. Consider the example `appsettings.json` that configures some of the options:

JSON

```
{  
  "Aspire": {  
    "Azure": {  
      "Messaging": {  
        "WebPubSub": {  
          "DisableHealthChecks": true,  
          "HubName": "your_hub_name"  
        }  
      }  
    }  
  }  
}
```

Use inline delegates

You can also pass the `Action<AzureMessagingWebPubSubSettings> configureSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddAzureWebPubSubServiceClient(  
  "wps",  
  settings => settings.DisableHealthChecks = true);
```

You can also setup the [WebPubSubServiceClientOptions](#) using the optional `Action<IAzureClientBuilder<WebPubSubServiceClient, WebPubSubServiceClientOptions>>` `configureClientBuilder` parameter of the `AddAzureWebPubSubHub` method. For example, to set the client ID for this client:

```
C#  
  
builder.AddAzureWebPubSubServiceClient(  
    "wps",  
    configureClientBuilder: clientBuilder =>  
        clientBuilder.ConfigureOptions(options => options.Retry.MaxRetries =  
    5));
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Azure Web PubSub integration handles exposes a configurable health check that reports as *healthy*, when the client can successfully connect to the Azure Web PubSub service.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Azure Web PubSub integration uses the following log categories:

- `Azure`
- `Azure.Core`
- `Azure.Identity`
- `Azure.Messaging.WebPubSub`

Tracing

The .NET Aspire Azure Web PubSub integration will emit the following tracing activities using OpenTelemetry:

- "Azure.Messaging.WebPubSub.*"

Metrics

The .NET Aspire Azure Web PubSub integration currently doesn't support metrics by default due to limitations with the Azure SDK for .NET. If that changes in the future, this section will be updated to reflect those changes.

See also

- [Azure Web PubSub](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

Use Dapr with .NET Aspire

Article • 08/29/2024

Distributed Application Runtime (Dapr) [↗](#) offers developer APIs that serve as a conduit for interacting with other services and dependencies and abstract the application from the specifics of those services and dependencies. Dapr and .NET Aspire work together to improve your local development experience. By using Dapr with .NET Aspire, you can focus on writing and implementing .NET-based distributed applications instead of spending extra time with local onboarding.

In this guide, you'll learn how to take advantage of Dapr's abstraction and .NET Aspire's opinionated configuration of cloud technologies to build simple, portable, resilient, and secured microservices at-scale on Azure.

Prerequisites

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

- [.NET 8.0](#) [↗](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or [the .NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) [↗](#) or [Podman](#) [↗](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) [↗](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) [↗](#) (Optional)
 - [C# Dev Kit: Extension](#) [↗](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

In addition to the prerequisites for .NET Aspire, you will need:

- Dapr version 1.13 or later

To install Dapr, see [Install the Dapr CLI](#) [↗](#). After installing the Dapr CLI, run the `dapr init` described in [Initialize Dapr in your local environment](#) [↗](#).

Important

If you attempt to run the app without the Dapr CLI, you'll receive the following error:

plaintext

Unable to locate the Dapr CLI.

Get started

To get started you need to add the Dapr hosting package to your app host project by installing the [Aspire.Hosting.Dapr](#) NuGet package.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Dapr
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Add a Dapr sidecar

Dapr uses the [sidecar pattern](#) to run alongside your application. The Dapr sidecar runs alongside your app as a lightweight, portable, and stateless HTTP server that listens for incoming HTTP requests from your app.

To add a sidecar to a .NET Aspire resource, call the [WithDaprSidecar](#) method on the desired resource. The `appId` parameter is the unique identifier for the Dapr application, but it's optional. If you don't provide an `appId`, the parent resource name is used instead.

C#

```
using Aspire.Hosting.Dapr;

var builder = DistributedApplication.CreateBuilder(args);

var apiService = builder
    .AddProject<Projects.Dapr_ApiService>("apiservice")
    .WithDaprSidecar();
```

The `WithDaprSidecar` method offers overloads to configure your Dapr sidecar options like app ID and ports. In the following example, the Dapr sidecar is configured with

specific ports for GRPC, HTTP, metrics, and a specific app ID.

C#

```
DaprSidecarOptions sidecarOptions = new()
{
    DaprGrpcPort = 50001,
    DaprHttpPort = 3500,
    MetricsPort = 9090
};

builder.AddProject<Projects.Dapr_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(apiService)
    .WithDaprSidecar(sidecarOptions);
```

Putting everything together, consider the following example of a .NET Aspire app host project that includes:

- A backend API that declares a Dapr sidecar with defaults.
- A frontend that declares a Dapr sidecar with specific options, such as explicit ports.

C#

```
using Aspire.Hosting.Dapr;

var builder = DistributedApplication.CreateBuilder(args);

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

DaprSidecarOptions sidecarOptions = new()
{
    DaprGrpcPort = 50001,
    DaprHttpPort = 3500,
    MetricsPort = 9090
};

builder.AddProject<Projects.Dapr_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(apiService)
    .WithDaprSidecar(sidecarOptions);

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

The .NET Aspire dashboard shows the Dapr sidecar as a resource, with its status and logs.

Type	Name	State	Start time	Source	Endpoints
Executable	apiservice-dapr-cli	Running	1:52:11 PM	dapr.exe run --app-port 5000 --app-id apiservic...	http://localhost:3500 , http://localhost:50001
Executable	webfrontend-dapr-cli	Running	1:52:11 PM	dapr.exe run --app-id webfrontend-dapr --app-...	http://localhost:53568 , http://localhost:53569
Project	apiservice	Running	1:52:11 PM	Dapr.ApiService.csproj	https://localhost:7487/weatherforecast
Project	webfrontend	Running	1:52:11 PM	Dapr.Web.csproj	https://localhost:7068 , http://localhost:5187

Adding the Dapr SDK

To use Dapr APIs from .NET Aspire resources, you can use the [Dapr SDK for ASP.NET Core \(Dapr.AspNetCore\) library](#). The Dapr SDK provides a set of APIs to interact with Dapr sidecars.

! Note

Use the `Dapr.AspNetCore` library for the Dapr integration with ASP.NET (DI integration, registration of subscriptions, etc.). Non-ASP.NET apps (such as console apps) can just use the [Dapr.Client library](#) to make calls through the Dapr sidecar.

.NET CLI

.NET CLI

```
dotnet add package Dapr.AspNetCore
```

Once installed into an ASP.NET Core project, the SDK can be added to the service builder.

C#

```
builder.Services.AddDaprClient();
```

An instance of `DaprClient` can now be injected into your services to interact with the Dapr sidecar through the Dapr SDK.

C#

```
using Dapr.Client;

namespace Dapr.Web;
```

```

public class WeatherApiClient(DaprClient client)
{
    public async Task<WeatherForecast[]> GetWeatherAsync(
        int maxItems = 10, CancellationToken cancellationToken = default)
    {
        List<WeatherForecast>? forecasts =
            await client.InvokeMethodAsync<List<WeatherForecast>>(
                HttpMethod.Get,
                "apiservice",
                "weatherforecast",
                cancellationToken);

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

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

```

`InvokeMethodAsync` is a method that sends an HTTP request to the Dapr sidecar. It is a generic method that takes:

- An HTTP verb
- The Dapr app ID of the service to call
- The method name
- A cancellation token

Depending on the HTTP verb, it can also take a request body and headers. The generic type parameter is the type of the response body.

The full *Program.cs* file for the frontend project shows:

- The Dapr client being added to the service builder
- The `WeatherApiClient` class that uses the Dapr client to call the backend service

C#

```

using Dapr.Web;
using Dapr.Web.Components;

var builder = WebApplication.CreateBuilder(args);

// Add service defaults & Aspire components.
builder.AddServiceDefaults();

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

```

```
.AddInteractiveServerComponents();

builder.Services.AddOutputCache();

builder.Services.AddDaprClient();

builder.Services.AddTransient<WeatherApiClient>();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseStaticFiles();
app.UseAntiforgery();

app.UseOutputCache();

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

app.MapDefaultEndpoints();

app.Run();
```

For example, in a Blazor project, the `WeatherApiClient` class can be injected into a component and used to call the backend service.

C#

```
@page "/weather"
@attribute [StreamRendering(true)]
@attribute [OutputCache(Duration = 5)]

@inject WeatherApiClient WeatherApi

<PageTitle>Weather</PageTitle>

<h1>Weather</h1>

<p>This component demonstrates showing data loaded from a backend API service.</p>

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
```

```

{
    <table class="table">
        <thead>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
        </tbody>
    </table>
}

@code {
    private WeatherForecast[]? forecasts;

    protected override async Task OnInitializedAsync()
    {
        forecasts = await WeatherApi.GetWeatherAsync();
    }
}

```

When the Dapr SDK is used, the Dapr sidecar is called over HTTP. The Dapr sidecar then forwards the request to the target service. While the target service runs in a separate process from the sidecar, the integration related to the service runs in the Dapr sidecar and is responsible for service discovery and routing the request to the target service.

Dapr and .NET Aspire

At first sight Dapr and .NET Aspire may look like they have overlapping functionality, and they do. However, they both take a different approach. .NET Aspire is an opinionated approach on how to build distributed applications on a cloud platform. Dapr is a runtime that abstracts away the common complexities of the underlying cloud platform. It relies on sidecars to provide abstractions for things like configuration, secret management, and messaging. The underlying technology can be easily switched out through configuration files, while your code does not need to change.

.NET Aspire makes setting up and debugging Dapr applications easier by providing a straightforward API to configure Dapr sidecars, and by exposing the sidecars as resources in the dashboard.

Explore Dapr components with .NET Aspire

Dapr provides many [built-in components](#), and when you use Dapr with .NET Aspire you can easily explore and configure these components. Don't confuse these components with .NET Aspire integrations. For example, consider the following:

- **Dapr—State stores**: Call [AddDaprStateStore](#) to add a configured state store to your .NET Aspire project.
- **Dapr—Pub Sub**: Call [AddDaprPubSub](#) to add a configured pub sub to your .NET Aspire project.
- **Dapr—Components**: Call [AddDaprComponent](#) to add a configured integration to your .NET Aspire project.

Next steps

[.NET Aspire Dapr sample app](#)

.NET Aspire Elasticsearch integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Elasticsearch integration. The `Aspire.Elastic.Clients.Elasticsearch` library registers a [ElasticsearchClient](#) in the DI container for connecting to a Elasticsearch. It enables corresponding health check and telemetry.

Prerequisites

- Elasticsearch cluster.
- Endpoint URI string for accessing the Elasticsearch API endpoint or a CloudId and an ApiKey from [Elastic Cloud](#)

Get started

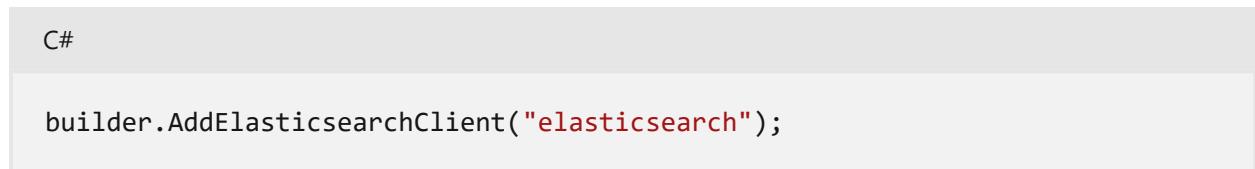
To get started with the .NET Aspire Elasticsearch integration, install the [Aspire.Elastic.Clients.Elasticsearch](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Elasticsearch client.



For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your project, call the `AddElasticsearchClient` extension method to register a `ElasticsearchClient` for use via the dependency injection container. The method takes a connection name parameter.



App host usage

To model the Elasticsearch resource in the app host, install the [Aspire.Hosting.Elasticsearch](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Elasticsearch
```

In the `Program.cs` file of `AppHost`, register a Elasticsearch cluster and consume the connection using the following methods:

C#

```
var elasticsearch = builder.AddElasticsearch("elasticsearch");

var myService = builder.AddProject<Projects.MyService>()
    .WithReference(elasticsearch);
```

The `WithReference` method configures a connection in the `MyService` project named `elasticsearch`. In the `Program.cs` file of `MyService`, the Elasticsearch connection can be consumed using:

C#

```
builder.AddElasticsearchClient("elasticsearch");
```

Configuration

The .NET Aspire Elasticsearch client integration provides multiple options to configure the server connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

```
builder.AddElasticsearchClient();
```

C#

```
builder.AddElasticsearchClient("elasticsearch");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{  
  "ConnectionStrings": {  
    "elasticsearch": "http://elastic:password@localhost:27011"  
  }  
}
```

Use configuration providers

The .NET Aspire Elasticsearch Client integration supports [Microsoft.Extensions.Configuration](#). It loads the `ElasticClientsElasticsearchSettings` from configuration by using the `Aspire:Elastic:Clients:Elasticsearch` key. Consider the following example `appsettings.json` that configures some of the options:

JSON

```
{  
  "Aspire": {  
    "Elastic": {  
      "Clients": {  
        "Elasticsearch": {  
          "Endpoint": "http://elastic:password@localhost:27011"  
        }  
      }  
    }  
  }  
}
```

Use inline delegates

Also you can pass the `Action<ElasticClientsElasticsearchSettings> configureSettings` delegate to set up some or all the options inline, for example to set the API key from code:

C#

```
builder.AddElasticsearchClient(  
  "elasticsearch",
```

```
    settings =>
        settings.Endpoint = new
Uri("http://elastic:password@localhost:27011"));
```

Use a CloudId and an ApiKey with configuration providers

When using [Elastic Cloud](#), you can provide the CloudId and ApiKey in `Aspire:Elastic:Clients:Elasticsearch` section when calling `builder.AddElasticsearchClient()`.

C#

```
builder.AddElasticsearchClient("elasticsearch");
```

Consider the following example `appsettings.json` that configures the options:

JSON

```
{
  "Aspire": {
    "Elastic": {
      "Clients": {
        "Elasticsearch": {
          "ApiKey": "<Valid ApiKey>",
          "CloudId": "<Valid CloudId>"
        }
      }
    }
  }
}
```

Use a CloudId and an ApiKey with inline delegates

C#

```
builder.AddElasticsearchClient(
  "elasticsearch",
  settings =>
  {
    settings.ApiKey = "<Valid ApiKey>";
    settings.CloudId = "<Valid CloudId>";
});
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Elasticsearch integration uses the configured client to perform a `PingAsync`. If the result is an HTTP 200 OK, the health check is considered healthy, otherwise it's unhealthy. Likewise, if there's an exception, the health check is considered unhealthy with the error propagating through the health check failure.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Tracing

The .NET Aspire Elasticsearch integration will emit the following tracing activities using OpenTelemetry:

- `Elastic.Transport`

See also

- [Elasticsearch .NET ↗](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Keycloak integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Keycloak integration. The `Aspire.Keycloak.Authentication` library registers JwtBearer and OpenId Connect authentication handlers in the DI container for connecting to a Keycloak server.

Prerequisites

- A Keycloak server instance.
- A Keycloak realm.
- For JwtBearer authentication, a configured audience in the Keycloak realm.
- For OpenId Connect authentication, the ID of a client configured in the Keycloak realm.

Get started

To get started with the .NET Aspire Keycloak integration, install the [Aspire.Keycloak.Authentication](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Keycloak client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Keycloak.Authentication
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Jwt bearer authentication usage example

In the `Program.cs` file of your ASP.NET Core API project, call the `AddKeycloakJwtBearer` extension method to add JwtBearer authentication, using a connection name, realm and any required JWT Bearer options:

C#

```
builder.Services.AddAuthentication()
    .AddKeycloakJwtBearer("keycloak", realm: "WeatherShop",
options =>
{
    options.Audience = "weather.api";
});
```

You can set many other options via the `Action<JwtBearerOptions> configureOptions` delegate.

OpenId Connect authentication usage example

In the `Program.cs` file of your Blazor project, call the `AddKeycloakOpenIdConnect` extension method to add OpenId Connect authentication, using a connection name, realm and any required OpenId Connect options:

C#

```
builder.Services.AddAuthentication(OpenIdConnectDefaults.AuthenticationScheme)
    .AddKeycloakOpenIdConnect(
        "keycloak",
        realm: "WeatherShop",
        options =>
    {
        options.ClientId = "WeatherWeb";
        options.ResponseType =
            OpenIdConnectResponseType.Code;
        options.Scope.Add("weather:all");
    });
});
```

You can set many other options via the `Action<OpenIdConnectOptions>? configureOptions` delegate.

App host usage

To model the Keycloak resource in the app host, install the [Aspire.Hosting.Keycloak](#) NuGet package in the `app host` project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Keycloak
```

Then, in the *Program.cs* file of `AppHost`, register a Keycloak server and consume the connection using the following methods:

C#

```
var keycloak = builder.AddKeycloak("keycloak", 8080);

var apiService = builder.AddProject<Projects.Keycloak_ApiService>
("apiservice")
    .WithReference(keycloak);

builder.AddProject<Projects.Keycloak_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(keycloak)
    .WithReference(apiService);
```

💡 Tip

For local development use a stable port for the Keycloak resource (8080 in the example above). It can be any port, but it should be stable to avoid issues with browser cookies that will persist OIDC tokens (which include the authority URL, with port) beyond the lifetime of the *app host*.

The `WithReference` method configures a connection in the `Keycloak.ApiService` and `Keycloak.Web` projects named `keycloak`.

In the *Program.cs* file of `Keycloak ApiService`, the Keycloak connection can be consumed using:

C#

```
builder.Services.AddAuthentication()
    .AddKeycloakJwtBearer("keycloak", realm: "WeatherShop");
```

In the *Program.cs* file of `Keycloak.Web`, the Keycloak connection can be consumed using:

C#

```
var oidcScheme = OpenIdConnectDefaults.AuthenticationScheme;

builder.Services.AddAuthentication(oidcScheme)
```

```
.AddKeycloakOpenIdConnect(  
    "keycloak",  
    realm: "WeatherShop",  
    oidcScheme);
```

See also

- [Keycloak ↗](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Milvus database integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Milvus database integration. The `Aspire.Milvus.Client` library registers a [MilvusClient](#) in the DI container for connecting to a Milvus server.

Prerequisites

- Milvus server and connection string for accessing the server API endpoint.

Get started

To get started with the .NET Aspire Milvus database integration, install the [Aspire.Milvus.Client](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Milvus database client.

```
.NET CLI
.NET CLI
dotnet add package Aspire.Milvus.Client
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your project, call the `AddMilvusClient` extension method to register a `MilvusClient` for use via the dependency injection container. The method takes a connection name parameter.

```
C#
builder.AddMilvusClient("milvus");
```

App host usage

To model the Milvus resource in the app host, install the [Aspire.Hosting.Milvus](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Milvus
```

In the *Program.cs* file of `AppHost`, register a Milvus server and consume the connection using the following methods:

C#

```
var milvus = builder.AddMilvus("milvus");

var myService = builder.AddProject<Projects.MyService>()
    .WithReference(milvus);
```

The `WithReference` method configures a connection in the `MyService` project named `milvus`. In the *Program.cs* file of `MyService`, the Milvus connection can be consumed using:

C#

```
builder.AddMilvusClient("milvus");
```

Milvus supports configuration-based (environment variable `COMMON_SECURITY_DEFAULTROOTPASSWORD`) default passwords. The default user is `root` and the default password is `Milvus`. To change the default password in the container, pass an `apiKey` parameter when calling the `AddMilvus` hosting API:

C#

```
var apiKey = builder.AddParameter("apiKey");

var milvus = builder.AddMilvus("milvus", apiKey);

var myService = builder.AddProject<Projects.MyService>()
    .WithReference(milvus);
```

The preceding code gets a parameter to pass to the `AddMilvus` API, and internally assigns the parameter to the `COMMON_SECURITY_DEFAULTROOTPASSWORD` environment

variable of the Milvus container. The `apiKey` parameter is usually specified as a *user secret*:

JSON

```
{  
  "Parameters": {  
    "apiKey": "Non-default P@ssw0rd"  
  }  
}
```

For more information, see [External parameters](#).

Configuration

The .NET Aspire Milvus Client integration provides multiple options to configure the server connection based on the requirements and conventions of your project.

 Tip

The default user is `root` and the default password is `Milvus`. Currently, Milvus doesn't support changing the superuser password at startup. It needs to be manually changed with the client.

Use a connection string

When using a connection string from the `ConnectionString`s configuration section, you can provide the name of the connection string when calling `builder.AddMilvusClient()`:

C#

```
builder.AddMilvusClient("milvus");
```

And then the connection string will be retrieved from the `ConnectionString`s configuration section:

JSON

```
{  
  "ConnectionString": {  
    "milvus": "Endpoint=http://localhost:19530/;Key=root:123456!@#$%"  
  }  
}
```

```
    }  
}
```

By default the `MilvusClient` uses the gRPC API endpoint.

Use configuration providers

The .NET Aspire Milvus Client integration supports `Microsoft.Extensions.Configuration`. It loads the `MilvusSettings` from configuration by using the `Aspire:Milvus:Client` key. Consider the following example `appsettings.json` that configures some of the options:

JSON

```
{  
  "Aspire": {  
    "Milvus": {  
      "Client": {  
        "Key": "root:123456!@#$%"  
      }  
    }  
  }  
}
```

Use inline delegates

Also you can pass the `Action<MilvusSettings> configureSettings` delegate to set up some or all the options inline, for example to set the API key from code:

C#

```
builder.AddMilvusClient(  
  "milvus",  
  settings => settings.Key = "root:12345!@#$%");
```

Health checks

By default, .NET Aspire integrations enable `health checks` for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Milvus database integration uses the configured client to perform a `HealthAsync`. If the result is *healthy*, the health check is considered healthy, otherwise it's unhealthy. Likewise, if there's an exception, the health check is considered unhealthy with the error propagating through the health check failure.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Milvus database integration uses standard .NET logging, and you'll see log entries from the following category:

- `Milvus.Client`

See also

- [Milvus .NET SDK ↗](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire MongoDB database integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire MongoDB database integration. The `Aspire.MongoDB.Driver` library:

- Registers a [IMongoClient](#) in the DI container for connecting to a MongoDB database.
- Automatically configures the following:
 - Health checks, logging and telemetry to improve app monitoring and diagnostics
- It supports both a local MongoDB Database and a [MongoDB Atlas](#) database deployed in the cloud.

Prerequisites

- MongoDB database and connection string for accessing the database.

Get started

To get started with the .NET Aspire MongoDB database integration, install the [Aspire.MongoDB.Driver](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the MongoDB database client.

```
.NET CLI
.NET CLI
dotnet add package Aspire.MongoDB.Driver
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the `AddMongoDBClient` extension to register a `IMongoClient` for use via the dependency injection container.

```
C#
```

```
builder.AddMongoDBClient("mongodb");
```

To retrieve your `IMongoClient` object, consider the following example service:

```
C#
```

```
public class ExampleService(IMongoClient mongoClient)
{
    // Use mongoClient...
}
```

After adding a `IMongoClient`, you can require the `IMongoClient` instance using DI.

App host usage

To model the MongoDB resource in the app host, install the [Aspire.Hosting.MongoDB](#) NuGet package in the `app host` project.

```
.NET CLI
```

```
.NET CLI
```

```
dotnet add package Aspire.Hosting.MongoDB
```

In your app host project, register the MongoDB database and consume the connection method and consume the service using the following methods:

```
C#
```

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

var mongo = builder.AddMongoDB("mongo");
var mongodb = mongo.AddDatabase("mongodb");

var myService = builder.AddProject<Projects.MyService>()
    .WithReference(mongodb);
```

If using MongoDB Atlas, you don't need the call to `.AddDatabase("mongodb")`; as the database creation will automatically be handled by Atlas.

Configuration

The .NET Aspire MongoDB database integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

```
builder.AddMongoDBClient();
```

C#

```
builder.AddMongoDBClient("MongoConnection");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

Consider the following MongoDB example JSON configuration:

JSON

```
{
  "ConnectionStrings": {
    "MongoConnection": "mongodb://server:port/test",
  }
}
```

Consider the following MongoDB Atlas example JSON configuration:

JSON

```
{
  "ConnectionStrings": {
    "MongoConnection":
      "mongodb+srv://username:password@server.mongodb.net/",
  }
}
```

For more information on how to format this connection string, see [MongoDB: ConnectionString documentation](#).

Use configuration providers

The .NET Aspire MongoDB database integration supports [Microsoft.Extensions.Configuration](#). It loads the `MongoDBSettings` from configuration files such as `appsettings.json` by using the `Aspire:MongoDB:Driver` key.

The following example shows an `appsettings.json` file that configures some of the available options:

JSON

```
{  
  "Aspire": {  
    "MongoDB": {  
      "Driver": {  
        "ConnectionString": "mongodb://server:port/test",  
        "DisableHealthChecks": false,  
        "HealthCheckTimeout": 10000,  
        "DisableTracing": false  
      },  
    }  
  }  
}
```

Use inline configurations

You can also pass the `Action<MongoDBSettings>` delegate to set up some or all the options inline:

C#

```
builder.AddMongoDBClient("mongodb",  
    static settings => settings.ConnectionString =  
    "mongodb://server:port/test");
```

Configuration options

Here are the configurable options with corresponding default values:

[] Expand table

Name	Description
<code>ConnectionString</code>	The connection string of the MongoDB database database to connect to.

Name	Description
<code>DisableHealthChecks</code>	A boolean value that indicates whether the database health check is disabled or not.
<code>HealthCheckTimeout</code>	An <code>int?</code> value that indicates the MongoDB health check timeout in milliseconds.
<code>DisableTracing</code>	A boolean value that indicates whether the OpenTelemetry tracing is disabled or not.

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

By default, the .NET Aspire MongoDB database integration handles the following:

- Adds a health check when enabled that verifies that a connection can be made commands can be run against the MongoDB database within a certain amount of time.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire MongoDB database integration uses standard .NET logging, and you'll see log entries from the following categories:

- `MongoDB[.*]`: Any log entries from the MongoDB namespace.

Tracing

The .NET Aspire MongoDB database integration will emit the following Tracing activities using OpenTelemetry:

- "MongoDB.Driver.Core.Extensions.DiagnosticSources"

Metrics

The .NET Aspire MongoDB database integration doesn't currently expose any OpenTelemetry metrics.

See also

- [MongoDB database ↗](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire MySQL database integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire MySQL database integration. The `Aspire.MySqlConnector` library:

- Registers a [MySqlDataSource](#) in the DI container for connecting MySQL database.
- Automatically configures the following:
 - Health checks, logging and telemetry to improve app monitoring and diagnostics

Prerequisites

- MySQL database and connection string for accessing the database.

Get started

To get started with the .NET Aspire MySQL database integration, install the [Aspire.MySqlConnector](#) NuGet package.

```
.NET CLI
dotnet add package Aspire.MySqlConnector
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the `AddMySqlDataSource` extension to register a `MySqlDataSource` for use via the dependency injection container.

```
C#
builder.AddMySqlDataSource("mysqlDb");
```

To retrieve your `MySqlDataSource` object, consider the following example service:

C#

```
public class ExampleService(MySqlDataSource dataSource)
{
    // Use dataSource...
}
```

After adding a `MySqlDataSource`, you can require the `MySqlDataSource` instance using DI.

App host usage

To model the MySQL resource in the app host, install the [Aspire.Hosting.MySql](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.MySql
```

In your app host project, register a MySQL database and consume the connection using the following methods:

C#

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

var mysql = builder.AddMySql("mysql");
var mysqlDb = mysql.AddDatabase("mysqlDb");

var myService = builder.AddProject<Projects.MyService>()
    .WithReference(mysqlDb);
```

When you want to explicitly provide a root MySQL password, you can provide it as a parameter. Consider the following alternative example:

C#

```
var password = builder.AddParameter("password", secret: true);

var mysql = builder.AddMySql("mysql", password);
var mysqlDb = mysql.AddDatabase("mysqlDb");
```

```
var myService = builder.AddProject<Projects.MyService>()
    .WithReference(mysqlDb);
```

For more information, see [External parameters](#).

Configuration

The .NET Aspire MySQL database integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

```
builder.AddMySqlDataSource();
```

C#

```
builder.AddMySqlDataSource("MySqlConnection");
```

Then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "MySqlConnection": "Server=mysql;Database=mysqlDb"
  }
}
```

For more information on how to format this connection string, see [MySqlConnector:ConnectionString documentation ↗](#).

Use configuration providers

The .NET Aspire MySQL database supports `Microsoft.Extensions.Configuration`. It loads the `MySqlConnectorSettings` from configuration files such as `appsettings.json` by using the `Aspire:MySqlConnector` key. If you have set up your configurations in the `Aspire:MySqlConnector` section, you can just call the method without passing any parameter.

The following example shows an `appsettings.json` file that configures some of the available options:

JSON

```
{  
  "Aspire": {  
    " MySqlConnector": {  
      "DisableHealthChecks": true,  
      "DisableTracing": true  
    }  
  }  
}
```

Use inline configurations

You can also pass the `Action<MySqlConnectorSettings>` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddMySqlDataSource("mysql",  
    static settings => settings.DisableHealthChecks = true);
```

Configuration options

Here are the configurable options with corresponding default values:

 Expand table

Name	Description
<code>ConnectionString</code>	The connection string of the MySQL database database to connect to.
<code>DisableHealthChecks</code>	A boolean value that indicates whether the database health check is disabled or not.
<code>DisableTracing</code>	A boolean value that indicates whether the OpenTelemetry tracing is disabled or not.
<code>DisableMetrics</code>	A boolean value that indicates whether the OpenTelemetry metrics are disabled or not.

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

By default, the .NET Aspire MySQL database integration handles the following:

- Adds a `MySqlHealthCheck`, which verifies that a connection can be made commands can be run against the MySQL database.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

See also

- [MySQL database ↗](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire NATS integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire NATS integration to send logs and traces to a NATS Server. The integration supports persistent logs and traces across application restarts via configuration.

Prerequisites

- [Install the NATS server](#)
- The URL to access the server.

Get started

To get started with the .NET Aspire NATS integration, install the [Aspire.NATS.Net](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the NATS client.

```
.NET CLI
.NET CLI
dotnet add package Aspire.NATS.Net
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your projects, call the `AddNatsClient` extension method to register an `INatsConnection` to send logs and traces to NATS and the .NET Aspire Dashboard. The method takes a connection name parameter.

```
C#
builder.AddNatsClient("nats");
```

You can then retrieve the `INatsConnection` instance using dependency injection. For example, to retrieve the client from a service:

```
C#  
  
public class ExampleService(INatsConnection client)  
{  
    // Use client...  
}
```

App host usage

To model the Nats resource in the app host, install the [Aspire.Hosting.Nats](#) NuGet package in the `app host` project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Nats
```

Then, in the `Program.cs` file of `AppHost`, register a NATS server and consume the connection using the following methods:

C#

```
var builder = DistributedApplication.CreateBuilder(args);  
  
var nats = builder.AddNats("nats");  
  
var myService = builder.AddProject<Projects.MyService>()  
    .WithReference(nats);
```

The `WithReference` method configures a connection in the `MyService` project named `nats`. In the `Program.cs` file of `MyService`, the NATS connection can be consumed using:

C#

```
builder.AddNatsClient("nats");
```

Configuration

The .NET Aspire NATS integration provides multiple options to configure the NATS connection based on the requirements and conventions of your project.

Use a connection string

Provide the name of the connection string when you call `builder.AddNatsClient()`:

C#

```
builder.AddNatsClient("myConnection");
```

The connection string is retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "myConnection": "nats://nats:4222"
  }
}
```

See the [ConnectionString documentation](#) for more information on how to format this connection string.

Use configuration providers

The .NET Aspire NATS integration supports [Microsoft.Extensions.Configuration](#). It loads the `NatsClientSettings` from configuration using the `Aspire:Nats:Client` key. Example `appsettings.json` that configures some of the options:

JSON

```
{
  "Aspire": {
    "Nats": {
      "Client": {
        "DisableHealthChecks": true
      }
    }
  }
}
```

Use inline delegates

Pass the `Action<NatsClientSettings> configureSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

```
C#
```

```
builder.AddNatsClient("nats", settings => settings.DisableHealthChecks = true);
```

Persistent logs and traces

Register NATS with a data directory in your `.AppHost` project to retain NATS's data and configuration across application restarts.

```
C#
```

```
var NATS = builder.AddNATS("NATS", NATSDataDirectory: "./NATSdata");
```

The directory specified must already exist.

NATS in the .NET Aspire manifest

NATS isn't part of the .NET Aspire deployment manifest. It's recommended you set up a secure production NATS server outside of .NET Aspire.

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire NATS integration handles the following:

- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not

metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire NATS integration uses the following log categories:

- `NATS`

Tracing

The .NET Aspire NATS integration emits the following tracing activities:

- `NATS.Net`

See also

- [NATS.Net quickstart ↗](#)
- [NATS integration README ↗](#)

.NET Aspire Oracle Entity Framework Component

Article • 08/29/2024

In this article, you learn how to use the The .NET Aspire Oracle Entity Framework Core integration. The `Aspire.Oracle.EntityFrameworkCore` library is used to register a `System.Data.Entity.DbContext` as a singleton in the DI container for connecting to Oracle databases. It also enables connection pooling, retries, health checks, logging and telemetry.

Get started

You need an Oracle database and connection string for accessing the database. To get started with the The .NET Aspire Oracle Entity Framework Core integration, install the [Aspire.Oracle.EntityFrameworkCore](#) NuGet package in the consuming client project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Oracle.EntityFrameworkCore
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the `AddOracleDatabaseDbContext` extension to register a `System.Data.Entity.DbContext` for use via the dependency injection container.

C#

```
builder.AddOracleDatabaseDbContext<MyDbContext>("oracledb");
```

You can then retrieve the `DbContext` instance using dependency injection. For example, to retrieve the client from a service:

C#

```
public class ExampleService(MyDbContext context)
{
    // Use context...
}
```

You might also need to configure specific options of Oracle database, or register a `DbContext` in other ways. In this case call the `EnrichOracleDatabaseDbContext` extension method, for example:

C#

```
var connectionString =
builder.Configuration.GetConnectionString("oracledb");

builder.Services.AddDbContextPool<MyDbContext>(
    dbContextOptionsBuilder =>
    dbContextOptionsBuilder.UseOracle(connectionString));

builder.EnrichOracleDatabaseDbContext<MyDbContext>();
```

App host usage

To model the Oracle server resource in the app host, install the [Aspire.Hosting.Oracle](#) NuGet package in the `app host` project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Oracle
```

In your app host project, register an Oracle container and consume the connection using the following methods:

C#

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

var oracle = builder.AddOracle("oracle");
var oracledb = oracle.AddDatabase("oracledb");
```

```
var myService = builder.AddProject<Projects.MyService>()
    .WithReference(oracledb);
```

When you want to explicitly provide a password, you can provide it as a parameter. Consider the following alternative example:

C#

```
var password = builder.AddParameter("password", secret: true);

var oracle = builder.AddOracle("oracle", password);
var oracledb = oracle.AddDatabase("oracledb");

var myService = builder.AddProject<Projects.MyService>()
    .WithReference(oracledb);
```

For more information, see [External parameters](#).

Configuration

The .NET Aspire Oracle Entity Framework Core integration provides multiple options to configure the database connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

```
builder.AddOracleDbContext<TContext>();
```

C#

```
builder.AddOracleDbContext<MyDbContext>("myConnection");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "myConnection": "Data Source=TORCL;User
Id=myUsername;Password=myPassword;"}
```

```
    }  
}
```

The `EnrichOracleDbContext` won't make use of the `ConnectionStrings` configuration section since it expects a `DbContext` to be registered at the point it is called.

For more information, see the [ODP.NET documentation](#).

Use configuration providers

The .NET Aspire Oracle Entity Framework Core integration supports `Microsoft.Extensions.Configuration`. It loads the `OracleEntityFrameworkCoreSettings` from configuration by using the `Aspire:Oracle:EntityFrameworkCore` key.

The following example shows an `appsettings.json` that configures some of the available options:

JSON

```
{  
  "Aspire": {  
    "Oracle": {  
      "EntityFrameworkCore": {  
        "DisableHealthChecks": true,  
        "DisableTracing": true,  
        "DisableMetrics": false,  
        "DisableRetry": false,  
        "Timeout": 30  
      }  
    }  
  }  
}
```

💡 Tip

The `Timeout` property is in seconds. When set as shown in the preceding example, the timeout is 30 seconds.

Use inline delegates

You can also pass the `Action<OracleEntityFrameworkCoreSettings> configureSettings` delegate to set up some or all the options inline, for example to disable health checks

from code:

```
C#
```

```
builder.AddOracleDatabaseDbContext<MyDbContext>(
    "oracle",
    static settings => settings.DisableHealthChecks = true);
```

or

```
C#
```

```
builder.EnrichOracleDatabaseDbContext<MyDbContext>(
    static settings => settings.DisableHealthChecks = true);
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The The .NET Aspire Oracle Entity Framework Core integration registers a basic health check that checks the database connection given a `TContext`. The health check is enabled by default and can be disabled using the `DisableHealthChecks` property in the configuration.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The The .NET Aspire Oracle Entity Framework Core integration uses the following log categories:

- `Microsoft.EntityFrameworkCore.Database.Command.CommandCreated`

- Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting
- Microsoft.EntityFrameworkCore.Database.Command.CommandExecuted
- Microsoft.EntityFrameworkCore.Database.Command.CommandError

Tracing

The The .NET Aspire Oracle Entity Framework Core integration will emit the following tracing activities using OpenTelemetry:

- OpenTelemetry.Instrumentation.EntityFrameworkCore

Metrics

The The .NET Aspire Oracle Entity Framework Core integration currently supports the following metrics:

- Microsoft.EntityFrameworkCore

See also

- [Entity Framework Core docs](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

Use Orleans with .NET Aspire

Article • 08/12/2024

Orleans has in-built support for .NET Aspire. .NET Aspire's application model lets you describe the services, databases, and other resources/infrastructure in your app and how they relate. Orleans provides a straightforward way to build distributed applications which are elastically scalable and fault-tolerant. .NET Aspire is used to configure and orchestrate Orleans and its dependencies, such as, by providing Orleans with database cluster membership and storage.

Orleans is represented as a resource in .NET Aspire. The Orleans resource includes configuration which your service needs to operate, such as cluster membership providers and storage providers.

Prerequisites

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

- [.NET 8.0](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or [the .NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

In addition to the prerequisites for .NET Aspire, you will need:

- Orleans version 8.1.0 or later

For more information, see [.NET Aspire setup and tooling](#).

Get started

To get started you need to add the Orleans hosting package to your app host project by installing the [Aspire.Hosting.Orleans](#) NuGet package.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Orleans
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

The Orleans resource is added to the .NET Aspire distributed application builder using the `AddOrleans(string name)` method, which returns an Orleans resource builder. The name provided to the Orleans resource is for diagnostic purposes. For most applications, a value of `"default"` will suffice.

C#

```
var orleans = builder.AddOrleans("default")
```

The Orleans resource builder offers methods to configure your Orleans resource. In the following example, the Orleans resource is configured with clustering and grain storage using the `WithClustering` and `WithGrainStorage` methods respectively:

C#

```
// Add the resources which you will use for Orleans clustering and
// grain state storage.
var storage = builder.AddAzureStorage("storage").RunAsEmulator();
var clusteringTable = storage.AddTables("clustering");
var grainStorage = storage.AddBlobs("grain-state");

// Add the Orleans resource to the Aspire DistributedApplication
// builder, then configure it with Azure Table Storage for clustering
// and Azure Blob Storage for grain storage.
var orleans = builder.AddOrleans("default")
    .WithClustering(clusteringTable)
    .WithGrainStorage("Default", grainStorage);
```

This tells Orleans that any service referencing it will also need to reference the `clusteringTable` resource.

To participate in an Orleans cluster, reference the Orleans resource from your service project, either using `WithReference(orleans)` to participate as an Orleans server, or `WithReference(orleans.AsClient())` to participate as a client. When you reference the Orleans resource from your service, those resources will also be referenced:

C#

```
// Add our server project and reference your 'orleans' resource from it.  
// it can join the Orleans cluster as a service.  
// This implicitly add references to the required resources.  
// In this case, that is the 'clusteringTable' resource declared earlier.  
builder.AddProject<Projects.OrleansServer>("silo")  
    .WithReference(orleans)  
    .WithReplicas(3);
```

Putting that all together, here is an example of an Aspire AppHost project which includes:

- An Orleans resource with clustering and storage.
- An Orleans server project, 'OrleansServer'.
- An Orleans client project, 'OrleansClient'.

C#

```
var builder = DistributedApplication.CreateBuilder(args);  
  
// Add the resources which you will use for Orleans clustering and  
// grain state storage.  
var storage = builder.AddAzureStorage("storage").RunAsEmulator();  
var clusteringTable = storage.AddTables("clustering");  
var grainStorage = storage.AddBlobs("grain-state");  
  
// Add the Orleans resource to the Aspire DistributedApplication  
// builder, then configure it with Azure Table Storage for clustering  
// and Azure Blob Storage for grain storage.  
var orleans = builder.AddOrleans("default")  
    .WithClustering(clusteringTable)  
    .WithGrainStorage("Default", grainStorage);  
  
// Add our server project and reference your 'orleans' resource from it.  
// it can join the Orleans cluster as a service.  
// This implicitly add references to the required resources.  
// In this case, that is the 'clusteringTable' resource declared earlier.  
builder.AddProject<Projects.OrleansServer>("silo")  
    .WithReference(orleans)  
    .WithReplicas(3);  
  
// Reference the Orleans resource as a client from the 'frontend'  
// project so that it can connect to the Orleans cluster.  
builder.AddProject<Projects.OrleansClient>("frontend")  
    .WithReference(orleans.AsClient())  
    .WithExternalHttpEndpoints()  
    .WithReplicas(3);  
  
// Build and run the application.
```

```
using var app = builder.Build();
await app.RunAsync();
```

To consume the .NET Aspire Orleans resource from an Orleans server project, there are a few steps:

1. Add the relevant .NET Aspire integrations. In this example, you're using *Aspire.Azure.Data.Tables* and *Aspire.Azure.Storage.Blobs*.
2. Add the Orleans provider packages for those .NET Aspire integrations. In this example, you're using *Microsoft.Orleans.Persistence.AzureStorage* and *Microsoft.Orleans.Clustering.AzureStorage*.
3. Add Orleans to the host application builder.

In the *Program.cs* file of your Orleans server project, you must configure the .NET Aspire integrations you are using and add Orleans to the host builder. Note that the names provided must match the names used in the .NET Aspire app host project: "clustering" for the clustering provider, and "grain-state" for the grain state storage provider:

C#

```
using Orleans.Runtime;
using OrleansContracts;

var builder = WebApplication.CreateBuilder(args);

builder.AddServiceDefaults();
builder.AddKeyedAzureTableClient("clustering");
builder.AddKeyedAzureBlobClient("grain-state");
builder.UseOrleans();

var app = builder.Build();

app.MapGet("/", () => "OK");

await app.RunAsync();

public sealed class CounterGrain(
    [PersistentState("count")] IPersistentState<int> count) : ICounterGrain
{
    public ValueTask<int> Get()
    {
        return ValueTask.FromResult(count.State);
    }

    public async ValueTask<int> Increment()
    {
        var result = ++count.State;
        await count.WriteStateAsync();
        return result;
    }
}
```

```
    }  
}
```

You do similarly in the *OrleansClient* project, adding the .NET Aspire resources which your project needs to join the Orleans cluster as a client, and configuring the host builder to add an Orleans client:

```
C#  
  
using OrleansContracts;  
  
var builder = WebApplication.CreateBuilder(args);  
  
builder.AddServiceDefaults();  
builder.AddKeyedAzureTableClient("clustering");  
builder.UseOrleansClient();  
  
var app = builder.Build();  
  
app.MapGet("/counter/{grainId}", async (IClusterClient client, string grainId) =>  
{  
    var grain = client.GetGrain<ICounterGrain>(grainId);  
    return await grain.Get();  
});  
  
app.MapPost("/counter/{grainId}", async (IClusterClient client, string grainId) =>  
{  
    var grain = client.GetGrain<ICounterGrain>(grainId);  
    return await grain.Increment();  
});  
  
app.UseFileServer();  
  
await app.RunAsync();
```

Enabling OpenTelemetry

By convention, .NET Aspire project include a project for defining default configuration and behavior for your service. This project is called the *service default* project and templates create it with a name ending in *ServiceDefaults*. To configure Orleans for OpenTelemetry in .NET Aspire, you will need to apply configuration to your service defaults project following the [Orleans observability](#) guide. In short, you will need to modify the `ConfigureOpenTelemetry` method to add the Orleans *meters* and *tracing* instruments. The following code snippet shows the *Extensions.cs* file from a service defaults project which has been modified to include metrics and traces from Orleans.

C#

```
public static IHostApplicationBuilder ConfigureOpenTelemetry(this
IHostApplicationBuilder builder)
{
    builder.Logging.AddOpenTelemetry(logging =>
    {
        logging.IncludeFormattedMessage = true;
        logging.IncludeScopes = true;
    });

    builder.Services.AddOpenTelemetry()
        .WithMetrics(metrics =>
    {
        metrics.AddAspNetCoreInstrumentation()
            .AddHttpClientInstrumentation()
            .AddRuntimeInstrumentation()
            .AddMeter("Microsoft.Orleans");
    })
        .WithTracing(tracing =>
    {
        tracing.AddSource("Microsoft.Orleans.Runtime");
        tracing.AddSource("Microsoft.Orleans.Application");

        tracing.AddAspNetCoreInstrumentation()
            .AddHttpClientInstrumentation();
    });
}

builder.AddOpenTelemetryExporters();

return builder;
}
```

Supported providers

The Orleans Aspire integration supports a limited subset of Orleans providers today:

- Clustering:
 - Redis
 - Azure Storage Tables
- Persistence:
 - Redis
 - Azure Storage Tables
 - Azure Storage Blobs
- Reminders:
 - Redis
 - Azure Storage Tables
- Grain directory:

- Redis
- Azure Storage Tables

Streaming providers are not supported as of Orleans version 8.1.0.

Next steps

[Explore the Orleans voting sample app](#)

.NET Aspire Pomelo MySQL Entity Framework Component

Article • 08/29/2024

In this article, you learn how to use the The .NET Aspire Pomelo MySQL Entity Framework Core integration. The `Aspire.Pomelo.EntityFrameworkCore.MySql` library is used to register a `System.Data.Entity.DbContext` as a singleton in the DI container for connecting to MySQL databases. It also enables connection pooling, retries, health checks, logging and telemetry.

Get started

You need a MySQL database and connection string for accessing the database. To get started with the The .NET Aspire Pomelo MySQL Entity Framework Core integration, install the [Aspire.Pomelo.EntityFrameworkCore.MySql](#) NuGet package in the consuming client project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Pomelo.EntityFrameworkCore.MySql
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the `AddMySqlDbContext` extension to register a `System.Data.Entity.DbContext` for use via the dependency injection container.

C#

```
builder.AddMySqlDbContext<MyDbContext>("mysqlDb");
```

You can then retrieve the `DbContext` instance using dependency injection. For example, to retrieve the client from a service:

C#

```
public class ExampleService(MyDbContext context)
{
    // Use context...
}
```

You might also need to configure specific options of your MySQL connection, or register a `DbContext` in other ways. In this case call the `EnrichMySqlDbContext` extension method, for example:

C#

```
var connectionString = builder.Configuration.GetConnectionString("mysqlDb");

builder.Services.AddDbContextPool<MyDbContext>(
    dbContextOptionsBuilder =>
    dbContextOptionsBuilder.UseMySql(connectionString, serverVersion));
builder.EnrichMySqlDbContext<MyDbContext>();
```

App host usage

To model the MySQL resource in the app host, install the [Aspire.Hosting.MySql](#) NuGet package in the `app host` project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.MySql
```

In your app host project, register a MySQL database and consume the connection using the following methods:

C#

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

var mysql = builder.AddMySql("mysql");
var mysqlDb = mysql.AddDatabase("mysqlDb");

var myService = builder.AddProject<Projects.MyService>()
    .WithReference(mysqlDb);
```

Configuration

The .NET Aspire Pomelo MySQL Entity Framework Core integration provides multiple options to configure the database connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

```
builder.AddMySqlDbContext<TContext>():
```

C#

```
builder.AddMySqlDbContext<MyDbContext>("myConnection");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "myConnection": "Server=myserver;Database=mysqldb"
  }
}
```

The `EnrichMySqlDbContext` won't make use of the `ConnectionStrings` configuration section since it expects a `DbContext` to be registered at the point it is called.

For more information, see the [MySqlConnector documentation ↗](#).

Use configuration providers

The .NET Aspire Pomelo MySQL Entity Framework Core integration supports [Microsoft.Extensions.Configuration](#). It loads the

```
PomeloEntityFrameworkCoreMySqlSettings
```

 from configuration by using the `Aspire:Pomelo:EntityFrameworkCore:MySQL` key.

The following example shows an `appsettings.json` that configures some of the available options:

JSON

```
{  
  "Aspire": {  
    "Pomelo": {  
      "EntityFrameworkCore": {  
        "MySql": {  
          "DisableHealthChecks": true,  
          "DisableTracing": true  
        }  
      }  
    }  
  }  
}
```

Use inline delegates

You can also pass the `Action<PomeloEntityFrameworkCoreMySqlSettings>` `configureSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddMySqlDbContext<MyDbContext>(  
  "mysqlDb1",  
  static settings => settings.DisableHealthChecks = true);
```

or

C#

```
builder.EnrichMySqlDbContext<MyDbContext>(  
  static settings => settings.DisableHealthChecks = true);
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The The .NET Aspire Pomelo MySQL Entity Framework Core integration registers a basic health check that checks the database connection given a `TContext`. The health check is enabled by default and can be disabled using the `DisableHealthChecks` property in the configuration.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The The .NET Aspire Pomelo MySQL Entity Framework Core integration uses the following log categories:

- `Microsoft.EntityFrameworkCore.Database.Command.CommandCreated`
- `Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting`
- `Microsoft.EntityFrameworkCore.Database.Command.CommandExecuted`
- `Microsoft.EntityFrameworkCore.Database.Command.CommandError`

Tracing

The The .NET Aspire Pomelo MySQL Entity Framework Core integration will emit the following tracing activities using OpenTelemetry:

- `OpenTelemetry.Instrumentation.EntityFrameworkCore`

Metrics

The The .NET Aspire Pomelo MySQL Entity Framework Core integration currently supports the following metrics:

- `Microsoft.EntityFrameworkCore`

See also

- [Entity Framework Core docs](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire PostgreSQL Entity Framework Core integration

Article • 09/07/2024

In this article, you learn how to use the .NET Aspire PostgreSQL Entity Framework Core integration. The `Aspire.Npgsql.EntityFrameworkCore.PostgreSQL` library is used to register a `DbContext` service for connecting to a PostgreSQL database. It also enables corresponding health checks, logging and telemetry.

PostgreSQL is a powerful, open source, object-relational database system. The .NET Aspire PostgreSQL Entity Framework integration streamlines essential database context and connection configurations for you by handling the following concerns:

- Registers `EntityFrameworkCore` in the DI container for connecting to PostgreSQL database.
- Automatically configures the following:
 - Connection pooling to efficiently managed HTTP requests and database connections
 - Automatic retries to increase app resiliency
 - Health checks, logging and telemetry to improve app monitoring and diagnostics

Prerequisites

- Azure subscription: [create one for free ↗](#)
- Azure Postgresql Database: learn more about how to [create an Azure Database for PostgreSQL](#).

Get started

To get started with the .NET Aspire PostgreSQL Entity Framework Core integration, install the [Aspire.Npgsql.EntityFrameworkCore.PostgreSQL ↗](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the PostgreSQL Entity Framework Core client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Npgsql.EntityFrameworkCore.PostgreSQL
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the [AddNpgsqlDbContext](#) extension to register a `DbContext` for use via the dependency injection container.

C#

```
builder.AddNpgsqlDbContext<YourDbContext>("postgresdb");
```

You can then retrieve the `YourDbContext` instance using dependency injection. For example, to retrieve the client from a service:

C#

```
public class ExampleService(YourDbContext context)
{
    // Use context...
}
```

App host usage

To model the PostgreSQL server resource in the app host, install the [Aspire.Hosting.PostgreSQL](#) NuGet package in the [app host](#) project.

.NET CLI

```
.NET CLI
```

```
dotnet add package Aspire.Hosting.PostgreSQL
```

In your app host project, register and consume the PostgreSQL integration using the following methods, such as [AddPostgres](#):

C#

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

var postgres = builder.AddPostgres("postgres");
var postgresdb = postgres.AddDatabase("postgresdb");

var myService = builder.AddProject<Projects.MyService>()
    .WithReference(postgresdb);
```

When you want to explicitly provide the username and password, you can provide those as parameters. Consider the following alternative example:

C#

```
var username = builder.AddParameter("username", secret: true);
var password = builder.AddParameter("password", secret: true);

var postgres = builder.AddPostgres("postgres", username, password);

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
    .WithReference(postgresdb);
```

For more information, see [External parameters](#).

When adding PostgreSQL resources to the `builder` with the `AddPostgres` method, you can chain calls to `WithPgWeb` to add the [sosedoff/pgweb](#) container. This container is a cross-platform client for PostgreSQL databases, that serves a web-based admin dashboard. Consider the following example:

C#

```
var postgres = builder.AddPostgres("postgres")
    .WithPgWeb();

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
    .WithReference(postgresdb);
```

All registered `PostgresDatabaseResource` instances are used to create a configuration file per instance, and each config is bound to the `pgweb` container bookmark directory. For more information, see [PgWeb docs: Server connection bookmarks](#).

When adding PostgreSQL resources to the `builder` with the `AddPostgres` method, you can chain calls to `WithPgAdmin` to add the [dpage/pgadmin4](#) container. This container

is a cross-platform client for PostgreSQL databases, that serves a web-based admin dashboard. Consider the following example:

```
C#  
  
var postgres = builder.AddPostgres("postgres")  
    .WithPgAdmin();  
  
var postgresdb = postgres.AddDatabase("postgresdb");  
  
var exampleProject = builder.AddProject<Projects.ExampleProject>()  
    .WithReference(postgresdb);
```

Azure app host usage

To deploy your PostgreSQL resources to Azure, you need to install the appropriate .NET Aspire hosting package:

```
.NET CLI  
  
.NET CLI  
  
dotnet add package Aspire.Hosting.Azure.PostgreSQL
```

After you've installed this package, you specify that your PostgreSQL resources will be hosted in Azure by calling the [PublishAsAzurePostgresFlexibleServer](#) extension method in your app host project:

```
C#  
  
var builder = DistributedApplication.CreateBuilder(args);  
  
var postgres = builder.AddPostgres("postgres")  
    .PublishAsAzurePostgresFlexibleServer();  
  
var postgresdb = postgres.AddDatabase("postgresdb");  
  
var exampleProject = builder.AddProject<Projects.ExampleProject>()  
    .WithReference(postgresdb);
```

The preceding call to `PublishAsAzurePostgresFlexibleServer` configures Postgres Server resource to be deployed as Azure Postgres Flexible Server. For more information, see [Azure Postgres Flexible Server](#).

Configuration

The .NET Aspire PostgreSQL Entity Framework Core integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you provide the name of the connection string when calling `AddNpgsqlDbContext`:

C#

```
builder.AddNpgsqlDbContext<MyDbContext>("myConnection");
```

The connection string is retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "myConnection": "Host=myserver;Database=test"
  }
}
```

The `EnrichNpgsqlDbContext` won't make use of the `ConnectionStrings` configuration section since it expects a `DbContext` to be registered at the point it is called.

For more information, see the [ConnectionString](#).

Use configuration providers

The .NET Aspire PostgreSQL Entity Framework Core integration supports [Microsoft.Extensions.Configuration](#). It loads the `NpgsqlEntityFrameworkCorePostgreSQLSettings` from configuration files such as `appsettings.json` by using the `Aspire:Npgsql:EntityFrameworkCore:PostgreSQL` key. If you have set up your configurations in the `Aspire:Npgsql:EntityFrameworkCore:PostgreSQL` section you can just call the method without passing any parameter.

The following example shows an `appsettings.json` file that configures some of the available options:

JSON

```
{  
  "Aspire": {  
    "Npgsql": {  
      "EntityFrameworkCore": {  
        "PostgreSQL": {  
          "ConnectionString": "YOUR_CONNECTIONSTRING",  
          "DbContextPooling": true,  
          "DisableHealthChecks": true,  
          "DisableTracing": true  
        }  
      }  
    }  
  }  
}
```

Use inline delegates

You can also pass the `Action<NpgsqlEntityFrameworkCorePostgreSQLSettings>` delegate to set up some or all the options inline, for example to set the `ConnectionString`:

C#

```
builder.AddNpgsqlDbContext<YourDbContext>(  
  "db",  
  static settings => settings.ConnectionString = "YOUR_CONNECTIONSTRING");
```

Configure multiple DbContext classes

If you want to register more than one `DbContext` with different configuration, you can use `$"Aspire:Npgsql:EntityFrameworkCore:PostgreSQL:{typeof(TContext).Name}"` configuration section name. The json configuration would look like:

JSON

```
{  
  "Aspire": {  
    "Npgsql": {  
      "EntityFrameworkCore": {  
        "PostgreSQL": {  
          "ConnectionString": "YOUR_CONNECTIONSTRING",  
          "DbContextPooling": true,  
          "DisableHealthChecks": true,  
          "DisableTracing": true,  
          "AnotherDbContext": {  
            "ConnectionString": "AnotherDbContext_CONNECTIONSTRING",  
            "DisableTracing": false  
          }  
        }  
      }  
    }  
  }  
}
```

```
        }
    }
}
}
```

Then calling the `AddNpgsqlDbContext` method with `AnotherDbContext` type parameter would load the settings from

`Aspire:Npgsql:EntityFrameworkCore:PostgreSQL:AnotherDbContext` section.

C#

```
builder.AddNpgsqlDbContext<AnotherDbContext>();
```

Configuration options

Here are the configurable options with corresponding default values:

[] Expand table

Name	Description
<code>ConnectionString</code>	The connection string of the SQL Server database to connect to.
<code>MaxRetryCount</code>	The maximum number of retry attempts. Default value is 6, set it to 0 to disable the retry mechanism.
<code>DisableHealthChecks</code>	A boolean value that indicates whether the database health check is disabled or not.
<code>DisableTracing</code>	A boolean value that indicates whether the OpenTelemetry tracing is disabled or not.
<code>DisableMetrics</code>	A boolean value that indicates whether the OpenTelemetry metrics are disabled or not.

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

By default, the .NET Aspire PostgreSQL Entity Framework Core integrations handles the following:

- Adds the [DbContextHealthCheck](#), which calls EF Core's `CanConnectAsync` method. The name of the health check is the name of the `TContext` type.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire PostgreSQL Entity Framework Core integration uses the following Log categories:

- `Microsoft.EntityFrameworkCore.ChangeTracking`
- `Microsoft.EntityFrameworkCore.Database.Command`
- `Microsoft.EntityFrameworkCore.Database.Connection`
- `Microsoft.EntityFrameworkCore.Database.Transaction`
- `Microsoft.EntityFrameworkCore.Infrastructure`
- `Microsoft.EntityFrameworkCore.Infrastructure`
- `Microsoft.EntityFrameworkCore.Migrations`
- `Microsoft.EntityFrameworkCore.Model`
- `Microsoft.EntityFrameworkCore.Model.Validation`
- `Microsoft.EntityFrameworkCore.Query`
- `Microsoft.EntityFrameworkCore.Update`

Tracing

The .NET Aspire PostgreSQL Entity Framework Core integration will emit the following Tracing activities using OpenTelemetry:

- "Npgsql"

Metrics

The .NET Aspire PostgreSQL Entity Framework Core integration will emit the following metrics using OpenTelemetry:

- Microsoft.EntityFrameworkCore:
 - ec_Microsoft_EntityFrameworkCore_active_db_contexts
 - ec_Microsoft_EntityFrameworkCore_total_queries
 - ec_Microsoft_EntityFrameworkCore_queries_per_second
 - ec_Microsoft_EntityFrameworkCore_total_save_changes
 - ec_Microsoft_EntityFrameworkCore_save_changes_per_second
 - ec_Microsoft_EntityFrameworkCore_compiled_query_cache_hit_rate
 - ec_Microsoft_Entity_total_execution_strategy_operation_failures
 - ec_Microsoft_E_execution_strategy_operation_failures_per_second
 - ec_Microsoft_EntityF_total_optimistic_concurrency_failures
 - ec_Microsoft_EntityF_optimistic_concurrency_failures_per_second
- Npgsql:
 - ec_Npgsql_bytes_written_per_second
 - ec_Npgsql_bytes_read_per_second
 - ec_Npgsql_commands_per_second
 - ec_Npgsql_total_commands
 - ec_Npgsql_current_commands
 - ec_Npgsql_failed_commands
 - ec_Npgsql_prepared_commands_ratio
 - ec_Npgsql_connection_pools
 - ec_Npgsql_multiplexing_average_commands_per_batch
 - ec_Npgsql_multiplexing_average_write_time_per_batch

See also

- [Azure Database for PostgreSQL documentation](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire PostgreSQL integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire PostgreSQL integration. The `Aspire.Npgsql` library is used to register a [NpgsqlDataSource](#) in the DI container for connecting to a PostgreSQL database. It also enables corresponding health checks, logging and telemetry.

Get started

To get started with the .NET Aspire PostgreSQL integration, install the [Aspire.Npgsql](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the PostgreSQL client.

```
.NET CLI
dotnet add package Aspire.Npgsql
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the [AddNpgsqlDataSource](#) extension to register an `NpgsqlDataSource` for use via the dependency injection container.

```
C#
builder.AddNpgsqlDataSource("postgresdb");
```

After adding `NpgsqlDataSource` to the builder, you can get the `NpgsqlDataSource` instance using dependency injection. For example, to retrieve your context object from service:

```
C#
```

```
public class ExampleService(NpgsqlDataSource dataSource)
{
    // Use dataSource...
}
```

App host usage

To model the PostgreSQL server resource in the app host, install the [Aspire.Hosting.PostgreSQL](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.PostgreSQL
```

In your app host project, register and consume the PostgreSQL integration using the following methods, such as [AddPostgres](#):

C#

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

var postgres = builder.AddPostgres("postgres");
var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
    .WithReference(postgresdb);
```

When you want to explicitly provide the username and password, you can provide those as parameters. Consider the following alternative example:

C#

```
var username = builder.AddParameter("username", secret: true);
var password = builder.AddParameter("password", secret: true);

var postgres = builder.AddPostgres("postgres", username, password);

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
    .WithReference(postgresdb);
```

For more information, see [External parameters](#).

When adding PostgreSQL resources to the `builder` with the `AddPostgres` method, you can chain calls to `WithPgWeb` to add the [sosedoff/pgweb](#) container. This container is a cross-platform client for PostgreSQL databases, that serves a web-based admin dashboard. Consider the following example:

C#

```
var postgres = builder.AddPostgres("postgres")
    .WithPgWeb();

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
    .WithReference(postgresdb);
```

All registered `PostgresDatabaseResource` instances are used to create a configuration file per instance, and each config is bound to the `pgweb` container bookmark directory. For more information, see [PgWeb docs: Server connection bookmarks](#).

When adding PostgreSQL resources to the `builder` with the `AddPostgres` method, you can chain calls to `WithPgAdmin` to add the [dpage/pgadmin4](#) container. This container is a cross-platform client for PostgreSQL databases, that serves a web-based admin dashboard. Consider the following example:

C#

```
var postgres = builder.AddPostgres("postgres")
    .WithPgAdmin();

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
    .WithReference(postgresdb);
```

Azure app host usage

To deploy your PostgreSQL resources to Azure, you need to install the appropriate .NET Aspire hosting package:

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.PostgreSQL
```

After you've installed this package, you specify that your PostgreSQL resources will be hosted in Azure by calling the [PublishAsAzurePostgresFlexibleServer](#) extension method in your app host project:

C#

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

var postgres = builder.AddPostgres("postgres")
    .PublishAsAzurePostgresFlexibleServer();

var postgresdb = postgres.AddDatabase("postgresdb");

var exampleProject = builder.AddProject<Projects.ExampleProject>()
    .WithReference(postgresdb);
```

The preceding call to `PublishAsAzurePostgresFlexibleServer` configures Postgres Server resource to be deployed as Azure Postgres Flexible Server. For more information, see [Azure Postgres Flexible Server](#).

Configuration

The .NET Aspire PostgreSQL integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling [AddNpgsqlDataSource](#):

C#

```
builder.AddNpgsqlDataSource("NpgsqlConnection");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{  
  "ConnectionStrings": {  
    "NpgsqlConnection": "Host=myserver;Database=postgresdb"  
  }  
}
```

For more information, see the [ConnectionString](#).

Use configuration providers

The .NET Aspire PostgreSQL integration supports [Microsoft.Extensions.Configuration](#). It loads the [NpgsqlSettings](#) from *appsettings.json* or other configuration files by using the `Aspire:Npgsql` key. Example *appsettings.json* that configures some of the options:

The following example shows an *appsettings.json* file that configures some of the available options:

JSON

```
{  
  "Aspire": {  
    "Npgsql": {  
      "DisableHealthChecks": true,  
      "DisableTracing": true  
    }  
  }  
}
```

Use inline delegates

You can also pass the `Action<NpgsqlSettings> configureSettings` delegate to set up some or all the options inline, for example to disable health checks:

C#

```
builder.AddNpgsqlDataSource(  
  "postgresdb",  
  settings => settings.DisableHealthChecks = true);
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

- Adds the [NpgsqlHealthCheck](#), which verifies that commands can be successfully executed against the underlying Postgres Database.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire PostgreSQL integration uses the following Log categories:

- `Npgsql.Connection`
- `Npgsql.Command`
- `Npgsql.Transaction`
- `Npgsql.Copy`
- `Npgsql.Replication`
- `Npgsql.Exception`

Tracing

The .NET Aspire PostgreSQL integration will emit the following Tracing activities using OpenTelemetry:

- "Npgsql"

Metrics

The .NET Aspire PostgreSQL integration will emit the following metrics using OpenTelemetry:

- `Npgsql:`
 - `ec_Npgsql_bytes_written_per_second`

- `ec_Npgsql_bytes_read_per_second`
- `ec_Npgsql_commands_per_second`
- `ec_Npgsql_total_commands`
- `ec_Npgsql_current_commands`
- `ec_Npgsql_failed_commands`
- `ec_Npgsql_prepared_commands_ratio`
- `ec_Npgsql_connection_pools`
- `ec_Npgsql_multiplexing_average_commands_per_batch`
- `ec_Npgsql_multiplexing_average_write_time_per_batch`

See also

- [PostgreSQL docs ↗](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Qdrant integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Qdrant integration. Use this integration to register a [QdrantClient](#) in the DI container for connecting to a Qdrant server.

Get started

To get started with the .NET Aspire Qdrant integration, install the [Aspire.Qdrant.Client](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Qdrant client.

```
.NET CLI  
.NET CLI  
dotnet add package Aspire.Qdrant.Client
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the *Program.cs* file of your client-consuming project, call the `AddQdrantClient` extension method to register a `QdrantClient` for use via the dependency injection container. The method takes a connection name parameter.

```
C#  
builder.AddQdrantClient("qdrant");
```

To retrieve your `QdrantClient` object consider the following example service:

```
C#  
public class ExampleService(QdrantClient client)  
{
```

```
// Use client...
}
```

App host usage

To model the Qdrant server resource in the app host, install the [Aspire.Hosting.Qdrant](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Qdrant
```

In your app host project, register a Qdrant server and consume the connection using the following methods:

C#

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

var qdrant = builder.AddQdrant("qdrant");

var myService = builder.AddProject<Projects.MyService>()
    .WithReference(qdrant);
```

When you want to explicitly provide the API key, you can provide it as a parameter. Consider the following alternative example:

C#

```
var apiKey = builder.AddParameter("apikey", secret: true);

var qdrant = builder.AddQdrant("qdrant", apiKey);

var myService = builder.AddProject<Projects.MyService>()
    .WithReference(qdrant);
```

For more information, see [External parameters](#).

Configuration

The .NET Aspire Qdrant Client integration provides multiple options to configure the server connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddQdrantClient()`:

C#

```
builder.AddQdrantClient("qdrant");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "qdrant": "Endpoint=http://localhost:6334;Key=123456!@#$%"
  }
}
```

By default the `QdrantClient` uses the gRPC API endpoint.

Use configuration providers

The .NET Aspire Qdrant Client integration supports [Microsoft.Extensions.Configuration](#). It loads the `QdrantClientSettings` from configuration by using the `Aspire:Qdrant:Client` key. Example `appsettings.json` that configures some of the options:

JSON

```
{
  "Aspire": {
    "Qdrant": {
      "Client": {
        "Key": "123456!@#$%"
      }
    }
  }
}
```

Use inline delegates

You can also pass the `Action<QdrantClientSettings> configureSettings` delegate to set up some or all the options inline, for example to set the API key from code:

C#

```
builder.AddQdrantClient("qdrant", settings => settings.ApiKey =
"12345!@#$%");
```

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Qdrant integration uses the following logging categories:

- "Qdrant.Client"

See also

- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire RabbitMQ integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire RabbitMQ client message-broker. The `Aspire.RabbitMQ.Client` library is used to register an [IConnection](#) in the dependency injection (DI) container for connecting to a RabbitMQ server. It enables corresponding health check, logging and telemetry.

Get started

To get started with the .NET Aspire RabbitMQ integration, install the [Aspire.RabbitMQ.Client](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the RabbitMQ client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.RabbitMQ.Client
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the [AddRabbitMQClient](#) extension method to register an `IConnection` for use via the dependency injection container. The method takes a connection name parameter.

C#

```
builder.AddRabbitMQClient("messaging");
```

You can then retrieve the `IConnection` instance using dependency injection. For example, to retrieve the connection from an example service:

C#

```
public class ExampleService(IConnection connection)
{
```

```
// Use connection...
}
```

App host usage

To model the RabbitMQ resource in the app host, install the [Aspire.Hosting.RabbitMQ](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.RabbitMQ
```

In your app host project, register a RabbitMQ server and consume the connection using the following methods, such as [AddRabbitMQ](#):

C#

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

var messaging = builder.AddRabbitMQ("messaging");

builder.AddProject<Projects.ExampleProject>()
    .WithReference(messaging);
```

The [WithReference](#) method configures a connection in the `ExampleProject` project named `messaging`.

When you want to explicitly provide the username and password, you can provide those as parameters. Consider the following alternative example:

C#

```
var username = builder.AddParameter("username", secret: true);
var password = builder.AddParameter("password", secret: true);

var messaging = builder.AddRabbitMQ("messaging", username, password);

// Service consumption
builder.AddProject<Projects.ExampleProject>()
    .WithReference(messaging);
```

For more information, see [External parameters](#).

Configuration

The .NET Aspire RabbitMQ integration provides multiple options to configure the connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddRabbitMQClient`:

C#

```
builder.AddRabbitMQClient("RabbitMQConnection");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "RabbitMQConnection": "amqp://username:password@localhost:5672"
  }
}
```

For more information on how to format this connection string, see the [RabbitMQ URI specification docs](#).

Use configuration providers

The .NET Aspire RabbitMQ integration supports [Microsoft.Extensions.Configuration](#). It loads the `RabbitMQClientSettings` from configuration by using the `Aspire:RabbitMQ:Client` key. Example `appsettings.json` that configures some of the options:

JSON

```
{
  "Aspire": {
    "RabbitMQ": {
      "Client": {
        "DisableHealthChecks": true
      }
    }
  }
}
```

```
    }  
}
```

Use inline delegates

Also you can pass the `Action<RabbitMQClientSettings> configureSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddRabbitMQClient(  
    "messaging",  
    static settings => settings.DisableHealthChecks = true);
```

You can also set up the [IConnectionFactory](#) using the `Action<IConnectionFactory> configureConnectionFactory` delegate parameter of the `AddRabbitMQClient` method. For example to set the client provided name for connections:

C#

```
builder.AddRabbitMQClient(  
    "messaging",  
    static configureConnectionFactory:  
        factory => factory.ClientProvidedName = "MyApp");
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire RabbitMQ integration handles the following:

- Adds the health check when `RabbitMQClientSettings.DisableHealthChecks` is `true`, which attempts to connect to and create a channel on the RabbitMQ server.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations](#)

[overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire RabbitMQ integration uses the following log categories:

- `RabbitMQ.Client`

Tracing

The .NET Aspire RabbitMQ integration will emit the following tracing activities using OpenTelemetry:

- "Aspire.RabbitMQ.Client"

Metrics

The .NET Aspire RabbitMQ integration currently doesn't support metrics by default. If that changes in the future, this section will be updated to reflect those changes.

See also

- [RabbitMQ .NET Client docs ↗](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

Stack Exchange Redis caching overview

Article • 08/29/2024

With .NET Aspire, there are several ways to use caching in your applications. One popular option is to use [Stack Exchange Redis](#), which is a high-performance data store that can be used to store frequently accessed data. This article provides an overview of Stack Exchange Redis caching and links to resources that help you use it in your applications.

To use multiple Redis caching integrations in your application, see [Tutorial: Implement caching with .NET Aspire integrations](#). If you're interested in using the Redis Cache for Azure, see [Tutorial: Deploy a .NET Aspire project with a Redis Cache to Azure](#).

Redis serialization protocol (RESP)

The Redis serialization protocol (RESP) is a binary-safe protocol that Redis uses to communicate with clients. RESP is a simple, text-based protocol that is easy to implement and efficient to parse. RESP is used to send commands to Redis and receive responses from Redis. RESP is designed to be fast and efficient, making it well-suited for use in high-performance applications. For more information, see [Redis serialization protocol specification](#).

In addition to Redis itself, there are two well-maintained implementations of RESP for .NET:

- [Garnet](#): Garnet is a remote cache-store from Microsoft Research that offers strong performance (throughput and latency), scalability, storage, recovery, cluster sharding, key migration, and replication features. Garnet can work with existing Redis clients.
- [Valkey](#): A flexible distributed key-value datastore that supports both caching and beyond caching workloads.

.NET Aspire lets you easily model either the Redis, Garnet, or Valkey RESP protocol in your applications and you can choose which one to use based on your requirements. All of the .NET Aspire Redis integrations can be used with either the Redis, Garnet, or Valkey RESP protocol.

Caching

Caching is a technique used to store frequently accessed data in memory. This helps to reduce the time it takes to retrieve the data from the original source, such as a database or a web service. Caching can significantly improve the performance of an application by reducing the number of requests made to the original source. To access the Redis `IConnectionMultiplexer` object, you use the `Aspire.StackExchange.Redis` NuGet package:

[.NET Aspire Stack Exchange Redis integration](#)

[.NET Aspire Stack Exchange Redis integration \(Garnet\)](#)

[.NET Aspire Stack Exchange Redis integration \(Valkey\)](#)

Distributed caching

Distributed caching is a type of caching that stores data across multiple servers. This allows the data to be shared between multiple instances of an application, which can help to improve scalability and performance. Distributed caching can be used to store a wide variety of data, such as session state, user profiles, and frequently accessed data. To use Redis distributed caching in your application (the `IDistributedCache` interface), use the `Aspire.StackExchange.Redis.DistributedCaching` NuGet package:

[.NET Aspire Stack Exchange Redis distributed caching integration](#)

[.NET Aspire Stack Exchange Redis distributed caching integration \(Garnet\)](#)

[.NET Aspire Stack Exchange Redis distributed caching integration \(Valkey\)](#)

Output caching

Output caching is a type of caching that stores the output of a web page or API response. This allows the response to be served directly from the cache, rather than generating it from scratch each time. Output caching can help to improve the performance of a web application by reducing the time it takes to generate a response. To use declarative Redis output caching with either the `OutputCache` attribute or the `CacheOutput` method in your application, use the `Aspire.StackExchange.Redis.OutputCaching` NuGet package:

[.NET Aspire Stack Exchange Redis output caching integration](#)

.NET Aspire Stack Exchange Redis output caching integration (Garnet)

.NET Aspire Stack Exchange Redis output caching integration (Valkey)

See also

- [Caching in .NET](#)
- [Overview of Caching in ASP.NET Core](#)
- [Distributed caching in .NET](#)
- [Distributed caching in ASP.NET Core](#)
- [Output caching middleware in ASP.NET Core](#)

.NET Aspire Stack Exchange Redis integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Stack Exchange Redis integration. The `Aspire.StackExchange.Redis` library is used to register an [IConnectionMultiplexer](#) in the DI container for connecting to a [Redis](#) server. It enables corresponding health checks, logging and telemetry.

Get started

To get started with the .NET Aspire Stack Exchange Redis integration, install the [Aspire.StackExchange.Redis](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Stack Exchange Redis client.

```
.NET CLI
.NET CLI
dotnet add package Aspire.StackExchange.Redis
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the [AddRedisClient](#) extension to register a `IConnectionMultiplexer` for use via the dependency injection container.

```
C#
builder.AddRedisClient("cache");
```

You can then retrieve the `IConnectionMultiplexer` instance using dependency injection. For example, to retrieve the connection multiplexer from a service:

```
C#
```

```
public class ExampleService(IConnectionMultiplexer connectionMultiplexer)
{
    // Use connection multiplexer...
}
```

App host usage

To model the Redis resource (`RedisResource`) in the app host, install the [Aspire.Hosting.Redis](#) NuGet package in the `app host` project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Redis
```

In your app host project, register the .NET Aspire Stack Exchange Redis as a resource using the [AddRedis](#) method and consume the service using the following methods:

C#

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

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

builder.AddProject<Projects.ExampleProject>()
    .WithReference(cache)
```

The [WithReference](#) method configures a connection in the `ExampleProject` project named `cache`. In the `Program.cs` file of `ExampleProject`, the Redis connection can be consumed using:

C#

```
builder.AddRedis("cache");
```

Configuration

The .NET Aspire Stack Exchange Redis integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddRedis`:

C#

```
builder.AddRedis("cache");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the [Stack Exchange Redis configuration docs ↗](#).

Use configuration providers

The .NET Aspire Stack Exchange Redis integration supports [Microsoft.Extensions.Configuration](#). It loads the `StackExchangeRedisSettings` from configuration by using the `Aspire:StackExchange:Redis` key. Example `appsettings.json` that configures some of the options:

JSON

```
{
  "Aspire": {
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        },
        "DisableHealthChecks": true,
        "DisableTracing": false
      }
    }
  }
}
```

```
    }  
}
```

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings>` delegate to set up some or all the options inline, for example to configure `DisableTracing`:

```
C#
```

```
builder.AddRedis(  
    "cache",  
    settings => settings.DisableTracing = true);
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Stack Exchange Redis integration handles the following:

- Adds the `StackExchange.Redis` health check, tries to open the connection and throws when it fails.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Stack Exchange Redis integration uses the following log categories:

- `Aspire.StackExchange.Redis`

Tracing

The .NET Aspire Stack Exchange Redis integration will emit the following tracing activities using OpenTelemetry:

- "OpenTelemetry.Instrumentation.StackExchangeRedis"

Metrics

The .NET Aspire Stack Exchange Redis integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- [Stack Exchange Redis docs ↗](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Stack Exchange Redis distributed caching integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Stack Exchange Redis distributed caching integration. The `Aspire.StackExchange.Redis.DistributedCaching` library is used to register an [IDistributedCache](#) provider for connecting to [Redis](#) server. It enables corresponding health checks, logging and telemetry.

Get started

To get started with the .NET Aspire Stack Exchange Redis distributed caching integration, install the [Aspire.StackExchange.Redis.DistributedCaching](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Stack Exchange Redis distributed caching client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.DistributedCaching
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the `AddRedisDistributedCache` extension to register the required services for distributed caching and add a `IDistributedCache` for use via the dependency injection container.

C#

```
builder.AddRedisDistributedCache("cache");
```

You can then retrieve the `IDistributedCache` instance using dependency injection. For example, to retrieve the cache from a service:

C#

```
public class ExampleService(IDistributedCache cache)
{
    // Use cache...
}
```

App host usage

To model the Redis resource (`RedisResource`) in the app host, install the [Aspire.Hosting.Redis](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Redis
```

In your app host project, register the .NET Aspire Stack Exchange Redis as a resource using the [AddRedis](#) method and consume the service using the following methods:

C#

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

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

builder.AddProject<Projects.ExampleProject>()
    .WithReference(cache)
```

The [WithReference](#) method configures a connection in the `ExampleProject` project named `cache`. In the `Program.cs` file of `ExampleProject`, the Redis connection can be consumed using:

C#

```
builder.AddRedis("cache");
```

Configuration

The .NET Aspire Stack Exchange Redis distributed caching integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

```
builder.AddRedisDistributedCache:
```

C#

```
builder.AddRedisDistributedCache("RedisConnection");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "RedisConnection": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the [Stack Exchange Redis configuration docs](#).

Use configuration providers

The .NET Aspire Stack Exchange Redis distributed caching integration supports [Microsoft.Extensions.Configuration](#). It loads the `StackExchangeRedisSettings` from configuration by using the `Aspire:StackExchange:Redis` key. Example `appsettings.json` that configures some of the options:

JSON

```
{
  "Aspire": {
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        }
      }
    }
  }
}
```

```
        },
        "DisableHealthChecks": true,
        "DisableTracing": false
    }
}
}
}
```

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings>` delegate to set up some or all the options inline, for example to configure `DisableTracing`:

C#

```
builder.AddRedisDistributedCache(
    "cache",
    settings => settings.DisableTracing = true);
```

You can also set up the [ConfigurationOptions](#) using the `Action<ConfigurationOptions> configureOptions` delegate parameter of the `AddRedisDistributedCache` method. For example to set the connection timeout:

C#

```
builder.AddRedisDistributedCache(
    "cache",
    configureOptions: options => options.ConnectTimeout = 3000);
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Stack Exchange Redis distributed caching integration handles the following:

- Adds the `StackExchange.Redis` health check, tries to open the connection and throws when it fails.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Stack Exchange Redis Distributed Caching integration uses the following Log categories:

- `Aspire.StackExchange.Redis`
- `Microsoft.Extensions.Caching.StackExchangeRedis`

Tracing

The .NET Aspire Stack Exchange Redis Distributed Caching integration will emit the following Tracing activities using OpenTelemetry:

- "OpenTelemetry.Instrumentation.StackExchangeRedis"

Metrics

The .NET Aspire Stack Exchange Redis Distributed Caching integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- [Stack Exchange Redis docs](#) ↗
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo](#) ↗

.NET Aspire Stack Exchange Redis output caching integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Stack Exchange Redis output caching integration. The `Aspire.StackExchange.Redis.OutputCaching` library is used to register an [ASP.NET Core Output Caching](#) provider backed by a [Redis](#) server. It enables corresponding health check, logging, and telemetry..

Get started

To get started with the .NET Aspire Stack Exchange Redis output caching integration, install the [Aspire.StackExchange.Redis.OutputCaching](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Stack Exchange Redis output caching client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.OutputCaching
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the [AddRedisOutputCache](#) extension to register the required services for output caching.

C#

```
builder.AddRedisOutputCache("cache");
```

Add the middleware to the request processing pipeline by calling [UseOutputCache](#).

C#

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

For minimal API apps, configure an endpoint to do caching by calling [CacheOutput](#), or by applying the [OutputCacheAttribute](#), as shown in the following examples:

C#

```
app.MapGet("/cached", () => { return "Hello world!"; }).CacheOutput();
app.MapGet("/attribute", [OutputCache] () => { return "Hello world!"; });
```

For apps with controllers, apply the `[OutputCache]` attribute to the action method. For Razor Pages apps, apply the attribute to the Razor page class.

App host usage

To model the Redis resource (`RedisResource`) in the app host, install the [Aspire.Hosting.Redis](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Redis
```

In your app host project, register the .NET Aspire Stack Exchange Redis as a resource using the [AddRedis](#) method and consume the service using the following methods:

C#

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

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

builder.AddProject<Projects.ExampleProject>()
    .WithReference(cache)
```

The [WithReference](#) method configures a connection in the `ExampleProject` project named `cache`. In the `Program.cs` file of `ExampleProject`, the Redis connection can be consumed using:

C#

```
builder.AddRedis("cache");
```

Configuration

The .NET Aspire Stack Exchange Redis output caching integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

```
builder.AddRedisOutputCache:
```

C#

```
builder.AddRedisOutputCache("RedisConnection");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "RedisConnection": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the [Stack Exchange Redis configuration docs ↗](#).

Use configuration providers

The .NET Aspire Stack Exchange Redis output caching integration supports [Microsoft.Extensions.Configuration](#). It loads the `StackExchangeRedisSettings` from configuration by using the `Aspire:StackExchange:Redis` key. Example `appsettings.json` that configures some of the options:

JSON

```
{  
    "Aspire": {  
        "StackExchange": {  
            "Redis": {  
                "ConfigurationOptions": {  
                    "ConnectTimeout": 3000,  
                    "ConnectRetry": 2  
                },  
                "DisableHealthChecks": true,  
                "DisableTracing": false  
            }  
        }  
    }  
}
```

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings> configurationSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddRedisOutputCache(  
    "cache",  
    static settings => settings.DisableHealthChecks = true);
```

You can also set up the `ConfigurationOptions` using the `Action<ConfigurationOptions> configureOptions` delegate parameter of the `AddRedisOutputCache` method. For example to set the connection timeout:

C#

```
builder.AddRedisOutputCache(  
    "cache",  
    static configureOptions: options => options.ConnectTimeout = 3000);
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Stack Exchange Redis output caching integration handles the following:

- Adds the `StackExchange.Redis` health check, tries to open the connection and throws when it fails.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Stack Exchange Redis output caching integration uses the following Log categories:

- `Aspire.StackExchange.Redis`
- `Microsoft.AspNetCore.OutputCaching.StackExchangeRedis`

Tracing

The .NET Aspire Stack Exchange Redis output caching integration will emit the following Tracing activities using OpenTelemetry:

- "OpenTelemetry.Instrumentation.StackExchangeRedis"

Metrics

The .NET Aspire Stack Exchange Redis output caching integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- [Stack Exchange Redis docs ↗](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Stack Exchange Redis integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Stack Exchange Redis integration. The `Aspire.StackExchange.Redis` library is used to register an [IConnectionMultiplexer](#) in the DI container for connecting to a [Redis](#) server. It enables corresponding health checks, logging and telemetry.

Get started

To get started with the .NET Aspire Stack Exchange Redis integration, install the [Aspire.StackExchange.Redis](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Stack Exchange Redis client.

```
.NET CLI  
.NET CLI  
dotnet add package Aspire.StackExchange.Redis
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the [AddRedisClient](#) extension to register a `IConnectionMultiplexer` for use via the dependency injection container.

```
C#  
builder.AddRedisClient("cache");
```

You can then retrieve the `IConnectionMultiplexer` instance using dependency injection. For example, to retrieve the connection multiplexer from a service:

```
C#
```

```
public class ExampleService(IConnectionMultiplexer connectionMultiplexer)
{
    // Use connection multiplexer...
}
```

App host usage

To model the Garnet resource (`GarnetResource`) in the app host, install the [Aspire.Hosting.Garnet](#) NuGet package in the `app host` project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Garnet
```

In your app host project, register .NET Aspire Garnet as a `GarnetResource` using the `AddGarnet` method and consume the service using the following methods:

C#

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

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

builder.AddProject<Projects.ExampleProject>()
    .WithReference(cache)
```

The `WithReference` method configures a connection in the `ExampleProject` project named `cache`. In the `Program.cs` file of `ExampleProject`, the Garnet connection can be consumed using:

C#

```
builder.AddGarnet("cache");
```

Configuration

The .NET Aspire Stack Exchange Redis integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddGarnet`:

```
C#
```

```
builder.AddGarnet("cache");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

```
JSON
```

```
{
  "ConnectionStrings": {
    "cache": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the [Stack Exchange Redis configuration docs ↗](#).

Use configuration providers

The .NET Aspire Stack Exchange Redis integration supports [Microsoft.Extensions.Configuration](#). It loads the `StackExchangeRedisSettings` from configuration by using the `Aspire:StackExchange:Redis` key. Example `appsettings.json` that configures some of the options:

```
JSON
```

```
{
  "Aspire": {
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        },
        "DisableHealthChecks": true,
        "DisableTracing": false
      }
    }
  }
}
```

```
    }  
}
```

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings>` delegate to set up some or all the options inline, for example to configure `DisableTracing`:

```
C#
```

```
builder.AddGarnet(  
    "cache",  
    settings => settings.DisableTracing = true);
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Stack Exchange Redis integration handles the following:

- Adds the `StackExchange.Redis` health check, tries to open the connection and throws when it fails.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Stack Exchange Redis integration uses the following log categories:

- `Aspire.StackExchange.Redis`

Tracing

The .NET Aspire Stack Exchange Redis integration will emit the following tracing activities using OpenTelemetry:

- "OpenTelemetry.Instrumentation.StackExchangeRedis"

Metrics

The .NET Aspire Stack Exchange Redis integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- [Stack Exchange Redis docs ↗](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Stack Exchange Redis distributed caching integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Stack Exchange Redis distributed caching integration. The `Aspire.StackExchange.Redis.DistributedCaching` library is used to register an [IDistributedCache](#) provider for connecting to [Redis](#) server. It enables corresponding health checks, logging and telemetry.

Get started

To get started with the .NET Aspire Stack Exchange Redis distributed caching integration, install the [Aspire.StackExchange.Redis.DistributedCaching](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Stack Exchange Redis distributed caching client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.DistributedCaching
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the `AddRedisDistributedCache` extension to register the required services for distributed caching and add a `IDistributedCache` for use via the dependency injection container.

C#

```
builder.AddRedisDistributedCache("cache");
```

You can then retrieve the `IDistributedCache` instance using dependency injection. For example, to retrieve the cache from a service:

C#

```
public class ExampleService(IDistributedCache cache)
{
    // Use cache...
}
```

App host usage

To model the Garnet resource (`GarnetResource`) in the app host, install the [Aspire.Hosting.Garnet](#) NuGet package in the `app host` project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Garnet
```

In your app host project, register .NET Aspire Garnet as a `GarnetResource` using the `AddGarnet` method and consume the service using the following methods:

C#

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

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

builder.AddProject<Projects.ExampleProject>()
    .WithReference(cache)
```

The `WithReference` method configures a connection in the `ExampleProject` project named `cache`. In the `Program.cs` file of `ExampleProject`, the Garnet connection can be consumed using:

C#

```
builder.AddGarnet("cache");
```

Configuration

The .NET Aspire Stack Exchange Redis distributed caching integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

```
builder.AddRedisDistributedCache:
```

C#

```
builder.AddRedisDistributedCache("RedisConnection");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "RedisConnection": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the [Stack Exchange Redis configuration docs](#).

Use configuration providers

The .NET Aspire Stack Exchange Redis distributed caching integration supports [Microsoft.Extensions.Configuration](#). It loads the `StackExchangeRedisSettings` from configuration by using the `Aspire:StackExchange:Redis` key. Example `appsettings.json` that configures some of the options:

JSON

```
{
  "Aspire": {
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        }
      }
    }
  }
}
```

```
        },
        "DisableHealthChecks": true,
        "DisableTracing": false
    }
}
}
}
```

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings>` delegate to set up some or all the options inline, for example to configure `DisableTracing`:

C#

```
builder.AddRedisDistributedCache(
    "cache",
    settings => settings.DisableTracing = true);
```

You can also set up the [ConfigurationOptions](#) using the `Action<ConfigurationOptions> configureOptions` delegate parameter of the `AddRedisDistributedCache` method. For example to set the connection timeout:

C#

```
builder.AddRedisDistributedCache(
    "cache",
    configureOptions: options => options.ConnectTimeout = 3000);
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Stack Exchange Redis distributed caching integration handles the following:

- Adds the `StackExchange.Redis` health check, tries to open the connection and throws when it fails.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Stack Exchange Redis Distributed Caching integration uses the following Log categories:

- `Aspire.StackExchange.Redis`
- `Microsoft.Extensions.Caching.StackExchangeRedis`

Tracing

The .NET Aspire Stack Exchange Redis Distributed Caching integration will emit the following Tracing activities using OpenTelemetry:

- "OpenTelemetry.Instrumentation.StackExchangeRedis"

Metrics

The .NET Aspire Stack Exchange Redis Distributed Caching integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- [Stack Exchange Redis docs](#) ↗
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo](#) ↗

.NET Aspire Stack Exchange Redis output caching integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Stack Exchange Redis output caching integration. The `Aspire.StackExchange.Redis.OutputCaching` library is used to register an [ASP.NET Core Output Caching](#) provider backed by a [Redis](#) server. It enables corresponding health check, logging, and telemetry..

Get started

To get started with the .NET Aspire Stack Exchange Redis output caching integration, install the [Aspire.StackExchange.Redis.OutputCaching](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Stack Exchange Redis output caching client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.OutputCaching
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the [AddRedisOutputCache](#) extension to register the required services for output caching.

C#

```
builder.AddRedisOutputCache("cache");
```

Add the middleware to the request processing pipeline by calling [UseOutputCache](#).

C#

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

For minimal API apps, configure an endpoint to do caching by calling [CacheOutput](#), or by applying the [OutputCacheAttribute](#), as shown in the following examples:

C#

```
app.MapGet("/cached", () => { return "Hello world!"; }).CacheOutput();
app.MapGet("/attribute", [OutputCache] () => { return "Hello world!"; });
```

For apps with controllers, apply the `[OutputCache]` attribute to the action method. For Razor Pages apps, apply the attribute to the Razor page class.

App host usage

To model the Garnet resource (`GarnetResource`) in the app host, install the [Aspire.Hosting.Garnet](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Garnet
```

In your app host project, register .NET Aspire Garnet as a `GarnetResource` using the `AddGarnet` method and consume the service using the following methods:

C#

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

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

builder.AddProject<Projects.ExampleProject>()
    .WithReference(cache)
```

The `WithReference` method configures a connection in the `ExampleProject` project named `cache`. In the `Program.cs` file of `ExampleProject`, the Garnet connection can be consumed using:

C#

```
builder.AddGarnet("cache");
```

Configuration

The .NET Aspire Stack Exchange Redis output caching integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

```
builder.AddRedisOutputCache:
```

C#

```
builder.AddRedisOutputCache("RedisConnection");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "RedisConnection": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the [Stack Exchange Redis configuration docs ↗](#).

Use configuration providers

The .NET Aspire Stack Exchange Redis output caching integration supports [Microsoft.Extensions.Configuration](#). It loads the `StackExchangeRedisSettings` from configuration by using the `Aspire:StackExchange:Redis` key. Example `appsettings.json` that configures some of the options:

JSON

```
{  
    "Aspire": {  
        "StackExchange": {  
            "Redis": {  
                "ConfigurationOptions": {  
                    "ConnectTimeout": 3000,  
                    "ConnectRetry": 2  
                },  
                "DisableHealthChecks": true,  
                "DisableTracing": false  
            }  
        }  
    }  
}
```

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings> configurationSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddRedisOutputCache(  
    "cache",  
    static settings => settings.DisableHealthChecks = true);
```

You can also set up the `ConfigurationOptions` using the `Action<ConfigurationOptions> configureOptions` delegate parameter of the `AddRedisOutputCache` method. For example to set the connection timeout:

C#

```
builder.AddRedisOutputCache(  
    "cache",  
    static configureOptions: options => options.ConnectTimeout = 3000);
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Stack Exchange Redis output caching integration handles the following:

- Adds the `StackExchange.Redis` health check, tries to open the connection and throws when it fails.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Stack Exchange Redis output caching integration uses the following Log categories:

- `Aspire.StackExchange.Redis`
- `Microsoft.AspNetCore.OutputCaching.StackExchangeRedis`

Tracing

The .NET Aspire Stack Exchange Redis output caching integration will emit the following Tracing activities using OpenTelemetry:

- "OpenTelemetry.Instrumentation.StackExchangeRedis"

Metrics

The .NET Aspire Stack Exchange Redis output caching integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- [Stack Exchange Redis docs ↗](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Stack Exchange Redis integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Stack Exchange Redis integration. The `Aspire.StackExchange.Redis` library is used to register an [IConnectionMultiplexer](#) in the DI container for connecting to a [Redis](#) server. It enables corresponding health checks, logging and telemetry.

Get started

To get started with the .NET Aspire Stack Exchange Redis integration, install the [Aspire.StackExchange.Redis](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Stack Exchange Redis client.

```
.NET CLI
.NET CLI
dotnet add package Aspire.StackExchange.Redis
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the [AddRedisClient](#) extension to register a `IConnectionMultiplexer` for use via the dependency injection container.

```
C#
builder.AddRedisClient("cache");
```

You can then retrieve the `IConnectionMultiplexer` instance using dependency injection. For example, to retrieve the connection multiplexer from a service:

```
C#
```

```
public class ExampleService(IConnectionMultiplexer connectionMultiplexer)
{
    // Use connection multiplexer...
}
```

App host usage

To model the Valkey resource (`ValkeyResource`) in the app host, install the [Aspire.Hosting.Valkey](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Valkey
```

In your app host project, register the .NET Aspire Valkey as a resource using the `AddValkey` method and consume the service using the following methods:

C#

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

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

builder.AddProject<Projects.ExampleProject>()
    .WithReference(cache)
```

The `WithReference` method configures a connection in the `ExampleProject` project named `cache`. In the `Program.cs` file of `ExampleProject`, the Valkey connection can be consumed using:

C#

```
builder.AddValkey("cache");
```

Configuration

The .NET Aspire Stack Exchange Redis integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

To model the Valkey resource (`ValkeyResource`) in the app host, install the [Aspire.Hosting.Valkey](#) NuGet package in the `app host` project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Valkey
```

In your app host project, register the .NET Aspire Valkey as a resource using the `AddValkey` method and consume the service using the following methods:

C#

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

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

builder.AddProject<Projects.ExampleProject>()
    .WithReference(cache)
```

The `WithReference` method configures a connection in the `ExampleProject` project named `cache`. In the `Program.cs` file of `ExampleProject`, the Valkey connection can be consumed using:

C#

```
builder.AddValkey("cache");
```

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling `builder.AddValkey`:

C#

```
builder.AddValkey("cache");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{  
  "ConnectionStrings": {  
    "cache": "localhost:6379"  
  }  
}
```

For more information on how to format this connection string, see the [Stack Exchange Redis configuration docs](#).

Use configuration providers

The .NET Aspire Stack Exchange Redis integration supports [Microsoft.Extensions.Configuration](#). It loads the `StackExchangeRedisSettings` from configuration by using the `Aspire:StackExchange:Redis` key. Example `appsettings.json` that configures some of the options:

JSON

```
{  
  "Aspire": {  
    "StackExchange": {  
      "Redis": {  
        "ConfigurationOptions": {  
          "ConnectTimeout": 3000,  
          "ConnectRetry": 2  
        },  
        "DisableHealthChecks": true,  
        "DisableTracing": false  
      }  
    }  
  }  
}
```

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings>` delegate to set up some or all the options inline, for example to configure `DisableTracing`:

To model the Valkey resource (`ValkeyResource`) in the app host, install the [Aspire.Hosting.Valkey](#) NuGet package in the `app host` project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Valkey
```

In your app host project, register the .NET Aspire Valkey as a resource using the `AddValkey` method and consume the service using the following methods:

C#

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

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

builder.AddProject<Projects.ExampleProject>()
    .WithReference(cache)
```

The `WithReference` method configures a connection in the `ExampleProject` project named `cache`. In the `Program.cs` file of `ExampleProject`, the Valkey connection can be consumed using:

C#

```
builder.AddValkey("cache");
```

C#

```
builder.AddValkey(
    "cache",
    settings => settings.DisableTracing = true);
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Stack Exchange Redis integration handles the following:

- Adds the `StackExchange.Redis` health check, tries to open the connection and throws when it fails.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Stack Exchange Redis integration uses the following log categories:

- `Aspire.StackExchange.Redis`

Tracing

The .NET Aspire Stack Exchange Redis integration will emit the following tracing activities using OpenTelemetry:

- "OpenTelemetry.Instrumentation.StackExchangeRedis"

Metrics

The .NET Aspire Stack Exchange Redis integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- [Stack Exchange Redis docs ↗](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Stack Exchange Redis distributed caching integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Stack Exchange Redis distributed caching integration. The `Aspire.StackExchange.Redis.DistributedCaching` library is used to register an [IDistributedCache](#) provider for connecting to [Redis](#) server. It enables corresponding health checks, logging and telemetry.

Get started

To get started with the .NET Aspire Stack Exchange Redis distributed caching integration, install the [Aspire.StackExchange.Redis.DistributedCaching](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Stack Exchange Redis distributed caching client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.DistributedCaching
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the `AddRedisDistributedCache` extension to register the required services for distributed caching and add a `IDistributedCache` for use via the dependency injection container.

C#

```
builder.AddRedisDistributedCache("cache");
```

You can then retrieve the `IDistributedCache` instance using dependency injection. For example, to retrieve the cache from a service:

C#

```
public class ExampleService(IDistributedCache cache)
{
    // Use cache...
}
```

App host usage

To model the Valkey resource (`ValkeyResource`) in the app host, install the [Aspire.Hosting.Valkey](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Valkey
```

In your app host project, register the .NET Aspire Valkey as a resource using the `AddValkey` method and consume the service using the following methods:

C#

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

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

builder.AddProject<Projects.ExampleProject>()
    .WithReference(cache)
```

The `WithReference` method configures a connection in the `ExampleProject` project named `cache`. In the `Program.cs` file of `ExampleProject`, the Valkey connection can be consumed using:

C#

```
builder.AddValkey("cache");
```

Configuration

The .NET Aspire Stack Exchange Redis distributed caching integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

```
builder.AddRedisDistributedCache:
```

C#

```
builder.AddRedisDistributedCache("RedisConnection");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "RedisConnection": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the [Stack Exchange Redis configuration docs](#).

Use configuration providers

The .NET Aspire Stack Exchange Redis distributed caching integration supports [Microsoft.Extensions.Configuration](#). It loads the `StackExchangeRedisSettings` from configuration by using the `Aspire:StackExchange:Redis` key. Example `appsettings.json` that configures some of the options:

JSON

```
{
  "Aspire": {
    "StackExchange": {
      "Redis": {
        "ConfigurationOptions": {
          "ConnectTimeout": 3000,
          "ConnectRetry": 2
        }
      }
    }
  }
}
```

```
        },
        "DisableHealthChecks": true,
        "DisableTracing": false
    }
}
}
}
```

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings>` delegate to set up some or all the options inline, for example to configure `DisableTracing`:

C#

```
builder.AddRedisDistributedCache(
    "cache",
    settings => settings.DisableTracing = true);
```

You can also set up the [ConfigurationOptions](#) using the `Action<ConfigurationOptions> configureOptions` delegate parameter of the `AddRedisDistributedCache` method. For example to set the connection timeout:

C#

```
builder.AddRedisDistributedCache(
    "cache",
    configureOptions: options => options.ConnectTimeout = 3000);
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Stack Exchange Redis distributed caching integration handles the following:

- Adds the `StackExchange.Redis` health check, tries to open the connection and throws when it fails.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Stack Exchange Redis Distributed Caching integration uses the following Log categories:

- `Aspire.StackExchange.Redis`
- `Microsoft.Extensions.Caching.StackExchangeRedis`

Tracing

The .NET Aspire Stack Exchange Redis Distributed Caching integration will emit the following Tracing activities using OpenTelemetry:

- "OpenTelemetry.Instrumentation.StackExchangeRedis"

Metrics

The .NET Aspire Stack Exchange Redis Distributed Caching integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- [Stack Exchange Redis docs](#) ↗
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo](#) ↗

.NET Aspire Stack Exchange Redis output caching integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Stack Exchange Redis output caching integration. The `Aspire.StackExchange.Redis.OutputCaching` library is used to register an [ASP.NET Core Output Caching](#) provider backed by a [Redis](#) server. It enables corresponding health check, logging, and telemetry..

Get started

To get started with the .NET Aspire Stack Exchange Redis output caching integration, install the [Aspire.StackExchange.Redis.OutputCaching](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Stack Exchange Redis output caching client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.StackExchange.Redis.OutputCaching
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the [AddRedisOutputCache](#) extension to register the required services for output caching.

C#

```
builder.AddRedisOutputCache("cache");
```

Add the middleware to the request processing pipeline by calling [UseOutputCache](#).

C#

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

For minimal API apps, configure an endpoint to do caching by calling [CacheOutput](#), or by applying the [OutputCacheAttribute](#), as shown in the following examples:

C#

```
app.MapGet("/cached", () => { return "Hello world!"; }).CacheOutput();
app.MapGet("/attribute", [OutputCache] () => { return "Hello world!"; });
```

For apps with controllers, apply the `[OutputCache]` attribute to the action method. For Razor Pages apps, apply the attribute to the Razor page class.

App host usage

To model the Valkey resource (`ValkeyResource`) in the app host, install the [Aspire.Hosting.Valkey](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Valkey
```

In your app host project, register the .NET Aspire Valkey as a resource using the `AddValkey` method and consume the service using the following methods:

C#

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

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

builder.AddProject<Projects.ExampleProject>()
    .WithReference(cache)
```

The [WithReference](#) method configures a connection in the `ExampleProject` project named `cache`. In the `Program.cs` file of `ExampleProject`, the Valkey connection can be consumed using:

C#

```
builder.AddValkey("cache");
```

Configuration

The .NET Aspire Stack Exchange Redis output caching integration provides multiple options to configure the Redis connection based on the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you can provide the name of the connection string when calling

```
builder.AddRedisOutputCache:
```

C#

```
builder.AddRedisOutputCache("RedisConnection");
```

And then the connection string will be retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "RedisConnection": "localhost:6379"
  }
}
```

For more information on how to format this connection string, see the [Stack Exchange Redis configuration docs ↗](#).

Use configuration providers

The .NET Aspire Stack Exchange Redis output caching integration supports [Microsoft.Extensions.Configuration](#). It loads the `StackExchangeRedisSettings` from configuration by using the `Aspire:StackExchange:Redis` key. Example `appsettings.json` that configures some of the options:

JSON

```
{  
    "Aspire": {  
        "StackExchange": {  
            "Redis": {  
                "ConfigurationOptions": {  
                    "ConnectTimeout": 3000,  
                    "ConnectRetry": 2  
                },  
                "DisableHealthChecks": true,  
                "DisableTracing": false  
            }  
        }  
    }  
}
```

Use inline delegates

You can also pass the `Action<StackExchangeRedisSettings> configurationSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddRedisOutputCache(  
    "cache",  
    static settings => settings.DisableHealthChecks = true);
```

You can also set up the `ConfigurationOptions` using the `Action<ConfigurationOptions> configureOptions` delegate parameter of the `AddRedisOutputCache` method. For example to set the connection timeout:

C#

```
builder.AddRedisOutputCache(  
    "cache",  
    static configureOptions: options => options.ConnectTimeout = 3000);
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Stack Exchange Redis output caching integration handles the following:

- Adds the `StackExchange.Redis` health check, tries to open the connection and throws when it fails.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Stack Exchange Redis output caching integration uses the following Log categories:

- `Aspire.StackExchange.Redis`
- `Microsoft.AspNetCore.OutputCaching.StackExchangeRedis`

Tracing

The .NET Aspire Stack Exchange Redis output caching integration will emit the following Tracing activities using OpenTelemetry:

- "OpenTelemetry.Instrumentation.StackExchangeRedis"

Metrics

The .NET Aspire Stack Exchange Redis output caching integration currently doesn't support metrics by default due to limitations with the `StackExchange.Redis` library.

See also

- [Stack Exchange Redis docs ↗](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire Seq integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire Seq integration to add OpenTelemetry Protocol (OTLP) exporters that send logs and traces to a Seq Server. The integration supports persistent logs and traces across application restarts via configuration.

Get started

To get started with the .NET Aspire Seq integration, install the [Aspire.Seq](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the Seq client.

```
.NET CLI
.NET CLI
dotnet add package Aspire.Seq
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the *Program.cs* file of your projects, call the `AddSeqEndpoint` extension method to register OpenTelemetry Protocol exporters to send logs and traces to Seq and the .NET Aspire Dashboard. The method takes a connection name parameter.

```
C#
builder.AddSeqEndpoint("seq");
```

App host usage

To model the Seq resource in the app host, install the [Aspire.Hosting.Seq](#) NuGet package in the [app host](#) project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Seq
```

In your app host project, register a Seq database and consume the connection using the following methods:

C#

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

var seq = builder.AddSeq("seq")
    .ExcludeFromManifest();

var myService = builder.AddProject<Projects.MyService>()
    .WithReference(seq);
```

The preceding code registers a Seq server and propagates its configuration.

ⓘ Important

You must accept the [Seq End User Licence Agreement](#) for Seq to start):

In the *Program.cs* file of the **MyService** project, configure logging and tracing to Seq using the following code:

C#

```
builder.AddSeqEndpoint("seq");
```

Seq in the .NET Aspire manifest

Seq shouldn't be part of the .NET Aspire deployment manifest, hence the chained call to `ExcludeFromManifest`. It's recommended you set up a secure production Seq server outside of .NET Aspire.

Persistent logs and traces

Register Seq with a data directory in your AppHost project to retain Seq's data and configuration across application restarts.

C#

```
var seq = builder.AddSeq("seq", seqDataDirectory: "./seqdata")
    .ExcludeFromManifest();
```

The directory specified must already exist.

Configuration

The .NET Aspire Seq integration provides options to configure the connection to Seq.

Use configuration providers

The .NET Aspire Seq integration supports [Microsoft.Extensions.Configuration](#). It loads the `SeqSettings` from configuration by using the `Aspire:Seq` key. Example `appsettings.json` that configures some of the options:

JSON

```
{
  "Aspire": {
    "Seq": {
      "DisableHealthChecks": true,
      "ServerUrl": "http://localhost:5341"
    }
  }
}
```

Use inline delegates

You can pass the `Action<SeqSettings> configureSettings` delegate to set up some or all the options inline, for example to disable health checks from code:

C#

```
builder.AddSeqEndpoint("seq", static settings =>
{
  settings.DisableHealthChecks = true;
  settings.ServerUrl = "http://localhost:5341"
});
```

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

The .NET Aspire Seq integration handles the following:

- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic.

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire Seq integration uses the following log categories:

- `Seq`

See also

- [SEQ Query Language ↗](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire SqlServer Entity Framework Core integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire SqlServer Entity Framework Core integration. The `Aspire.Microsoft.EntityFrameworkCore.SqlServer` library is used to:

- Registers `EntityFrameworkCore DbContext` service for connecting to a SQL database.
- Automatically configures the following:
 - Connection pooling to efficiently manage HTTP requests and database connections
 - Health checks, logging and telemetry to improve app monitoring and diagnostics

Prerequisites

- SQL database and connection string for accessing the database.

Get started

To get started with the .NET Aspire SQL Server Entity Framework Core integration, install the [Aspire.Microsoft.EntityFrameworkCore.SqlServer](#) NuGet package in the client-consuming project, i.e., the project for the application that uses the SQL Server Entity Framework Core client.

```
.NET CLI
.NET CLI
dotnet add package Aspire.Microsoft.EntityFrameworkCore.SqlServer
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your integration-consuming project, call the `AddSqlServerDbContext` extension to register a `DbContext` for use via the dependency injection container.

C#

```
builder.AddSqlServerDbContext<YourDbContext>("sqlldb");
```

To retrieve `YourDbContext` object from a service:

C#

```
public class ExampleService(YourDbContext client)
{
    // Use client...
}
```

App host usage

To model the SqlServer resource in the app host, install the [Aspire.Hosting.SqlServer](#) NuGet package in the `app host` project.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.SqlServer
```

In your app host project, register a SqlServer database and consume the connection using the following methods:

C#

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

var sql = builder.AddSqlServer("sql");
var sqlDb = sql.AddDatabase("sqlDb");

var myService = builder.AddProject<Projects.MyService>()
    .WithReference(sqlDb);
```

Configuration

The .NET Aspire SQL Server Entity Framework Core integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use connection string

When using a connection string from the `ConnectionStrings` configuration section, you provide the name of the connection string when calling

```
builder.AddSqlServerDbContext<TContext>():
```

C#

```
builder.AddSqlServerDbContext<MyDbContext>("myConnection");
```

The connection string is retrieved from the `ConnectionStrings` configuration section:

JSON

```
{
  "ConnectionStrings": {
    "myConnection": "Data Source=myserver;Initial Catalog=master"
  }
}
```

The `EnrichSqlServerDbContext` won't make use of the `ConnectionStrings` configuration section since it expects a `DbContext` to be registered at the point it's called.

For more information, see the [ConnectionString](#).

Use configuration providers

The .NET Aspire SQL Server Entity Framework Core integration supports [Microsoft.Extensions.Configuration](#). It loads the [Microsoft.EntityFrameworkCore.SqlServerSettings](#) from configuration files such as `appsettings.json` by using the `Aspire:Microsoft:EntityFrameworkCore:SqlServer` key. If you have set up your configurations in the `Aspire:Microsoft:EntityFrameworkCore:SqlServer` section you can just call the method without passing any parameter.

The following is an example of an `appsettings.json` file that configures some of the available options:

JSON

```
{  
  "Aspire": {  
    "Microsoft": {  
      "EntityFrameworkCore": {  
        "SqlServer": {  
          "ConnectionString": "YOUR_CONNECTIONSTRING",  
          "DbContextPooling": true,  
          "DisableHealthChecks": true,  
          "DisableTracing": true,  
          "DisableMetrics": false  
        }  
      }  
    }  
  }  
}
```

Use inline configurations

You can also pass the `Action<Microsoft.EntityFrameworkCore.SqlServerSettings>` delegate to set up some or all the options inline, for example to turn off the metrics:

C#

```
builder.AddSqlServerDbContext<YourDbContext>(  
  "sql",  
  static settings =>  
    settings.DisableMetrics = true);
```

Configure multiple DbContext connections

If you want to register more than one `DbContext` with different configuration, you can use `$"Aspire.Microsoft.EntityFrameworkCore.SqlServer:{typeof(TContext).Name}"` configuration section name. The json configuration would look like:

JSON

```
{  
  "Aspire": {  
    "Microsoft": {  
      "EntityFrameworkCore": {  
        "SqlServer": {  
          "ConnectionString": "YOUR_CONNECTIONSTRING",  
          "DbContextPooling": true,  
          "DisableHealthChecks": true,  
          "DisableTracing": true,  
          "DisableMetrics": false,  
          "AnotherDbContext": {  
            "ConnectionString": "ANOTHER_CONNECTIONSTRING",  
            "DbContextPooling": true,  
            "DisableHealthChecks": true,  
            "DisableTracing": true,  
            "DisableMetrics": false  
          }  
        }  
      }  
    }  
  }  
}
```

```
        "ConnectionString": "AnotherDbContext_CONNECTIONSTRING",
        "DisableTracing": false
    }
}
}
}
}
}
```

Then calling the `AddSqlServerDbContext` method with `AnotherDbContext` type parameter would load the settings from

Aspire:Microsoft.EntityFrameworkCore.SqlServer.AnotherDbContext section.

C#

```
builder.AddSqlServerDbContext<AnotherDbContext>("another-sql");
```

Configuration options

Here are the configurable options with corresponding default values:

 Expand table

Name	Description
ConnectionString	The connection string of the SQL Server database to connect to.
DbContextPooling	A boolean value that indicates whether the db context will be pooled or explicitly created every time it's requested
MaxRetryCount	The maximum number of retry attempts. Default value is 6, set it to 0 to disable the retry mechanism.
DisableHealthChecks	A boolean value that indicates whether the database health check is disabled or not.
DisableTracing	A boolean value that indicates whether the OpenTelemetry tracing is disabled or not.
DisableMetrics	A boolean value that indicates whether the OpenTelemetry metrics are disabled or not.
Timeout	The time in seconds to wait for the command to execute.

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

By default, the .NET Aspire Sql Server Entity Framework Core integration handles the following:

- Adds the [DbContextHealthCheck](#), which calls EF Core's `CanConnectAsync` method. The name of the health check is the name of the `TContext` type.
- Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire SQL Server Entity Framework Core integration uses the following Log categories:

- `Microsoft.EntityFrameworkCore.ChangeTracking`
- `Microsoft.EntityFrameworkCore.Database.Command`
- `Microsoft.EntityFrameworkCore.Database.Connection`
- `Microsoft.EntityFrameworkCore.Database.Transaction`
- `Microsoft.EntityFrameworkCore.Infrastructure`
- `Microsoft.EntityFrameworkCore.Migrations`
- `Microsoft.EntityFrameworkCore.Model`
- `Microsoft.EntityFrameworkCore.Model.Validation`
- `Microsoft.EntityFrameworkCore.Query`
- `Microsoft.EntityFrameworkCore.Update`

Tracing

The .NET Aspire SQL Server Entity Framework Core integration will emit the following Tracing activities using OpenTelemetry:

- "OpenTelemetry.Instrumentation.EntityFrameworkCore"

Metrics

The .NET Aspire SQL Server Entity Framework Core integration will emit the following metrics using OpenTelemetry:

- Microsoft.EntityFrameworkCore:
 - `ec_Microsoft_EntityFrameworkCore_active_db_contexts`
 - `ec_Microsoft_EntityFrameworkCore_total_queries`
 - `ec_Microsoft_EntityFrameworkCore_queries_per_second`
 - `ec_Microsoft_EntityFrameworkCore_total_save_changes`
 - `ec_Microsoft_EntityFrameworkCore_save_changes_per_second`
 - `ec_Microsoft_EntityFrameworkCore_compiled_query_cache_hit_rate`
 - `ec_Microsoft_Entity_total_execution_strategy_operation_failures`
 - `ec_Microsoft_E_execution_strategy_operation_failures_per_second`
 - `ec_Microsoft_EntityFramew_total_optimistic_concurrency_failures`
 - `ec_Microsoft_EntityF_optimistic_concurrency_failures_per_second`
- [Azure SQL Database documentation](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

.NET Aspire SQL Server integration

Article • 08/29/2024

In this article, you learn how to use the .NET Aspire SQL Server integration. The `Aspire.Microsoft.Data.SqlClient` library:

- Registers a scoped `Microsoft.Data.SqlClient.SqlConnection` factory in the DI container for connecting Azure SQL, MS SQL database.
- Automatically configures the following:
 - Connection pooling to efficiently managed HTTP requests and database connections
 - Automatic retries to increase app resiliency
 - Health checks, logging and telemetry to improve app monitoring and diagnostics

Prerequisites

- An [Azure SQL Database](#) or [SQL Server](#) database and the connection string for accessing the database.

Get started

To get started with the .NET Aspire SQL Server integration, install the `Aspire.Microsoft.Data.SqlClient` NuGet package in the client-consuming project, i.e., the project for the application that uses the SQL Server client.

.NET CLI

.NET CLI

```
dotnet add package Aspire.Microsoft.Data.SqlClient
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

Example usage

In the `Program.cs` file of your client-consuming project, call the [AddSqlServerClient](#) extension to register a `SqlConnection` for use via the dependency injection container.

```
C#
```

```
builder.AddSqlServerClient("sqlDb");
```

To retrieve your `SqlConnection` object an example service:

```
C#
```

```
public class ExampleService(SqlConnection client)
{
    // Use client...
}
```

After adding a `SqlConnection`, you can get the scoped `SqlConnection` instance using DI.

App host usage

To model the SqlServer resource in the app host, install the [Aspire.Hosting.SqlServer](#) NuGet package in the [app host](#) project.

```
.NET CLI
```

```
.NET CLI
```

```
dotnet add package Aspire.Hosting.SqlServer
```

In your app host project, register a SqlServer database and consume the connection using the following methods:

```
C#
```

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

var sql = builder.AddSqlServer("sql");
var sqlDb = sql.AddDatabase("sqlDb");

var myService = builder.AddProject<Projects.MyService>()
    .WithReference(sqlDb);
```

When you want to explicitly provide a root SQL password, you can provide it as a parameter. Consider the following alternative example:

```
C#  
  
var password = builder.AddParameter("password", secret: true);  
  
var sql = builder.AddSqlServer("sql", password);  
var sqldb = sql.AddDatabase("sqldb");  
  
var myService = builder.AddProject<Projects.MyService>()  
    .WithReference(sqldb);
```

For more information, see [External parameters](#).

Configuration

The .NET Aspire SQL Server integration provides multiple configuration approaches and options to meet the requirements and conventions of your project.

Use a connection string

When using a connection string from the `ConnectionStrings` configuration section, you provide the name of the connection string when calling `AddSqlServerClient`:

```
C#  
  
builder.AddSqlServerClient("myConnection");
```

The connection string is retrieved from the `ConnectionStrings` configuration section:

```
JSON  
  
{  
  "ConnectionStrings": {  
    "myConnection": "Data Source=myserver;Initial Catalog=master"  
  }  
}
```

For more information, see the [ConnectionString](#).

Use configuration providers

The .NET Aspire SQL Server supports [Microsoft.Extensions.Configuration](#). It loads the `MicrosoftDataSqlClientSettings` from configuration files such as `appsettings.json` by using the `Aspire:SqlServer:SqlClient` key. If you have set up your configurations in the `Aspire:SqlServer:SqlClient` section, you can just call the method without passing any parameter.

The following example shows an `appsettings.json` file that configures some of the available options:

JSON

```
{  
  "Aspire": {  
    "SqlServer": {  
      "SqlClient": {  
        "ConnectionString": "YOUR_CONNECTIONSTRING",  
        "DisableHealthChecks": false,  
        "DisableMetrics": true  
      }  
    }  
  }  
}
```

Use inline configurations

You can also pass the `Action<MicrosoftDataSqlClientSettings>` delegate to set up some or all the options inline, for example to turn off the `DisableMetrics`:

C#

```
builder.AddSqlServerClient(  
  static settings => settings.DisableMetrics = true);
```

Configuring connections to multiple databases

If you want to add more than one `SqlConnection` you could use named instances. The json configuration would look like:

JSON

```
{  
  "Aspire": {  
    "SqlServer": {  
      "SqlClient": {  
        "INSTANCE_NAME": {  
          "ConnectionString": "YOUR_CONNECTIONSTRING",  
          "DisableHealthChecks": false,  
          "DisableMetrics": true  
        }  
      }  
    }  
  }  
}
```

```
        "ServiceUri": "YOUR_URI",
        "DisableHealthChecks": true
    }
}
}
}
}
```

To load the named configuration section from the json config call the `AddSqlServerClient` method by passing the `INSTANCE_NAME`.

C#

```
builder.AddSqlServerClient("INSTANCE NAME");
```

Configuration options

Here are the configurable options with corresponding default values:

 Expand table

Name	Description
ConnectionString	The connection string of the SQL Server database to connect to.
DisableHealthChecks	A boolean value that indicates whether the database health check is disabled or not.
DisableTracing	A boolean value that indicates whether the OpenTelemetry tracing is disabled or not.
DisableMetrics	A boolean value that indicates whether the OpenTelemetry metrics are disabled or not.

Health checks

By default, .NET Aspire integrations enable [health checks](#) for all services. For more information, see [.NET Aspire integrations overview](#).

By default, the .NET Aspire Sql Server integration handles the following:

- Adds the [SqlServerHealthCheck](#), which verifies that a connection can be made commands can be run against the SQL Database.
 - Integrates with the `/health` HTTP endpoint, which specifies all registered health checks must pass for app to be considered ready to accept traffic

Observability and telemetry

.NET Aspire integrations automatically set up Logging, Tracing, and Metrics configurations, which are sometimes known as *the pillars of observability*. For more information about integration observability and telemetry, see [.NET Aspire integrations overview](#). Depending on the backing service, some integrations may only support some of these features. For example, some integrations support logging and tracing, but not metrics. Telemetry features can also be disabled using the techniques presented in the [Configuration](#) section.

Logging

The .NET Aspire SQL Server integration currently doesn't enable logging by default due to limitations of the `SqlClient`.

Tracing

The .NET Aspire SQL Server integration will emit the following Tracing activities using OpenTelemetry:

- "OpenTelemetry.Instrumentation.SqlClient"

Metrics

The .NET Aspire SQL Server integration will emit the following metrics using OpenTelemetry:

- Microsoft.Data.SqlClient.EventSource
 - `active-hard-connections`
 - `hard-connects`
 - `hard-disconnects`
 - `active-soft-connects`
 - `soft-connects`
 - `soft-disconnects`
 - `number-of-non-pooled-connections`
 - `number-of-pooled-connections`
 - `number-of-active-connection-pool-groups`
 - `number-of-inactive-connection-pool-groups`
 - `number-of-active-connection-pools`
 - `number-of-inactive-connection-pools`

- `number-of-active-connections`
- `number-of-free-connections`
- `number-of-stasis-connections`
- `number-of-reclaimed-connections`

See also

- [Azure SQL Database](#)
- [SQL Server](#)
- [.NET Aspire integrations](#)
- [.NET Aspire GitHub repo ↗](#)

Create custom .NET Aspire hosting integrations

Article • 09/24/2024

.NET Aspire improves the development experience by providing reusable building blocks that can be used to quickly arrange application dependencies and expose them to your own code. One of the key building blocks of an Aspire-based application is the *resource*. Consider the code below:

C#

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

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

var db = builder.AddPostgres("pgserver")
    .AddDatabase("inventorydb");

builder.AddProject<Projects.InventoryService>("inventoryservice")
    .WithReference(redis)
    .WithReference(db);
```

In the preceding code there are four resources represented:

1. `cache`: A Redis container.
2. `pgserver`: A Postgres container.
3. `inventorydb`: A database hosted on `pgserver`.
4. `inventoryservice`: An ASP.NET Core application.

Most .NET Aspire-related code that the average developer writes, centers around adding resources to the [app model](#) and creating references between them.

Key elements of a .NET Aspire custom resource

Building a custom resource in .NET Aspire requires the following:

1. A custom resource type that implements [`IResource`](#)
2. An extension method for [`IDistributedApplicationBuilder`](#) named `Add{CustomResource}` where `{CustomResource}` is the name of the custom resource.

When custom resource requires optional configuration, developers may wish to implement `With*` suffixed extension methods to make these configuration options

discoverable using the *builder pattern*.

A practical example: MailDev

To help understand how to develop custom resources, this article shows an example of how to build a custom resource for [MailDev](#). MailDev is an open-source tool which provides a local mail server designed to allow developers to test e-mail sending behaviors within their app. For more information, see [the MailDev GitHub repository](#).

In this example you create a new .NET Aspire project as a test environment for the MailDev resource that you create. While you can create custom resources in existing .NET Aspire projects it's a good idea to consider whether the custom resource might be used across multiple .NET Aspire-based solutions and should be developed as a reusable integration.

Set up the starter project

Create a new .NET Aspire project that is used to test out the new resource that we're developing.

.NET CLI

```
dotnet new aspire -o MailDevResource  
cd MailDevResource  
dir
```

Once the project is created, you should see a listing containing the following:

- `MailDevResource.AppHost`: The [app host](#) used to test out the custom resource.
- `MailDevResource.ServiceDefaults`: The [service defaults](#) project for use in service-related projects.
- `MailDevResource.sln`: The solution file referencing both projects.

Verify that the project can build and run successfully by executing the following command:

.NET CLI

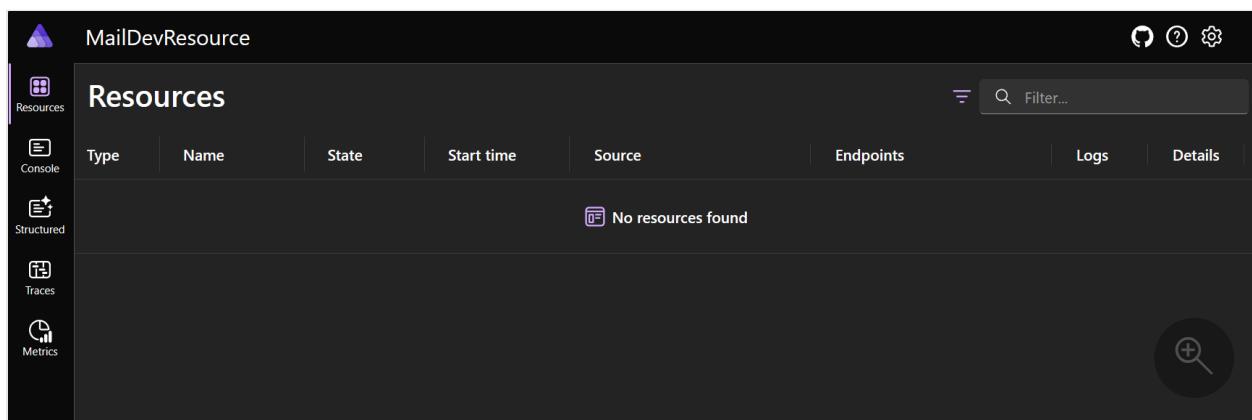
```
dotnet run --project MailDevResource.AppHost/MailDevResource.AppHost.csproj
```

The console output should look similar to the following:

.NET CLI

```
Building...
info: Aspire.Hosting.DistributedApplication[0]
    Aspire version: 8.0.0+d215c528c07c7919c3ac30b35d92f4e51a60523b
info: Aspire.Hosting.DistributedApplication[0]
    Distributed application starting.
info: Aspire.Hosting.DistributedApplication[0]
    Application host directory is:
    ..\docs-
aspire\docs\extensibility\snippets\MailDevResource\MailDevResource.AppHost
info: Aspire.Hosting.DistributedApplication[0]
    Now listening on: https://localhost:17251
info: Aspire.Hosting.DistributedApplication[0]
    Login to the dashboard at https://localhost:17251/login?
t=928db244c720c5022a7a9bf5cf3a3526
info: Aspire.Hosting.DistributedApplication[0]
    Distributed application started. Press Ctrl+C to shut down.
```

Select the [dashboard link in the browser](#) to see the .NET Aspire dashboard:



The screenshot shows the .NET Aspire dashboard interface. On the left, there's a sidebar with icons for 'Console', 'Structured', 'Traces', and 'Metrics'. The main area is titled 'Resources' and has a table with columns: Type, Name, State, Start time, Source, Endpoints, Logs, and Details. A message 'No resources found' is displayed in the center of the table. At the bottom right of the main area is a search icon.

Press **Ctrl + C** to shut down the app (you can close the browser tab).

Create library for resource extension

.NET Aspire resources are just classes and methods contained within a class library that references the .NET Aspire Hosting library (`Aspire.Hosting`). By placing the resource in a separate project, you can more easily share it between .NET Aspire-based apps and potentially package and share it on NuGet.

1. Create the class library project named *MailDev.Hosting*.

.NET CLI

```
dotnet new classlib -o MailDev.Hosting
```

2. Add `Aspire.Hosting` to the class library as a package reference.

.NET CLI

```
dotnet add ./MailDev.Hosting/MailDev.Hosting.csproj package  
Aspire.Hosting --version 8.0.0
```

 **Important**

The version you specify here should match the version of the .NET Aspire workload installed.

3. Add class library reference to the *MailDevResource.AppHost* project.

.NET CLI

```
dotnet add ./MailDevResource.AppHost/MailDevResource.AppHost.csproj  
reference ./MailDev.Hosting/MailDev.Hosting.csproj
```

4. Add class library project to the solution file.

.NET CLI

```
dotnet sln ./MailDevResource.sln add  
./MailDev.Hosting/MailDev.Hosting.csproj
```

Once the following steps are performed, you can launch the project:

.NET CLI

```
dotnet run --project  
./MailDevResource.AppHost/MailDevResource.AppHost.csproj
```

This results in a warning being displayed to the console:

Output

```
.\.nuget\packages\aspire.hosting.apphost\8.0.0\build\Aspire.Hosting.AppHost.  
targets(174,5): warning ASPIRE004:  
'..\MailDev.Hosting\MailDev.Hosting.csproj' is referenced by an A  
spire Host project, but it is not an executable. Did you mean to set  
IsAspireProjectResource="false"? [D:\source\repos\docs-  
aspire\docs\extensibility\snippets\MailDevResource\MailDevResource.AppHost\M
```

```
ailDevRe  
source.AppHost.csproj]
```

This is because .NET Aspire treats project references in the app host as if they're service projects. To tell .NET Aspire that the project reference should be treated as a nonservice project modify the *MailDevResource.AppHost**MailDevResource.AppHost.csproj* files reference to the `MailDev.Hosting` project to be the following:

XML

```
<ItemGroup>  
    <!-- The IsAspireProjectResource attribute tells .NET Aspire to treat this  
        reference as a standard project reference and not attempt to generate  
        a metadata file -->  
    <ProjectReference Include="..\MailDev.Hosting\MailDev.Hosting.csproj"  
        IsAspireProjectResource="false" />  
</ItemGroup>
```

Now when you launch the app host, there's no warning displayed to the console.

Define the resource types

The *MailDev.Hosting* class library contains the resource type and extension methods for adding the resource to the app host. You should first think about the experience that you want to give developers when using your custom resource. In the case of this custom resource, you would want developers to be able to write code like the following:

C#

```
var builder = DistributedApplication.CreateBuilder(args);  
  
var maildev = builder.AddMailDev("maildev");  
  
builder.AddProject<Projects.NewsletterService>("newsletterservice")  
    .WithReference(maildev);
```

To achieve this, you need a custom resource named `MailDevResource` which implements `IResourceWithConnectionString` so that consumers can use it with `WithReference` extension to inject the connection details for the MailDev server as a connection string.

MailDev is available as a container resource, so you'll also want to derive from `ContainerResource` so that we can make use of various pre-existing container-focused extensions in .NET Aspire.

Replace the contents of the `Class1.cs` file in the `MailDev.Hosting` project, and rename the file to `MailDevResource.cs` with the following code:

```
C#  
  
// For ease of discovery, resource types should be placed in  
// the Aspire.Hosting.ApplicationModel namespace. If there is  
// likelihood of a conflict on the resource name consider using  
// an alternative namespace.  
namespace Aspire.Hosting.ApplicationModel;  
  
public sealed class MailDevResource(string name) : ContainerResource(name),  
IResourceWithConnectionString  
{  
    // Constants used to refer to well known-endpoint names, this is  
    // specific  
    // for each resource type. MailDev exposes an SMTP endpoint and a HTTP  
    // endpoint.  
    internal const string SmtpEndpointName = "smtp";  
    internal const string HttpEndpointName = "http";  
  
    // An EndpointReference is a core .NET Aspire type used for keeping  
    // track of endpoint details in expressions. Simple literal values  
    cannot  
    // be used because endpoints are not known until containers are  
    launched.  
    private EndpointReference? _smtpReference;  
  
    public EndpointReference SmtpEndpoint =>  
        _smtpReference ??= new(this, SmtpEndpointName);  
  
    // Required property on IResourceWithConnectionString. Represents a  
    connection  
    // string that applications can use to access the MailDev server. In  
    this case  
    // the connection string is composed of the SmtpEndpoint endpoint  
    reference.  
    public ReferenceExpression ConnectionStringExpression =>  
        ReferenceExpression.Create(  
            $"smtp://{{SmtpEndpoint.Property(EndpointProperty.Host)}}:  
{{SmtpEndpoint.Property(EndpointProperty.Port)}}"  
        );  
}
```

In the preceding custom resource, the `EndpointReference` and `ReferenceExpression` are examples of several types which implement a collection of interfaces, such as `IManifestExpressionProvider`, `IValueProvider`, and `IValueWithReferences`. For more information about these types and their role in .NET Aspire, see [technical details](#).

Define the resource extensions

To make it easy for developers to use the custom resource an extension method named `AddMailDev` needs to be added to the `MailDev.Hosting` project. The `AddMailDev` extension method is responsible for configuring the resource so it can start successfully as a container.

Add the following code to a new file named `MailDevResourceBuilderExtensions.cs` in the `MailDev.Hosting` project:

```
C#  
  
using Aspire.Hosting.ApplicationModel;  
  
// Put extensions in the Aspire.Hosting namespace to ease discovery as  
// referencing  
// the .NET Aspire hosting package automatically adds this namespace.  
namespace Aspire.Hosting;  
  
public static class MailDevResourceBuilderExtensions  
{  
    /// <summary>  
    /// Adds the <see cref="MailDevResource"/> to the given  
    /// <paramref name="builder"/> instance. Uses the "2.0.2" tag.  
    /// </summary>  
    /// <param name="builder">The <see  
    cref="IDistributedApplicationBuilder"/>.</param>  
    /// <param name="name">The name of the resource.</param>  
    /// <param name="httpPort">The HTTP port.</param>  
    /// <param name="smtpPort">The SMTP port.</param>  
    /// <returns>  
    /// An <see cref="IResourceBuilder<MailDevResource>"/> instance that  
    /// represents the added MailDev resource.  
    /// </returns>  
    public static IResourceBuilder<MailDevResource> AddMailDev(  
        this IDistributedApplicationBuilder builder,  
        string name,  
        int? httpPort = null,  
        int? smtpPort = null)  
    {  
        // The AddResource method is a core API within .NET Aspire and is  
        // used by resource developers to wrap a custom resource in an  
        // IResourceBuilder<T> instance. Extension methods to customize  
        // the resource (if any exist) target the builder interface.  
        var resource = new MailDevResource(name);  
  
        return builder.AddResource(resource)  
            .WithImage(MailDevContainerImageTags.Image)  
            .WithImageRegistry(MailDevContainerImageTags.Registry)  
            .WithImageTag(MailDevContainerImageTags.Tag)  
            .WithHttpEndpoint(  
                targetPort: 1080,  
                port: httpPort,  
                name: MailDevResource.HttpEndpointName)  
    }  
}
```

```

        .WithEndpoint(
            targetPort: 1025,
            port: smtpPort,
            name: MailDevResource.SmtpEndpointName);
    }
}

// This class just contains constant strings that can be updated
// periodically
// when new versions of the underlying container are released.
internal static class MailDevContainerImageTags
{
    internal const string Registry = "docker.io";

    internal const string Image = "maildev/maildev";

    internal const string Tag = "2.0.2";
}

```

Validate custom integration inside the app host

Now that the basic structure for the custom resource is complete it's time to test it in a real AppHost project. Open the *Program.cs* file in the *MailDevResource.AppHost* project and update it with the following code:

C#

```

var builder = DistributedApplication.CreateBuilder(args);

var maildev = builder.AddMailDev("maildev");

builder.Build().Run();

```

After updating the *Program.cs* file, launch the app host project and open the dashboard:

.NET CLI

```

dotnet run --project
./MailDevResource.AppHost/MailDevResource.AppHost.csproj

```

After a few moments the dashboard shows that the `maildev` resource is running and a hyperlink will be available that navigates to the MailDev web app, which shows the content of each e-mail that your app sends.

The .NET Aspire dashboard should look similar to the following:

The screenshot shows the .NET Aspire interface under the 'MailDevResource' project. On the left, there's a sidebar with icons for 'Console', 'Structured', 'Traces', and 'Metrics'. The main area is titled 'Resources' and contains a table with one row. The table columns are 'Type' (Container), 'Name' (maildev), 'State' (Running), 'Start time' (1:21:25 PM), 'Source' (docker.io/maildev/maildev:2.0.2), 'Endpoints' (http://localhost:61389), 'Logs' (View), and 'Details' (View). A search bar at the top right says 'Filter...'. A magnifying glass icon is in the bottom right corner.

The MailDev web app should look similar to the following:

The screenshot shows the MailDev web application. At the top, there's a header with a mail icon, the text 'MailDev', and several small icons. Below the header is a search bar with the placeholder 'Search...'. The main content area has a heading 'Now receiving all emails on port 1025'. It includes instructions for sending emails, a link to the 'Configure your project' guide, and a link to the 'GitHub repository'. It also states 'You are running MailDev 2.0.2'. A magnifying glass icon is in the bottom right corner.

Add a .NET service project to the app host for testing

Once .NET Aspire can successfully launch the MailDev integration, it's time to consume the connection information for MailDev within a .NET project. In .NET Aspire it's common for there to be a *hosting package* and one or more *component packages*. For example consider:

- **Hosting package:** Used to represent resources within the app model.
 - `Aspire.Hosting.Redis`
- **Component packages:** Used to configure and consume client libraries.
 - `Aspire.StackExchange.Redis`
 - `Aspire.StackExchange.Redis.DistributedCaching`
 - `Aspire.StackExchange.Redis.OutputCaching`

In the case of the MailDev resource, the .NET platform already has a simple mail transfer protocol (SMTP) client in the form of `SmtpClient`. In this example you use this existing API for the sake of simplicity, although other resource types may benefit from custom integration libraries to assist developers.

In order to test the end-to-end scenario, you need a .NET project which we can inject the connection information into for the MailDev resource. Add a Web API project:

1. Create a new .NET project named `MailDevResource.NewsletterService`.

```
dotnet new webapi --use-minimal-apis -o  
MailDevResource.NewsletterService
```

2. Add a reference to the *MailDev.Hosting* project.

.NET CLI

```
dotnet add  
.~/MailDevResource.NewsletterService/MailDevResource.NewsletterService.c  
sproj reference ./MailDev.Hosting/MailDev.Hosting.csproj
```

3. Add a reference to the *MailDevResource.AppHost* project.

.NET CLI

```
dotnet add ./MailDevResource.AppHost/MailDevResource.AppHost.csproj  
reference  
.~/MailDevResource.NewsletterService/MailDevResource.NewsletterService.c  
sproj
```

4. Add the new project to the solution file.

.NET CLI

```
dotnet sln ./MailDevResource.sln add  
.~/MailDevResource.NewsletterService/MailDevResource.NewsletterService.c  
sproj
```

After the project has been added and references have been updated, open the *Program.cs* of the *MailDevResource.AppHost.csproj* project, and update the source file to look like the following:

C#

```
var builder = DistributedApplication.CreateBuilder(args);  
  
var maildev = builder.AddMailDev("maildev");  
  
builder.AddProject<Projects.MailDevResource_NewsletterService>  
("newsletterservice")  
    .WithReference(maildev);  
  
builder.Build().Run();
```

After updating the `Program.cs` file, launch the app host again. Then verify that the Newsletter Service started and that the environment variable `ConnectionStrings__maildev` was added to the process. From the **Resources** page, find the `newsletterservice` row, and select the **View** link on the **Details** column:

The screenshot shows the Azure DevOps Resources page. On the left, there's a sidebar with icons for Console, Structured, Traces, and Metrics. The main area has a table titled "Resources" with columns: Type, Name, State, Start time, Source, Endpoints, Logs, and Details. Two items are listed: "Container maildev" (Running, docker.io/maildev/maildev:2.0.2, http://localhost:62775) and "Project newsletterservice" (Running, MailDevResource.NewsletterService.cs..., https://localhost:7251/weatherforecast). Below this, a section titled "Project: newsletterservice" shows a table for "Environment variables". A red arrow points from the text in the preceding paragraph to the "Environment variables" section. The table has columns: Name and Value. It lists several variables: ASPNETCORE_ENVIRONMENT (value: *****), ASPNETCORE_HTTPS_PORT (value: *****), ASPNETCORE_URLS (value: *****), ConnectionStrings__maildev (value: smtp://localhost:62775, highlighted with a red box), and DOTNET_SYSTEM_CONSOLE_ALLOW_ANSI_COLOR_REDIRECTION (value: *****).

The preceding screenshot shows the environment variables for the `newsletterservice` project. The `ConnectionStrings__maildev` environment variable is the connection string that was injected into the project by the `maildev` resource.

Use connection string to send messages

To use the SMTP connection details that were injected into the newsletter service project, you inject an instance of `SmtpClient` into the dependency injection container as a singleton. Add the following code to the `Program.cs` file in the `MailDevResource.NewsletterService` project to set up the singleton service. In the `Program` class, immediately following the `// Add services to the container` comment, add the following code:

```
C#  
  
builder.Services.AddSingleton<SmtpClient>(sp =>  
{  
    var smtpUri = new  
    Uri(builder.Configuration.GetConnectionString("maildev"));  
  
    var smtpClient = new SmtpClient(smtpUri.Host, smtpUri.Port);  
  
    return smtpClient;  
});
```

💡 Tip

This code snippet relies on the official `SmtpClient`, however; this type is obsolete on some platforms and not recommended on others. For a more modern approach using [MailKit](#), see [Create custom .NET Aspire client integrations](#).

To test the client, add two simple `subscribe` and `unsubscribe` POST methods to the newsletter service. Add the following code replacing the "weatherforecast" `MapGet` call in the `Program.cs` file of the `MailDevResource.NewsletterService` project to set up the ASP.NET Core routes:

```
C#  
  
app.MapPost("/subscribe", async (SmtpClient smtpClient, string email) =>  
{  
    using var message = new MailMessage("newsletter@yourcompany.com", email)  
    {  
        Subject = "Welcome to our newsletter!",  
        Body = "Thank you for subscribing to our newsletter!"  
    };  
  
    await smtpClient.SendMailAsync(message);  
});  
  
app.MapPost("/unsubscribe", async (SmtpClient smtpClient, string email) =>  
{  
    using var message = new MailMessage("newsletter@yourcompany.com", email)  
    {  
        Subject = "You are unsubscribed from our newsletter!",  
        Body = "Sorry to see you go. We hope you will come back soon!"  
    };  
  
    await smtpClient.SendMailAsync(message);  
});
```

💡 Tip

Remember to reference the `System.Net.Mail` and `Microsoft.AspNetCore.Mvc` namespaces in `Program.cs` if your code editor doesn't automatically add them.

Once the `Program.cs` file is updated, launch the app host and use your browser, or `curl` to hit the following URLs (alternatively if you're using Visual Studio you can use `.http` files):

```
POST /subscribe?email=test@test.com HTTP/1.1
```

```
Host: localhost:7251
```

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

To use this API, you can use `curl` to send the request. The following `curl` command sends an HTTP `POST` request to the `subscribe` endpoint, and it expects an `email` query string value to subscribe to the newsletter. The `Content-Type` header is set to `application/json` to indicate that the request body is in JSON format.:

Windows

PowerShell

```
curl -H @{ ContentType = "application/json" } -Method POST  
https://localhost:7251/subscribe?email=test@test.com
```

The next API is the `unsubscribe` endpoint. This endpoint is used to unsubscribe from the newsletter.

HTTP

```
POST /unsubscribe?email=test@test.com HTTP/1.1
```

```
Host: localhost:7251
```

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

To unsubscribe from the newsletter, you can use the following `curl` command, passing an `email` parameter to the `unsubscribe` endpoint as a query string:

Windows

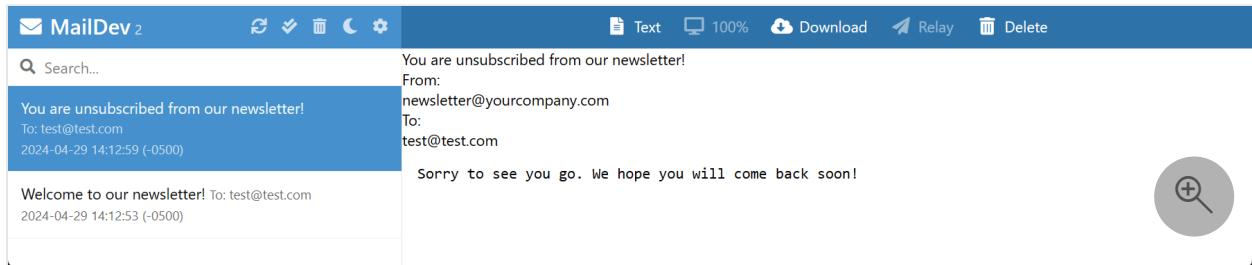
PowerShell

```
curl -H @{ ContentType = "application/json" } -Method POST  
https://localhost:7251/unsubscribe?email=test@test.com
```

💡 Tip

Make sure that you replace the `https://localhost:7251` with the correct localhost port (the URL of the app host that you are running).

If those API calls return a successful response (HTTP 200, Ok) then you should be able to select on the `maildev` resource the dashboard and the MailDev UI will show the emails that have been sent to the SMTP endpoint.



Technical details

In the following sections, various technical details are discussed which are important to understand when developing custom resources for .NET Aspire.

Secure networking

In this example, the `MailDev` resource is a container resource which is exposed to the host machine over HTTP and SMTP. The `MailDev` resource is a development tool and isn't intended for production use. To instead use HTTPS, see [MailDev: Configure HTTPS](#).

When developing custom resources that expose network endpoints, it's important to consider the security implications of the resource. For example, if the resource is a database, it's important to ensure that the database is secure and that the connection string isn't exposed to the public internet.

The `ReferenceExpression` and `EndpointReference` type

In the preceding code, the `MailDevResource` had two properties:

- `SmtpEndpoint`: `EndpointReference` type.
- `ConnectionStringExpression`: `ReferenceExpression` type.

These types are among several which are used throughout .NET Aspire to represent configuration data, which isn't finalized until the .NET Aspire project is either run or published to the cloud via a tool such as [Azure Developer CLI \(azd\)](#).

The fundamental problem that these types help to solve, is deferring resolution of concrete configuration information until *all* the information is available.

For example, the `MailDevResource` exposes a property called `ConnectionStringExpression` as required by the [IResourceWithConnectionString](#) interface. The type of the property is [ReferenceExpression](#) and is created by passing in an interpolated string to the [Create](#) method.

C#

```
public ReferenceExpression ConnectionStringExpression =>
    ReferenceExpression.Create(
        $"smtp://{{SmtpEndpoint.Property(EndpointProperty.Host)}:{{SmtpEndpoint.Property(EndpointProperty.Port)}}}"
    );
```

The signature for the [Create](#) method is as follows:

C#

```
public static ReferenceExpression Create(
    in ExpressionInterpolatedStringHandler handler)
```

This isn't a regular [String](#) argument. The method makes use of the [interpolated string handler pattern](#), to capture the interpolated string template and the values referenced within it to allow for custom processing. In the case of .NET Aspire, these details are captured in a [ReferenceExpression](#) which can be evaluated as each value referenced in the interpolated string becomes available.

Here's how the flow of execution works:

1. A resource which implements [IResourceWithConnectionString](#) is added to the model (for example, `AddMailDev(...)`).
2. The `IResourceBuilder<MailDevResource>` is passed to the [WithReference](#) which has a special overload for handling [IResourceWithConnectionString](#) implementors.
3. The [WithReference](#) wraps the resource in a [ConnectionStringReference](#) instance and the object is captured in a [EnvironmentCallbackAnnotation](#) which is evaluated after the .NET Aspire project is built and starts running.
4. As the process that references the connection string starts .NET Aspire starts evaluating the expression. It first gets the [ConnectionStringReference](#) and calls [IValueProvider.GetValueAsync](#).
5. The `GetValueAsync` method gets the value of the [ConnectionStringExpression](#) property to get the [ReferenceExpression](#) instance.
6. The [IValueProvider.GetValueAsync](#) method then calls [GetValueAsync](#) to process the previously captured interpolated string.

7. Because the interpolated string contains references to other reference types such as `EndpointReference` they're also evaluated and real value substituted (which at this time is now available).

Manifest publishing

The `IManifestExpressionProvider` interface is designed to solve the problem of sharing connection information between resources at deployment. The solution for this particular problem is described in the [.NET Aspire inner-loop networking overview](#). Similarly to local development, many of the values are necessary to configure the app, yet they can't be determined until the app is being deployed via a tool, such as `azd` (Azure Developer CLI).

To solve this problem [.NET Aspire produces a manifest file](#) which `azd` and other deployment tools interpret. Rather than specifying concrete values for connection information between resources an expression syntax is used which deployment tools evaluate. Generally the manifest file isn't visible to developers but it's possible to generate one for manual inspection. The command below can be used on the app host to produce a manifest.

.NET CLI

```
dotnet run --project MailDevResource.AppHost/MailDevResource.AppHost.csproj  
-- --publisher manifest --output-path aspire-manifest.json
```

This command produces a manifest file like the following:

JSON

```
{  
  "resources": {  
    "maildev": {  
      "type": "container.v0",  
      "connectionString": "smtp://{{maildev.bindings.smtp.host}}:  
{maildev.bindings.smtp.port}",  
      "image": "docker.io/maildev/maildev:2.0.2",  
      "bindings": {  
        "http": {  
          "scheme": "http",  
          "protocol": "tcp",  
          "transport": "http",  
          "targetPort": 1080  
        },  
        "smtp": {  
          "scheme": "tcp",  
          "protocol": "tcp",  
          "port": 1025  
        }  
      }  
    }  
  }  
}
```

```

        "transport": "tcp",
        "targetPort": 1025
    }
}
},
"newsletterservice": {
    "type": "project.v0",
    "path":
"../MailDevResource.NewsletterService/MailDevResource.NewsletterService.cspr
oj",
    "env": {
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EXCEPTION_LOG_ATTRIBUTES": "true",
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EVENT_LOG_ATTRIBUTES": "true",
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_RETRY": "in_memory",
        "ASPNETCORE_FORWARDEDHEADERS_ENABLED": "true",
        "ConnectionStrings__maildev": "{maildev.connectionString}"
    },
    "bindings": {
        "http": {
            "scheme": "http",
            "protocol": "tcp",
            "transport": "http"
        },
        "https": {
            "scheme": "https",
            "protocol": "tcp",
            "transport": "http"
        }
    }
}
}
}

```

Because the `MailDevResource` implements `IResourceWithConnectionString` the manifest publishing logic in .NET Aspire knows that even though `MailDevResource` is a container resource, it also needs a `connectionString` field. The `connectionString` field references other parts of the `maildev` resource in the manifest to produce the final string:

JSON

```
{
    // ... other content omitted.
    "connectionString": "smtp://{{maildev.bindings.smtp.host}:
{maildev.bindings.smtp.port}}"
}
```

.NET Aspire knows how to form this string because it looks at `ConnectionStringExpression` and builds up the final string via the

[IManifestExpressionProvider](#) interface (in much the same way as the [IValueProvider](#) interface is used).

The `MailDevResource` automatically gets included in the manifest because it's derived from [ContainerResource](#). Resource authors can choose to suppress outputting content to the manifest by using the [ExcludeFromManifest](#) extension method on the resource builder.

C#

```
public static IResourceBuilder<MailDevResource> AddMailDev(
    this IDistributedApplicationBuilder builder,
    string name,
    int? httpPort = null,
    int? smtpPort = null)
{
    var resource = new MailDevResource(name);

    return builder.AddResource(resource)
        .WithImage(MailDevContainerImageTags.Image)
        .WithImageRegistry(MailDevContainerImageTags.Registry)
        .WithImageTag(MailDevContainerImageTags.Tag)
        .WithHttpEndpoint(
            targetPort: 1080,
            port: httpPort,
            name: MailDevResource.HttpEndpointName)
        .WithEndpoint(
            targetPort: 1025,
            port: smtpPort,
            name: MailDevResource.SmtpEndpointName)
        .ExcludeFromManifest(); // This line was added
}
```

Careful consideration should be given as to whether the resource should be present in the manifest, or whether it should be suppressed. If the resource is being added to the manifest, it should be configured in such a way that it's safe and secure to use.

Summary

In the custom resource tutorial, you learned how to create a custom .NET Aspire resource which uses an existing containerized application (MailDev). You then used that to improve the local development experience by making it easy to test e-mail capabilities that might be used within an app. These learnings can be applied to building out other custom resources that can be used in .NET Aspire-based applications. This specific example didn't include any custom integrations, but it's possible to build out custom integrations to make it easier for developers to use the resource. In this

scenario you were able to rely on the existing `SmtpClient` class in the .NET platform to send e-mails.

Next steps

[Create custom .NET Aspire client integrations](#)

Create custom .NET Aspire client integrations

Article • 09/24/2024

This article is a continuation of the [Create custom .NET Aspire hosting integrations](#) article. It guides you through creating a .NET Aspire client integration that uses [MailKit](#) to send emails. This integration is then added into the Newsletter app you previously built. The previous example omitted the creation of a client integration and instead relied on the existing .NET `SmtpClient`. It's best to use MailKit's `SmtpClient` over the official .NET `SmtpClient` for sending emails, as it's more modern and supports more features/protocols. For more information, see [.NET SmtpClient: Remarks](#).

Prerequisites

If you're following along, you should have a Newsletter app from the steps in the [Create custom .NET Aspire hosting integration](#) article.

💡 Tip

This article is inspired by existing .NET Aspire integrations, and based on the team's official guidance. There are places where said guidance varies, and it's important to understand the reasoning behind the differences. For more information, see [.NET Aspire integration requirements](#).

Create library for integration

[.NET Aspire integrations](#) are delivered as NuGet packages, but in this example, it's beyond the scope of this article to publish a NuGet package. Instead, you create a class library project that contains the integration and reference it as a project. .NET Aspire integration packages are intended to wrap a client library, such as MailKit, and provide production-ready telemetry, health checks, configurability, and testability. Let's start by creating a new class library project.

1. Create a new class library project named `MailKit.Client` in the same directory as the `MailDevResource.sln` from the previous article.

.NET CLI

```
dotnet new classlib -o MailKit.Client
```

2. Add the project to the solution.

.NET CLI

```
dotnet sln ./MailDevResource.sln add  
MailKit.Client/MailKit.Client.csproj
```

The next step is to add all the NuGet packages that the integration relies on. Rather than having you add each package one-by-one from the .NET CLI, it's likely easier to copy and paste the following XML into the *MailKit.Client.csproj* file.

XML

```
<ItemGroup>  
  <PackageReference Include="MailKit" Version="4.7.1.1" />  
  <PackageReference Include="Microsoft.Extensions.Configuration.Binder"  
Version="8.2.0" />  
  <PackageReference Include="Microsoft.Extensions.Resilience"  
Version="8.7.0" />  
  <PackageReference Include="Microsoft.Extensions.Hosting.Abstractions"  
Version="8.0.0" />  
  <PackageReference Include="Microsoft.Extensions.Diagnostics.HealthChecks"  
Version="8.0.7" />  
  <PackageReference Include="OpenTelemetry.Extensions.Hosting"  
Version="1.9.0" />  
</ItemGroup>
```

Define integration settings

Whenever you're creating a .NET Aspire integration, it's best to understand the client library that you're mapping to. With MailKit, you need to understand the configuration settings that are required to connect to a Simple Mail Transfer Protocol (SMTP) server. But it's also important to understand if the library has support for *health checks*, *tracing* and *metrics*. MailKit supports *tracing* and *metrics*, through its [Telemetry.SmtpClient class](#). When adding *health checks*, you should use any established or existing health checks where possible. Otherwise, you might consider implementing your own in the integration. Add the following code to the `MailKit.Client` project in a file named `MailKitClientSettings.cs`:

C#

```
using System.Data.Common;

namespace MailKit.Client;

/// <summary>
/// Provides the client configuration settings for connecting MailKit to an
/// SMTP server.
/// </summary>
public sealed class MailKitClientSettings
{
    internal const string DefaultConfigSectionName = "MailKit:Client";

    /// <summary>
    /// Gets or sets the SMTP server <see cref="Uri"/>.
    /// </summary>
    /// <value>
    /// The default value is <see langword="null"/>.
    /// </value>
    public Uri? Endpoint { get; set; }

    /// <summary>
    /// Gets or sets a boolean value that indicates whether the database
    /// health check is disabled or not.
    /// </summary>
    /// <value>
    /// The default value is <see langword="false"/>.
    /// </value>
    public bool DisableHealthChecks { get; set; }

    /// <summary>
    /// Gets or sets a boolean value that indicates whether the
    /// OpenTelemetry tracing is disabled or not.
    /// </summary>
    /// <value>
    /// The default value is <see langword="false"/>.
    /// </value>
    public bool DisableTracing { get; set; }

    /// <summary>
    /// Gets or sets a boolean value that indicates whether the
    /// OpenTelemetry metrics are disabled or not.
    /// </summary>
    /// <value>
    /// The default value is <see langword="false"/>.
    /// </value>
    public bool DisableMetrics { get; set; }

    internal void ParseConnectionString(string? connectionString)
    {
        if (string.IsNullOrWhiteSpace(connectionString))
        {
            throw new InvalidOperationException($"""
                ConnectionString is missing.
                It should be provided in 'ConnectionStrings':
            """);
        }
    }
}
```

```

<connectionName>'  

        or '{DefaultConfigSectionName}:Endpoint' key.  

        configuration section.  

        """");  

    }  

    if (Uri.TryCreate(connectionString, UriKind.Absolute, out var uri))  

    {  

        Endpoint = uri;  

    }  

    else  

    {  

        var builder = new DbConnectionStringBuilder  

        {  

            ConnectionString = connectionString  

        };  

        if (builder.TryGetValue("Endpoint", out var endpoint) is false)  

        {  

            throw new InvalidOperationException($"""  

                The 'ConnectionStrings:<connectionName>' (or  

'Endpoint' key in  

                '{DefaultConfigSectionName}') is missing.  

                """");  

        }  

        if (Uri.TryCreate(endpoint.ToString(), UriKind.Absolute, out  

uri) is false)  

        {  

            throw new InvalidOperationException($"""  

                The 'ConnectionStrings:<connectionName>' (or  

'Endpoint' key in  

                '{DefaultConfigSectionName}') isn't a valid URI.  

                """");  

        }  

        Endpoint = uri;  

    }  

}
}

```

The preceding code defines the `MailKitClientSettings` class with:

- `Endpoint` property that represents the connection string to the SMTP server.
- `DisableHealthChecks` property that determines whether health checks are enabled.
- `DisableTracing` property that determines whether tracing is enabled.
- `DisableMetrics` property that determines whether metrics are enabled.

Parse connection string logic

The settings class also contains a `ParseConnectionString` method that parses the connection string into a valid `Uri`. The configuration is expected to be provided in the following format:

- `ConnectionStrings:<connectionName>`: The connection string to the SMTP server.
- `MailKit:Client:ConnectionString`: The connection string to the SMTP server.

If neither of these values are provided, an exception is thrown.

Expose client functionality

The goal of .NET Aspire integrations is to expose the underlying client library to consumers through dependency injection. With MailKit and for this example, the `SmtpClient` class is what you want to expose. You're not wrapping any functionality, but rather mapping configuration settings to an `SmtpClient` class. It's common to expose both standard and keyed-service registrations for integrations. Standard registrations are used when there's only one instance of a service, and keyed-service registrations are used when there are multiple instances of a service. Sometimes, to achieve multiple registrations of the same type you use a factory pattern. Add the following code to the `MailKit.Client` project in a file named *MailKitClientFactory.cs*:

C#

```
using MailKit.Net.Smtp;

namespace MailKit.Client;

/// <summary>
/// A factory for creating <see cref="ISmtpClient"/> instances
/// given a <paramref name="smtpUri"/> (and optional <paramref
/// name="credentials"/>).
/// </summary>
/// <param name="settings">
/// The <see cref="MailKitClientSettings"/> settings for the SMTP server
/// </param>
public sealed class MailKitClientFactory(MailKitClientSettings settings) : IDisposable
{
    private readonly SemaphoreSlim _semaphore = new(1, 1);

    private SmtpClient? _client;

    /// <summary>
    /// Gets an <see cref="ISmtpClient"/> instance in the connected state
    /// (and that's been authenticated if configured).
    /// </summary>
    /// <param name="cancellationToken">Used to abort client creation and
}
```

```

connection.</param>
    /// <returns>A connected (and authenticated) <see cref="ISsmtpClient"/>
instance.</returns>
    /// <remarks>
    /// Since both the connection and authentication are considered
    expensive operations,
    /// the <see cref="ISsmtpClient"/> returned is intended to be used for
    the duration of a request
    /// (registered as 'Scoped') and is automatically disposed of.
    /// </remarks>
public async Task<ISsmtpClient> GetSmtpClientAsync(
    CancellationToken cancellationToken = default)
{
    await _semaphore.WaitAsync(cancellationToken);

    try
    {
        if (_client is null)
        {
            _client = new SmtpClient();

            await _client.ConnectAsync(settings.Endpoint,
cancellationToken)
                .ConfigureAwait(false);
        }
    }
    finally
    {
        _semaphore.Release();
    }

    return _client;
}

public void Dispose()
{
    _client?.Dispose();
    _semaphore.Dispose();
}
}

```

The `MailKitClientFactory` class is a factory that creates an `ISsmtpClient` instance based on the configuration settings. It's responsible for returning an `ISsmtpClient` implementation that has an active connection to a configured SMTP server. Next, you need to expose the functionality for the consumers to register this factory with the dependency injection container. Add the following code to the `MailKit.Client` project in a file named `MailKitExtensions.cs`:

C#

```
using MailKit;
using MailKit.Client;
using MailKit.Net.Smtp;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace Microsoft.Extensions.Hosting;

/// <summary>
/// Provides extension methods for registering a <see cref="SmtpClient"/> as
/// a
/// scoped-lifetime service in the services provided by the <see
/// cref="IHostApplicationBuilder"/>.
/// </summary>
public static class MailKitExtensions
{
    /// <summary>
    /// Registers 'Scoped' <see cref="MailKitClientFactory" /> for creating
    /// connected <see cref="SmtpClient"/> instance for sending emails.
    /// </summary>
    /// <param name="builder">
    /// The <see cref="IHostApplicationBuilder" /> to read config from and
    /// add services to.
    /// </param>
    /// <param name="connectionName">
    /// A name used to retrieve the connection string from the
    /// ConnectionStrings configuration section.
    /// </param>
    /// <param name="configureSettings">
    /// An optional delegate that can be used for customizing options.
    /// It's invoked after the settings are read from the configuration.
    /// </param>
    public static void AddMailKitClient(
        IHostApplicationBuilder builder,
        string connectionName,
        Action<MailKitClientSettings>? configureSettings = null) =>
        AddMailKitClient(
            builder,
            MailKitClientSettings.DefaultConfigSectionName,
            configureSettings,
            connectionName,
            serviceKey: null);

    /// <summary>
    /// Registers 'Scoped' <see cref="MailKitClientFactory" /> for creating
    /// connected <see cref="SmtpClient"/> instance for sending emails.
    /// </summary>
    /// <param name="builder">
    /// The <see cref="IHostApplicationBuilder" /> to read config from and
    /// add services to.
    /// </param>
    /// <param name="name">
    /// The name of the component, which is used as the <see
    /// cref="ServiceDescriptor.ServiceKey"/> of the

```

```
    /// service and also to retrieve the connection string from the
    ConnectionStrings configuration section.
    /// </param>
    /// <param name="configureSettings">
    /// An optional method that can be used for customizing options. It's
    invoked after the settings are
    /// read from the configuration.
    /// </param>
    public static void AddKeyedMailKitClient(
        this IHostApplicationBuilder builder,
        string name,
        Action<MailKitClientSettings>? configureSettings = null)
    {
        ArgumentNullException.ThrowIfNull(name);

        AddMailKitClient(
            builder,
            $"{MailKitClientSettings.DefaultConfigSectionName}:{name}",
            configureSettings,
            connectionName: name,
            serviceKey: name);
    }

    private static void AddMailKitClient(
        this IHostApplicationBuilder builder,
        string configurationSectionName,
        Action<MailKitClientSettings>? configureSettings,
        string connectionName,
        object? serviceKey)
    {
        ArgumentNullException.ThrowIfNull(builder);

        var settings = new MailKitClientSettings();

        builder.Configuration
            .GetSection(configurationSectionName)
            .Bind(settings);

        if (builder.Configuration.GetConnectionString(connectionName) is
string connectionString)
        {
            settings.ParseConnectionString(connectionString);
        }

        configureSettings?.Invoke(settings);

        if (serviceKey is null)
        {
            builder.Services.AddScoped(CreateMailKitClientFactory);
        }
        else
        {
            builder.Services.AddKeyedScoped(serviceKey, (sp, key) =>
CreateMailKitClientFactory(sp));
        }
    }
}
```

```
MailKitClientFactory CreateMailKitClientFactory(IServiceProvider _)
{
    return new MailKitClientFactory(settings);
}

if (settings.DisableHealthChecks is false)
{
    builder.Services.AddHealthChecks()
        .AddCheck<MailKitHealthCheck>(
            name: serviceKey is null ? "MailKit" :
            $"MailKit_{connectionName}",
            failureStatus: default,
            tags: []));
}

if (settings.DisableTracing is false)
{
    builder.Services.AddOpenTelemetry()
        .WithTracing(
            traceBuilder => traceBuilder.AddSource(
                Telemetry.SmtpClient.ActivitySourceName));
}

if (settings.DisableMetrics is false)
{
    // Required by MailKit to enable metrics
    Telemetry.SmtpClient.Configure();

    builder.Services.AddOpenTelemetry()
        .WithMetrics(
            metricsBuilder => metricsBuilder.AddMeter(
                Telemetry.SmtpClient.MeterName));
}
}
```

The preceding code adds two extension methods on the `IHostApplicationBuilder` type, one for the standard registration of MailKit and another for keyed-registration of MailKit.

Tip

Extension methods for .NET Aspire integrations should extend the `IHostApplicationBuilder` type and follow the `Add<MeaningfulName>` naming convention where the `<MeaningfulName>` is the type or functionality you're adding. For this article, the `AddMailKitClient` extension method is used to add the MailKit client. It's likely more in-line with the official quidance to use `AddMailKitSmtpClient`

instead of `AddMailKitClient`, since this only registers the `Smtplib` and not the entire MailKit library.

Both extensions ultimately rely on the private `AddMailKitClient` method to register the `MailKitClientFactory` with the dependency injection container as a [scoped service](#). The reason for registering the `MailKitClientFactory` as a scoped service is because the connection operations are considered expensive and should be reused within the same scope where possible. In other words, for a single request, the same `ISmtplib` instance should be used. The factory holds on to the instance of the `Smtplib` that it creates and disposes of it.

Configuration binding

One of the first things that the private implementation of the `AddMailKitClient` methods does, is to bind the configuration settings to the `MailKitClientSettings` class. The settings class is instantiated and then `Bind` is called with the specific section of configuration. Then the optional `configureSettings` delegate is invoked with the current settings. This allows the consumer to further configure the settings, ensuring that manual code settings are honored over configuration settings. After that, depending on whether the `serviceKey` value was provided, the `MailKitClientFactory` should be registered with the dependency injection container as either a standard or keyed service.

Important

It's intentional that the `implementationFactory` overload is called when registering services. The `createMailKitClientFactory` method throws when the configuration is invalid. This ensures that creation of the `MailKitClientFactory` is deferred until it's needed and it prevents the app from erroring out before logging is available.

The registration of health checks, and telemetry are described in a bit more detail in the following sections.

Add health checks

[Health checks](#) are a way to monitor the health of an integration. With MailKit, you can check if the connection to the SMTP server is healthy. Add the following code to the `MailKit.Client` project in a file named `MailKitHealthCheck.cs`:

```

using Microsoft.Extensions.Diagnostics.HealthChecks;

namespace MailKit.Client;

internal sealed class MailKitHealthCheck(MailKitClientFactory factory) : IHealthCheck
{
    public async Task<HealthCheckResult> CheckHealthAsync(
        HealthCheckContext context,
        CancellationToken cancellationToken = default)
    {
        try
        {
            // The factory connects (and authenticates).
            _ = await factory.GetSmtpClientAsync(cancellationToken);

            return HealthCheckResult.Healthy();
        }
        catch (Exception ex)
        {
            return HealthCheckResult.Unhealthy(exception: ex);
        }
    }
}

```

The preceding health check implementation:

- Implements the `IHealthCheck` interface.
- Accepts the `MailKitClientFactory` as a primary constructor parameter.
- Satisfies the `CheckHealthAsync` method by:
 - Attempting to get an `ISmtpClient` instance from the `factory`. If successful, it returns `HealthCheckResult.Healthy`.
 - If an exception is thrown, it returns `HealthCheckResult.Unhealthy`.

As previously shared in the registration of the `MailKitClientFactory`, the `MailKitHealthCheck` is conditionally registered with the `IHealthChecksBuilder`:

```

C#

if (settings.DisableHealthChecks is false)
{
    builder.Services.AddHealthChecks()
        .AddCheck<MailKitHealthCheck>(
            name: serviceKey is null ? "MailKit" :
            $"MailKit_{connectionName}",
            failureStatus: default,
            tags: []);
}

```

The consumer could choose to omit health checks by setting the `DisableHealthChecks` property to `true` in the configuration. A common pattern for integrations is to have optional features and .NET Aspire integrates strongly encourages these types of configurations. For more information on health checks and a working sample that includes a user interface, see [.NET Aspire ASP.NET Core HealthChecksUI sample](#).

Wire up telemetry

As a best practice, the [MailKit client library exposes telemetry](#). .NET Aspire can take advantage of this telemetry and display it in the [.NET Aspire dashboard](#). Depending on whether or not tracing and metrics are enabled, telemetry is wired up as shown in the following code snippet:

C#

```
if (settings.DisableTracing is false)
{
    builder.Services.AddOpenTelemetry()
        .WithTracing(
            traceBuilder => traceBuilder.AddSource(
                Telemetry.SmtpClient.ActivitySourceName));
}

if (settings.DisableMetrics is false)
{
    // Required by MailKit to enable metrics
    Telemetry.SmtpClient.Configure();

    builder.Services.AddOpenTelemetry()
        .WithMetrics(
            metricsBuilder => metricsBuilder.AddMeter(
                Telemetry.SmtpClient.MeterName));
}
```

Update the Newsletter service

With the integration library created, you can now update the Newsletter service to use the MailKit client. The first step is to add a reference to the `MailKit.Client` project. Add the `MailKit.Client.csproj` project reference to the `MailDevResource.NewsletterService` project:

.NET CLI

```
dotnet add
./MailDevResource.NewsletterService/MailDevResource.NewsletterService.csproj
```

```
reference MailKit.Client/MailKit.Client.csproj
```

Next, add a reference to the `ServiceDefaults` project:

.NET CLI

```
dotnet add
./MailDevResource.NewsletterService/MailDevResource.NewsletterService.csproj
reference
MailDevResource.ServiceDefaults/MailDevResource.ServiceDefaults.csproj
```

The final step is to replace the existing `Program.cs` file in the `MailDevResource.NewsletterService` project with the following C# code:

C#

```
using System.Net.Mail;
using MailKit.Client;
using MailKit.Net.Smtp;
using MimeKit;

var builder = WebApplication.CreateBuilder(args);

builder.AddServiceDefaults();

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

// Add services to the container.
builder.AddMailKitClient("maildev");

var app = builder.Build();

app.MapDefaultEndpoints();

// Configure the HTTP request pipeline.

app.UseSwagger();
app.UseSwaggerUI();
app.UseHttpsRedirection();

app.MapPost("/subscribe",
    async (MailKitClientFactory factory, string email) =>
{
    ISmtpClient client = await factory.GetSmtpClientAsync();

    using var message = new MailMessage("newsletter@yourcompany.com", email)
    {
        Subject = "Welcome to our newsletter!",
        Body = "Thank you for subscribing to our newsletter!"
    };
}
```

```

    await client.SendAsync(MimeMessage.CreateFromMailMessage(message));
});

app.MapPost("/unsubscribe",
    async (MailKitClientFactory factory, string email) =>
{
    ISmtpClient client = await factory.GetSmtpClientAsync();

    using var message = new MailMessage("newsletter@yourcompany.com", email)
    {
        Subject = "You are unsubscribed from our newsletter!",
        Body = "Sorry to see you go. We hope you will come back soon!"
    };

    await client.SendAsync(MimeMessage.CreateFromMailMessage(message));
});

app.Run();

```

The most notable changes in the preceding code are:

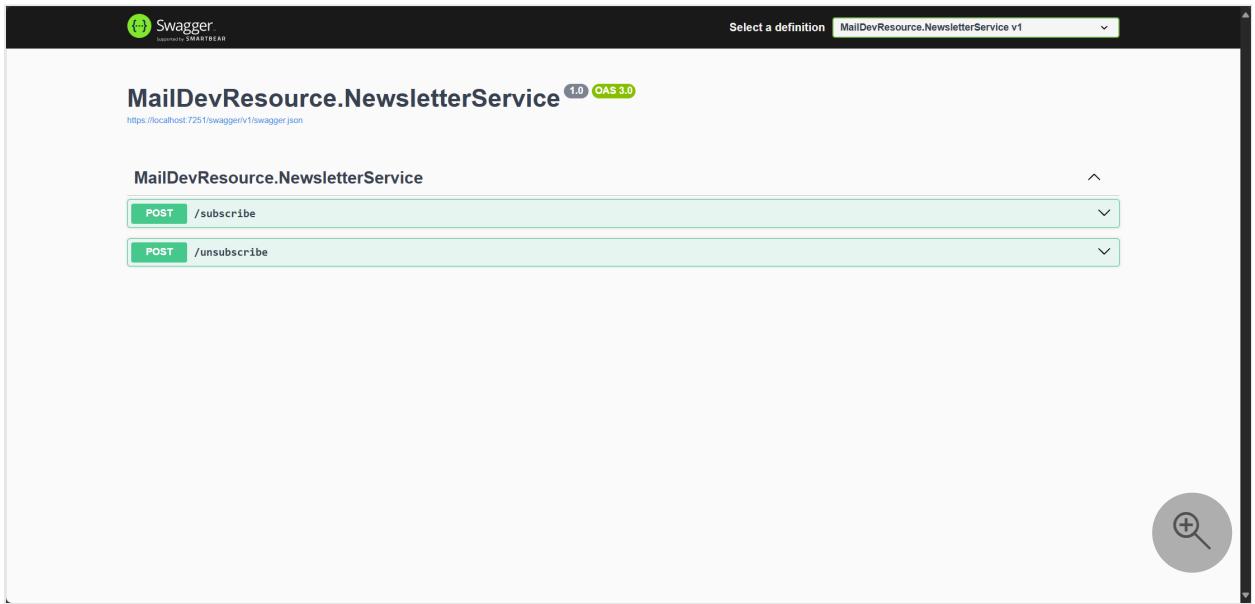
- The updated `using` statements that include the `MailKit.Client`, `MailKit.Net.Smtp`, and `MimeKit` namespaces.
- The replacement of the registration for the official .NET `SmtpClient` with the call to the `AddMailkitClient` extension method.
- The replacement of both `/subscribe` and `/unsubscribe` map post calls to instead inject the `MailKitClientFactory` and use the `ISmtpClient` instance to send the email.

Run the sample

Now that you've created the MailKit client integration and updated the Newsletter service to use it, you can run the sample. From your IDE, select `F5` or run `dotnet run` from the root directory of the solution to start the application—you should see the [.NET Aspire dashboard](#):

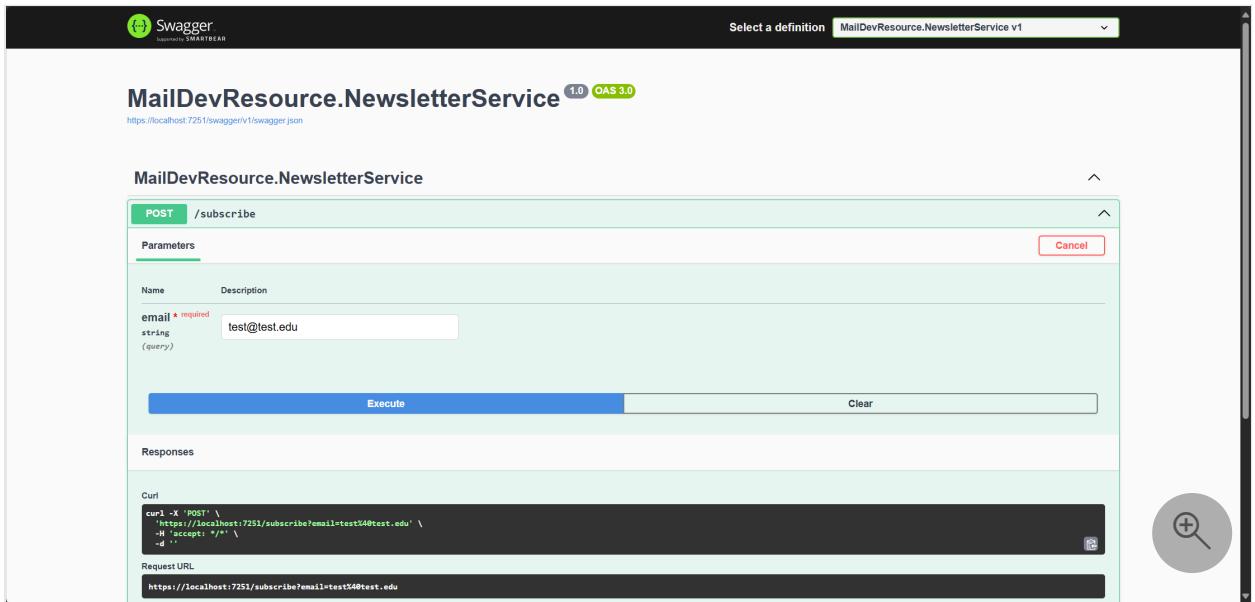
Type	Name	State	Start time	Source	Endpoints	Logs	Details
Container	maildev	Running	1:10:42 PM	docker.io/maildev/maildev:2.0.2	http://localhost:54052 [+1]	View	View
Project	newsletterservice	Running	1:10:42 PM	MailDevResource.NewsletterS...	https://localhost:7251/swagger [+1]	View	View

Once the application is running, navigate to the Swagger UI at <https://localhost:7251/swagger> and test the `/subscribe` and `/unsubscribe` endpoints. Select the down arrow to expand the endpoint:



The screenshot shows the Swagger UI interface for the `MailDevResource.NewsletterService`. At the top, there's a navigation bar with the Swagger logo and the text "Supported by SMARTBEAR". A dropdown menu "Select a definition" is set to "MailDevResource.NewsletterService v1". Below the header, the service name is displayed along with its version (1.0) and OAS 3.0 compliance. Two POST methods are listed under the "MailDevResource.NewsletterService" section: `/subscribe` and `/unsubscribe`. Each method has a green button labeled "POST" and its corresponding URL. To the right of the methods is a search icon.

Then select the `Try it out` button. Enter an email address, and then select the `Execute` button.



The screenshot shows the expanded view of the `/subscribe` endpoint. It includes a "Parameters" section with a table for the `email` parameter, which is marked as required and has a type of string. The value `test@test.edu` is entered. Below the parameters are two buttons: "Execute" (highlighted in blue) and "Clear". Under the "Responses" section, a "Curl" block contains a sample cURL command to execute the request. The "Request URL" field below it shows the full URL: `https://localhost:7251/subscribe?email=test%40test.edu`.

Repeat this several times, to add multiple email addresses. You should see the email sent to the MailDev inbox:

The screenshot shows the MailDev application interface. On the left, there's a sidebar with a search bar and four received emails from 'Welcome to our newsletter!' to various test addresses. The right side displays a message about receiving emails on port 1025, links for configuration and GitHub, and the application version (2.0.2).

Welcome to our newsletter! To: test@test.edu
2024-07-16 10:25:49 (-0500)

Welcome to our newsletter! To: test@test.dev
2024-07-16 10:25:44 (-0500)

Welcome to our newsletter! To: test@test.com
2024-07-16 10:25:40 (-0500)

Welcome to our newsletter! To: test@test.org
2024-07-16 10:25:15 (-0500)

Now receiving all emails on port 1025

For information on how to send emails from your application, please refer to the "[Configure your project](#)" guide.

For questions, feature requests or to report issues visit the [GitHub repository](#).

You are running MailDev 2.0.2

Stop the application by selecting **ctrl + c** in the terminal window where the application is running, or by selecting the stop button in your IDE.

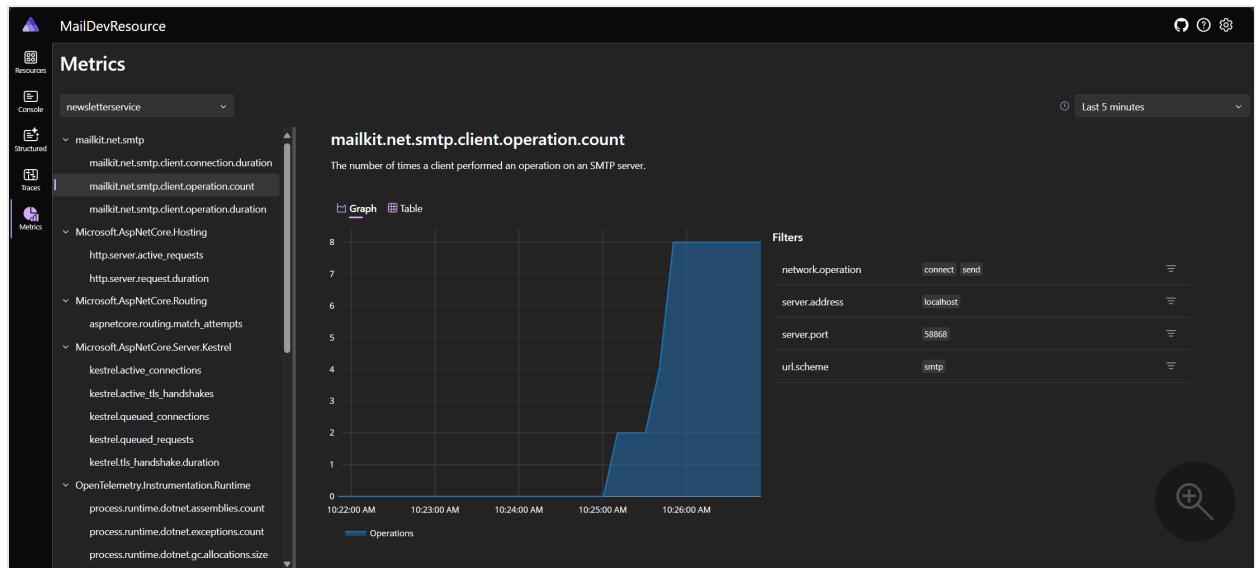
View MailKit telemetry

The MailKit client library exposes telemetry that can be viewed in the .NET Aspire dashboard. To view the telemetry, navigate to the .NET Aspire dashboard at <https://localhost:7251>. Select the `newsletter` resource to view the telemetry on the **Metrics** page:

The screenshot shows the .NET Aspire Metrics dashboard for the 'newsletter' resource. The left sidebar lists resources like 'Console', 'Structured', 'Traces', and 'Metrics'. Under 'Metrics', it shows a tree view with 'mailkit.net.smtp' expanded, showing three metrics: 'mailkit.net.smtp.client.connection', 'mailkit.net.smtp.client.operation.count', and 'mailkit.net.smtp.client.operation.duration'. The main panel displays these metrics with their descriptions. A search icon is visible in the bottom right corner.

Instrument	Description
mailkit.net.smtp.client.connection.duration	The duration of successfully established connections to an SMTP server.
mailkit.net.smtp.client.operation.count	The number of times a client performed an operation on an SMTP server.
mailkit.net.smtp.client.operation.duration	The amount of time it takes for the SMTP server to perform an operation.

Open up the Swagger UI again, and make some requests to the `/subscribe` and `/unsubscribe` endpoints. Then, navigate back to the .NET Aspire dashboard and select the `newsletter` resource. Select a metric under the `mailkit.net.smtp` node, such as `mailkit.net.smtp.client.operation.count`. You should see the telemetry for the MailKit client:



Summary

In this article, you learned how to create a .NET Aspire integration that uses MailKit to send emails. You also learned how to integrate this integration into the Newsletter app you previously built. You learned about the core principles of .NET Aspire integrations, such as exposing the underlying client library to consumers through dependency injection, and how to add health checks and telemetry to the integration. You also learned how to update the Newsletter service to use the MailKit client.

Go forth and build your own .NET Aspire integrations. If you believe that there's enough community value in the integration you're building, consider publishing it as a [NuGet package](#) for others to use. Furthermore, consider submitting a pull request to the [.NET Aspire GitHub repository](#) ↗ for consideration to be included in the official .NET Aspire integrations.

Next steps

[Secure communication between hosting and client integrations](#)

Secure communication between hosting and client integrations

Article • 09/24/2024

This article is a continuation of two previous articles demonstrating the creation of [custom hosting integrations](#) and [custom client integrations](#).

One of the primary benefits to .NET Aspire is how it simplifies the configurability of resources and consuming clients (or integrations). This article demonstrates how to share authentication credentials from a custom resource in a hosting integration, to the consuming client in a custom client integration. The custom resource is a MailDev container that allows for either incoming or outgoing credentials. The custom client integration is a MailKit client that sends emails.

Prerequisites

Since this article continues from previous content, you should have already created the resulting solution as a starting point for this article. If you haven't already, complete the following articles:

- [Create custom .NET Aspire hosting integrations](#)
- [Create custom .NET Aspire client integrations](#)

The resulting solution from these previous articles contains the following projects:

- *MailDev.Hosting*: Contains the custom resource type for the MailDev container.
- *MailDevResource.AppHost*: The [app host](#) that uses the custom resource and defines it as a dependency for a Newsletter service.
- *MailDevResource.NewsletterService*: An ASP.NET Core Web API project that sends emails using the MailDev container.
- *MailDevResource.ServiceDefaults*: Contains the [default service configurations](#) intended for sharing.
- *MailKit.Client*: Contains the custom client integration that exposes the MailKit `SmtpClient` through a factory.

Update the MailDev resource

To flow authentication credentials from the MailDev resource to the MailKit integration, you need to update the MailDev resource to include the username and password parameters.

The MailDev container supports basic authentication for both incoming and outgoing simple mail transfer protocol (SMTP). To configure the credentials for incoming, you need to set the `MAILDEV_INCOMING_USER` and `MAILDEV_INCOMING_PASS` environment variables. For more information, see [MailDev: Usage ↗](#). Update the `MailDevResource.cs` file in the `MailDev.Hosting` project, by replacing its contents with the following C# code:

C#

```
// For ease of discovery, resource types should be placed in
// the Aspire.Hosting.ApplicationModel namespace. If there is
// likelihood of a conflict on the resource name consider using
// an alternative namespace.
namespace Aspire.Hosting.ApplicationModel;

public sealed class MailDevResource(
    string name,
    ParameterResource? username,
    ParameterResource password)
    : ContainerResource(name), IResourceWithConnectionString
{
    // Constants used to refer to well known-endpoint names, this is
    // specific
    // for each resource type. MailDev exposes an SMTP and HTTP endpoints.
    internal const string SmtpEndpointName = "smtp";
    internal const string HttpEndpointName = "http";

    private const string DefaultUsername = "mail-dev";

    // An EndpointReference is a core .NET Aspire type used for keeping
    // track of endpoint details in expressions. Simple literal values
    // cannot
    // be used because endpoints are not known until containers are
    // launched.
    private EndpointReference? _smtpReference;

    /// <summary>
    /// Gets the parameter that contains the MailDev SMTP server username.
    /// </summary>
    public ParameterResource? UsernameParameter { get; } = username;

    internal ReferenceExpression UserNameReference =>
        UsernameParameter is not null ?
            ReferenceExpression.Create($"{UsernameParameter}") :
            ReferenceExpression.Create($"{DefaultUsername}");

    /// <summary>
    /// Gets the parameter that contains the MailDev SMTP server password.
    /// </summary>
    public ParameterResource PasswordParameter { get; } = password;

    public EndpointReference SmtpEndpoint =>
        _smtpReference ??= new(this, SmtpEndpointName);
```

```

    // Required property on IResourceWithConnectionString. Represents a
    connection
    // string that applications can use to access the MailDev server. In
    this case
    // the connection string is composed of the SmtpEndpoint endpoint
    reference.
    public ReferenceExpression ConnectionStringExpression =>
        ReferenceExpression.Create(
            $"Endpoint=smtp://{{SmtpEndpoint.Property(EndpointProperty.Host)}}:
            {{SmtpEndpoint.Property(EndpointProperty.Port)}};Username=
            {UserNameReference};Password={PasswordParameter}"
        );
}

```

These updates add a `UserNameParameter` and `PasswordParameter` property. These properties are used to store the parameters for the MailDev username and password. The `ConnectionStringExpression` property is updated to include the username and password parameters in the connection string. Next, update the `MailDevResourceBuilderExtensions.cs` file in the `MailDev.Hosting` project with the following C# code:

C#

```

using Aspire.Hosting.ApplicationModel;

// Put extensions in the Aspire.Hosting namespace to ease discovery as
referencing
// the .NET Aspire hosting package automatically adds this namespace.
namespace Aspire.Hosting;

public static class MailDevResourceBuilderExtensions
{
    private const string UserEnvVarName = "MAILDEV_INCOMING_USER";
    private const string PasswordEnvVarName = "MAILDEV_INCOMING_PASS";

    /// <summary>
    /// Adds the <see cref="MailDevResource"/> to the given
    /// <paramref name="builder"/> instance. Uses the "2.0.2" tag.
    /// </summary>
    /// <param name="builder">The <see
    cref="IDistributedApplicationBuilder"/>.</param>
    /// <param name="name">The name of the resource.</param>
    /// <param name="httpPort">The HTTP port.</param>
    /// <param name="smtpPort">The SMTP port.</param>
    /// <returns>
    /// An <see cref="IResourceBuilder<MailDevResource>"/> instance that
    /// represents the added MailDev resource.
    /// </returns>
    public static IResourceBuilder<MailDevResource> AddMailDev(

```

```

        this IDistributedApplicationBuilder builder,
        string name,
        int? httpPort = null,
        int? smtpPort = null,
        IResourceBuilder<ParameterResource>? userName = null,
        IResourceBuilder<ParameterResource>? password = null)
    {
        var passwordParameter = password?.Resource ??

ParameterResourceBuilderExtensions.CreateDefaultPasswordParameter(
    builder, $"{name}-password");

        // The AddResource method is a core API within .NET Aspire and is
        // used by resource developers to wrap a custom resource in an
        // IResourceBuilder<T> instance. Extension methods to customize
        // the resource (if any exist) target the builder interface.
        var resource = new MailDevResource(
            name, userName?.Resource, passwordParameter);

        return builder.AddResource(resource)
            .WithImage(MailDevContainerImageTags.Image)
            .WithImageRegistry(MailDevContainerImageTags.Registry)
            .WithImageTag(MailDevContainerImageTags.Tag)
            .WithHttpEndpoint(
                targetPort: 1080,
                port: httpPort,
                name: MailDevResource.HttpEndpointName)
            .WithEndpoint(
                targetPort: 1025,
                port: smtpPort,
                name: MailDevResource.SmtpEndpointName)
            .WithEnvironment(context =>
{
    context.EnvironmentVariables[UserEnvVarName] =
resource.UserNameReference;
    context.EnvironmentVariables[PasswordEnvVarName] =
resource.PasswordParameter;
});
    }
}

// This class just contains constant strings that can be updated
// periodically
// when new versions of the underlying container are released.
internal static class MailDevContainerImageTags
{
    internal const string Registry = "docker.io";

    internal const string Image = "maildev/maildev";

    internal const string Tag = "2.0.2";
}

```

The preceding code updates the `AddMailDev` extension method to include the `userName` and `password` parameters. The `WithEnvironment` method is updated to include the `UserEnvVarName` and `PasswordEnvVarName` environment variables. These environment variables are used to set the MailDev username and password.

Update the app host

Now that the resource is updated to include the username and password parameters, you need to update the app host to include these parameters. Update the `Program.cs` file in the `MailDevResource.AppHost` project with the following C# code:

C#

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

var mailDevUsername = builder.AddParameter("maildev-username");
var mailDevPassword = builder.AddParameter("maildev-password");

var maildev = builder.AddMailDev(
    name: "maildev",
    userName: mailDevUsername,
    password: mailDevPassword);

builder.AddProject<Projects.MailDevResource_NewsletterService>
("newsletterservice")
    .WithReference(maildev);

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

The preceding code adds two parameters for the MailDev username and password. It assigns these parameters to the `MAILDEV_INCOMING_USER` and `MAILDEV_INCOMING_PASS` environment variables. The `AddMailDev` method has two chained calls to `WithEnvironment` which includes these environment variables. For more information on parameters, see [External parameters](#).

Next, configure the secrets for these parameters. Right-click on the `MailDevResource.AppHost` project and select `Manage User Secrets`. Add the following JSON to the `secrets.json` file:

JSON

```
{
  "Parameters:maildev-username": "@admin",
  "Parameters:maildev-password": "t3st1ng"
}
```

⚠ Warning

These credentials are for demonstration purposes only and MailDev is intended for local development. These credentials are fictitious and shouldn't be used in a production environment.

Update the MailKit integration

It's good practice for client integrations to expect connection strings to contain various key/value pairs, and to parse these pairs into the appropriate properties. Update the `MailKitClientSettings.cs` file in the `MailKit.Client` project with the following C# code:

C#

```
using System.Data.Common;
using System.Net;

namespace MailKit.Client;

/// <summary>
/// Provides the client configuration settings for connecting MailKit to an
/// SMTP server.
/// </summary>
public sealed class MailKitClientSettings
{
    internal const string DefaultConfigSectionName = "MailKit:Client";

    /// <summary>
    /// Gets or sets the SMTP server <see cref="Uri"/>.
    /// </summary>
    /// <value>
    /// The default value is <see langword="null"/>.
    /// </value>
    public Uri? Endpoint { get; set; }

    /// <summary>
    /// Gets or sets the network credentials that are optionally
    /// configurable for SMTP
    /// server's that require authentication.
    /// </summary>
    /// <value>
    /// The default value is <see langword="null"/>.
    /// </value>
    public NetworkCredential? Credentials { get; set; }

    /// <summary>
    /// Gets or sets a boolean value that indicates whether the database
    /// health check is disabled or not.
    /// </summary>
```



```

        if (Uri.TryCreate(endpoint.ToString(), UriKind.Absolute, out
uri) is false)
{
    throw new InvalidOperationException($""""
        The 'ConnectionStrings:<connectionName>' (or
'Endpoint' key in
        '{DefaultConfigSectionName}') isn't a valid URI.
""");
}

Endpoint = uri;

if (builder.TryGetValue("Username", out var username) &&
    builder.TryGetValue("Password", out var password))
{
    Credentials = new(
        username.ToString(), password.ToString());
}
}
}
}

```

The preceding settings class, now includes a `Credentials` property of type `NetworkCredential`. The `ParseConnectionString` method is updated to parse the `Username` and `Password` keys from the connection string. If the `Username` and `Password` keys are present, a `NetworkCredential` is created and assigned to the `Credentials` property.

With the settings class updated to understand and populate the credentials, update the factory to conditionally use the credentials if they're configured. Update the `MailKitClientFactory.cs` file in the `MailKit.Client` project with the following C# code:

C#

```

using System.Net;
using MailKit.Net.Smtp;

namespace MailKit.Client;

/// <summary>
/// A factory for creating <see cref="ISmtpClient"/> instances
/// given a <paramref name="smtpUri"/> (and optional <paramref
/// name="credentials"/>).
/// </summary>
/// <param name="settings">
/// The <see cref="MailKitClientSettings"/> settings for the SMTP server
/// </param>
public sealed class MailKitClientFactory(MailKitClientSettings settings) :
IDisposable

```

```
{  
    private readonly SemaphoreSlim _semaphore = new(1, 1);  
  
    private SmtpClient? _client;  
  
    /// <summary>  
    /// Gets an <see cref="ISsmtpClient"/> instance in the connected state  
    /// (and that's been authenticated if configured).  
    /// </summary>  
    /// <param name="cancellationToken">Used to abort client creation and  
    /// connection.</param>  
    /// <returns>A connected (and authenticated) <see cref="ISsmtpClient"/>  
    /// instance.</returns>  
    /// <remarks>  
    /// Since both the connection and authentication are considered  
    /// expensive operations,  
    /// the <see cref="ISsmtpClient"/> returned is intended to be used for  
    /// the duration of a request  
    /// (registered as 'Scoped') and is automatically disposed of.  
    /// </remarks>  
    public async Task<ISsmtpClient> GetSmtpClientAsync(  
        CancellationToken cancellationToken = default)  
{  
    await _semaphore.WaitAsync(cancellationToken);  
  
    try  
    {  
        if (_client is null)  
        {  
            _client = new SmtpClient();  
  
            await _client.ConnectAsync(settings.Endpoint,  
            cancellationToken)  
                .ConfigureAwait(false);  
  
            if (settings.Credentials is not null)  
            {  
                await _client.AuthenticateAsync(settings.Credentials,  
                cancellationToken)  
                    .ConfigureAwait(false);  
            }  
        }  
    }  
    finally  
    {  
        _semaphore.Release();  
    }  
  
    return _client;  
}  
  
public void Dispose()  
{  
    _client?.Dispose();  
    _semaphore.Dispose();  
}
```

```
    }  
}
```

When the factory determines that credentials have been configured, it authenticates with the SMTP server after connecting before returning the `SmtpClient`.

Run the sample

Now that you've updated the resource, corresponding integration projects, and the app host, you're ready to run the sample app. To run the sample from your IDE, select `F5` or use `dotnet run` from the root directory of the solution to start the application—you should see the [.NET Aspire dashboard](#). Navigate to the `maildev` container resource and view the details. You should see the username and password parameters in the resource details, under the **Environment Variables** section:

The screenshot shows the .NET Aspire dashboard interface. On the left, there's a sidebar with icons for Console, Structured, Traces, and Metrics. The main area is titled "MailDevResource". Under "Resources", there's a table with columns: Type, Name, State, Start time, Source, Endpoints, Logs, and Details. A red arrow points to the "maildev" row, which is a Container and is currently Running. A red box surrounds the "View" button in the Details column for this row. Below this, another row shows a Project named "newsletterservice" also running. Under the "Container: maildev" heading, there are sections for "Resource" (with a dropdown menu) and "Endpoints". The "Endpoints" table lists: smtp (tcp://localhost:53367), smtp target port (tcp://127.0.0.1:53375), http (http://localhost:53366), and http target port (http://127.0.0.1:53376). At the bottom, there's a "Environment variables" section with a table. A red box highlights the first two rows: "MAILDEV_INCOMING_PASS" with value "t3st1ng" and "MAILDEV_INCOMING_USER" with value "@admin". There are also search and refresh icons on the right side of this section.

Likewise, you should see the connection string in the `newsletterservice` resource details, under the **Environment Variables** section:

The screenshot shows the MailDevResource interface. On the left, there are navigation tabs: Resources, Console, Structured, Traces, and Metrics. The 'Resources' tab is selected. In the main area, there is a table with columns: Type, Name, State, Start time, Source, Endpoints, Logs, and Details. There are two rows: one for a 'Container' named 'maildev' (Running, 12:37:57 PM, docker.io/maildev/maildev:2..., http://localhost:53366, View) and one for a 'Project' named 'newsletterservice' (Running, 12:37:57 PM, MailDevResource.Newsletter..., https://localhost:7251/swagger, View). A red arrow points from the 'maildev' row to the 'newsletterservice' row. The 'newsletterservice' row has a red box around its 'View' button. Below the table, a modal window titled 'Project: newsletterservice' is open. It has tabs for 'Resource', 'Endpoints', and 'Environment variables'. Under 'Environment variables', there is a table with columns: Name and Value. The table contains several entries, including 'ConnectionString_maildev' which has a red box around it. The value for 'ConnectionString_maildev' is 'Endpoint=smtp://localhost:53367;Username=@admin;Password=t3st1ng'. There are also other entries like 'ASPNETCORE_ENVIRONMENT', 'ASPNETCORE_HTTPS_PORT', 'ASPNETCORE_URLS', 'DOTNET_SYSTEM_CONSOLE_ALLOW_ANSI_COLOR_REDIRECTION', 'LOGGING_CONSOLE_FORMATTERNAME', and 'LOGGING_CONSOLE_FORMATTEROPTIONS_TIMESTAMPFORM...'. Each entry has a red icon next to it.

Validate that everything is working as expected.

Summary

This article demonstrated how to flow authentication credentials from a custom resource to a custom client integration. The custom resource is a MailDev container that allows for either incoming or outgoing credentials. The custom client integration is a MailKit client that sends emails. By updating the resource to include the `username` and `password` parameters, and updating the integration to parse and use these parameters, authentication flows credentials from the hosting integration to the client integration.

.NET Aspire deployments

Article • 06/15/2024

.NET Aspire projects are built with cloud-agnostic principles, allowing deployment flexibility across various platforms supporting .NET and containers. Users can adapt the provided guidelines for deployment on other cloud environments or local hosting. The manual deployment process, while feasible, involves exhaustive steps prone to errors. Users prefer leveraging CI/CD pipelines and cloud-specific tooling for a more streamlined deployment experience tailored to their chosen infrastructure.

Deployment manifest

To enable deployment tools from Microsoft and other cloud providers to understand the structure of .NET Aspire projects, specialized targets of the [AppHost project](#) can be executed to generate a manifest file describing the projects/services used by the app and the properties necessary for deployment, such as environment variables.

For more information on the schema of the manifest and how to run app host project targets, see [.NET Aspire manifest format for deployment tool builders](#).

Deploy to Azure

.NET Aspire enables deployment to Azure Container Apps. The number of environments .NET Aspire can deploy to will grow over time.

Azure Container Apps

.NET Aspire projects are designed to run in containerized environments. Azure Container Apps is a fully managed environment that enables you to run microservices and containerized applications on a serverless platform. The [Azure Container Apps](#) topic describes how to deploy Aspire apps to ACA manually, using bicep, or using the Azure Developer CLI (azd).

Use Application Insights for .NET Aspire telemetry

.NET Aspire projects are designed to emit telemetry using OpenTelemetry which uses a provider model. .NET Aspire projects can direct their telemetry to Azure Monitor / Application Insights using the Azure Monitor telemetry distro. For more information, see [Use Application Insights for .NET Aspire telemetry](#) for step-by-step instructions.

Deploy to Kubernetes

Kubernetes is a popular container orchestration platform that can run .NET Aspire projects. To deploy .NET Aspire projects to Kubernetes clusters, you need to map the .NET Aspire JSON manifest to a Kubernetes YAML manifest file. There are two ways to do this: by using the Aspir8 project, or by manually creating Kubernetes manifests.

The Aspir8 project

Aspir8, an open-source project, handles the generation of deployment YAML based on the .NET Aspire app host manifest. The project outputs a .NET global tool that can be used to perform a series of tasks, resulting in the generation of Kubernetes manifests:

- `aspire init`: Initializes the Aspir8 project in the current directory.
- `aspire generate`: Generates Kubernetes manifests based on the .NET Aspire app host manifest.
- `aspire apply`: Applies the generated Kubernetes manifests to the Kubernetes cluster.
- `aspire destroy`: Deletes the resources created by the `apply` command.

With these commands, you can build your apps, containerize them, and deploy them to Kubernetes clusters. For more information, see [Aspir8 ↗](#).

Manually create Kubernetes manifests

Alternatively, the Kubernetes manifests can be created manually. This involves more effort and is more time consuming. For more information, see [Deploy a .NET microservice to Kubernetes](#).

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Deploy a .NET Aspire project to Azure Container Apps

Article • 06/15/2024

.NET Aspire projects are designed to run in containerized environments. Azure Container Apps is a fully managed environment that enables you to run microservices and containerized applications on a serverless platform. This article will walk you through creating a new .NET Aspire solution and deploying it to Microsoft Azure Container Apps using the Azure Developer CLI (`azd`). You'll learn how to complete the following tasks:

- ✓ Provision an Azure resource group and Container Registry
- ✓ Publish the .NET Aspire projects as container images in Azure Container Registry
- ✓ Provision a Redis container in Azure
- ✓ Deploy the apps to an Azure Container Apps environment
- ✓ View application console logs to troubleshoot application issues

Prerequisites

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

- [.NET 8.0](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or [the .NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

As an alternative to this tutorial and for a more in-depth guide, see [Deploy a .NET Aspire project to Azure Container Apps using azd \(in-depth guide\)](#).

Deploy .NET Aspire projects with `azd`

With .NET Aspire and Azure Container Apps (ACA), you have a great hosting scenario for building out your cloud-native apps with .NET. We built some great new features into

the Azure Developer CLI (`azd`) specific for making .NET Aspire development and deployment to Azure a friction-free experience. You can still use the Azure CLI and/or Bicep options when you need a granular level of control over your deployments. But for new projects, you won't find an easier path to success for getting a new microservice topology deployed into the cloud.

Create a .NET Aspire project

As a starting point, this article assumes that you've created a .NET Aspire project from the [.NET Aspire Starter Application](#) template. For more information, see [Quickstart: Build your first .NET Aspire project](#).

Resource naming

When you create new Azure resources, it's important to follow the naming requirements. For Azure Container Apps, the name must be 2-32 characters long and consist of lowercase letters, numbers, and hyphens. The name must start with a letter and end with an alphanumeric character.

For more information, see [Naming rules and restrictions for Azure resources](#).

Install the Azure Developer CLI

The process for installing `azd` varies based on your operating system, but it is widely available via `winget`, `brew`, `apt`, or directly via `curl`. To install `azd`, see [Install Azure Developer CLI](#).

Initialize the template

1. Open a new terminal window and `cd` into the *AppHost* project directory of your .NET Aspire solution.
2. Execute the `azd init` command to initialize your project with `azd`, which will inspect the local directory structure and determine the type of app.

```
Azure Developer CLI
```

```
azd init
```

For more information on the `azd init` command, see [azd init](#).

3. Select **Use code in the current directory** when `azd` prompts you with two app initialization options.

Output

```
? How do you want to initialize your app? [Use arrows to move, type to filter]
> Use code in the current directory
    Select a template
```

4. After scanning the directory, `azd` prompts you to confirm that it found the correct .NET Aspire *AppHost* project. Select the **Confirm and continue initializing my app** option.

Output

```
Detected services:
```

```
.NET (Aspire)
Detected in:
D:\source\repos\AspireSample\AspireSample.AppHost\AspireSample.AppHost.csproj
```

```
azd will generate the files necessary to host your app on Azure using
Azure Container Apps.
```

```
? Select an option [Use arrows to move, type to filter]
> Confirm and continue initializing my app
    Cancel and exit
```

5. Enter an environment name, which is used to name provisioned resources in Azure and managing different environments such as `dev` and `prod`.

Output

```
Generating files to run your app on Azure:
```

```
(✓) Done: Generating ./azure.yaml
(✓) Done: Generating ./next-steps.md
```

```
SUCCESS: Your app is ready for the cloud!
```

```
You can provision and deploy your app to Azure by running the azd up
command in this directory. For more information on configuring your
app, see ./next-steps.md
```

`azd` generates a number of files and places them into the working directory. These files are:

- *azure.yaml*: Describes the services of the app, such as .NET Aspire AppHost project, and maps them to Azure resources.
- *.azure/config.json*: Configuration file that informs `azd` what the current active environment is.
- *.azure/aspireazddev/.env*: Contains environment specific overrides.

Deploy the template

1. Once an `azd` template is initialized, the provisioning and deployment process can be executed as a single command from the *AppHost* project directory using `azd up`:

Azure Developer CLI

```
azd up
```

2. Select the subscription you'd like to deploy to from the list of available options:

Output

```
Select an Azure Subscription to use: [Use arrows to move, type to filter]
  1. SampleSubscription01 (xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxx)
  2. SamepleSubscription02 (xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxx)
```

3. Select the desired Azure location to use from the list of available options:

Output

```
Select an Azure location to use: [Use arrows to move, type to filter]
  42. (US) Central US (centralus)
  43. (US) East US (eastus)
> 44. (US) East US 2 (eastus2)
  46. (US) North Central US (northcentralus)
  47. (US) South Central US (southcentralus)
```

After you make your selections, `azd` executes the provisioning and deployment process.

Output

By default, a service can only be reached from inside the Azure Container Apps environment it is running in. Selecting a service here will also allow it to be reached from the Internet.

? Select which services to expose to the Internet webfrontend

? Select an Azure Subscription to use: 1. <YOUR SUBSCRIPTION>

? Select an Azure location to use: 1. <YOUR LOCATION>

Packaging services (azd package)

SUCCESS: Your application was packaged for Azure in less than a second.

Provisioning Azure resources (azd provision)

Provisioning Azure resources can take some time.

Subscription: <YOUR SUBSCRIPTION>

Location: <YOUR LOCATION>

You can view detailed progress in the Azure Portal:

<LINK TO DEPLOYMENT>

(✓) Done: Resource group: <YOUR RESOURCE GROUP>

(✓) Done: Container Registry: <ID>

(✓) Done: Log Analytics workspace: <ID>

(✓) Done: Container Apps Environment: <ID>

SUCCESS: Your application was provisioned in Azure in 1 minute 13 seconds.
You can view the resources created under the resource group <YOUR RESOURCE GROUP> in Azure Portal:

<LINK TO RESOURCE GROUP OVERVIEW>

Deploying services (azd deploy)

(✓) Done: Deploying service apiservice

- Endpoint: <YOUR UNIQUE apiservice APP>.azurecontainerapps.io/

(✓) Done: Deploying service webfrontend

- Endpoint: <YOUR UNIQUE webfrontend APP>.azurecontainerapps.io/

SUCCESS: Your application was deployed to Azure in 1 minute 39 seconds.
You can view the resources created under the resource group <YOUR RESOURCE GROUP> in Azure Portal:

<LINK TO RESOURCE GROUP OVERVIEW>

SUCCESS: Your up workflow to provision and deploy to Azure completed in 3 minutes 50 seconds.

The `azd up` command acts as wrapper for the following individual `azd` commands to provision and deploy your resources in a single step:

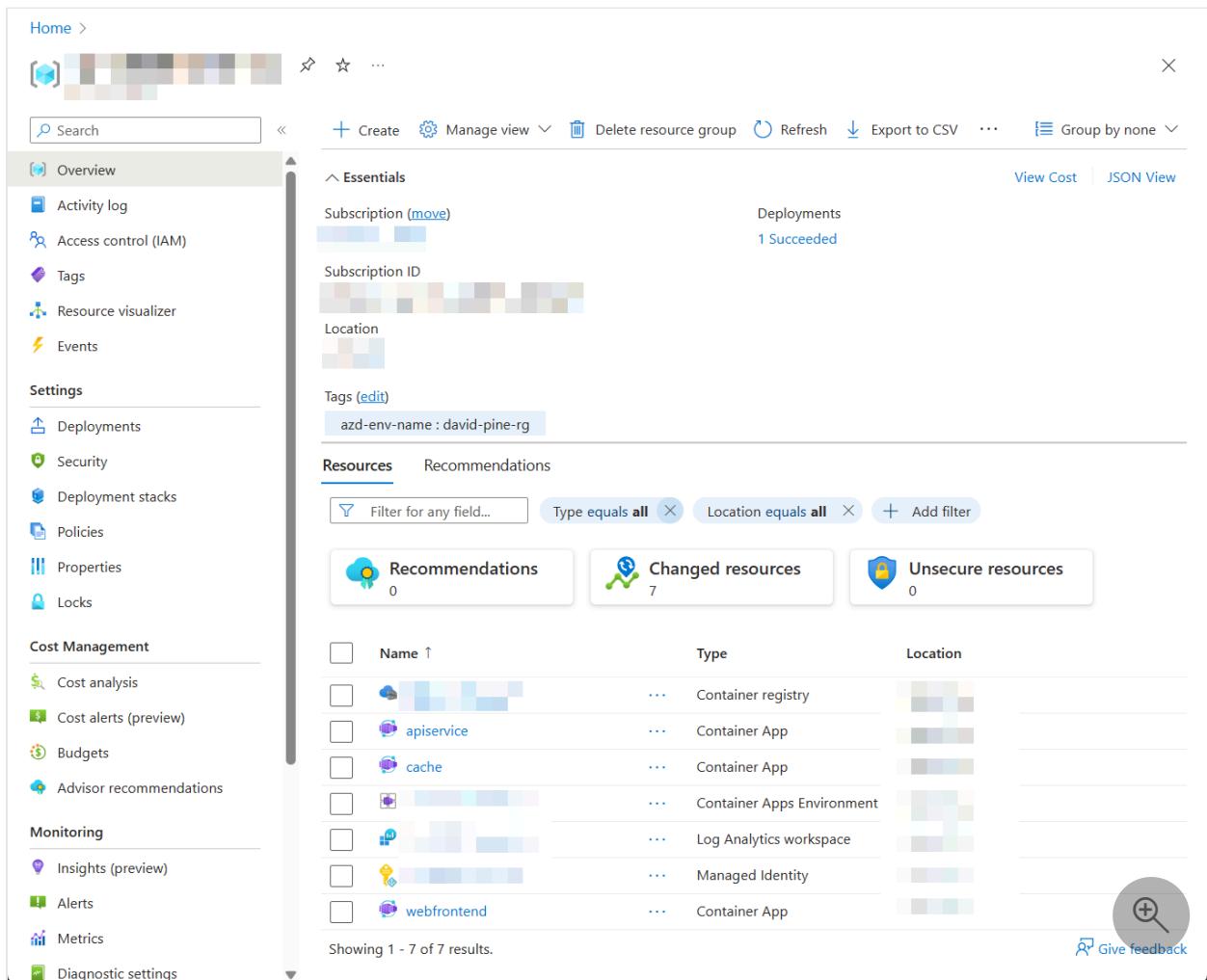
1. **azd package**: The app projects and their dependencies are packaged into containers.
2. **azd provision**: The Azure resources the app will need are provisioned.
3. **azd deploy**: The projects are pushed as containers into an Azure Container Registry instance, and then used to create new revisions of Azure Container Apps in which

the code will be hosted.

When the `azd up` stages complete, your app will be available on Azure, and you can open the Azure portal to explore the resources. `azd` also outputs URLs to access the deployed apps directly.

Test the deployed app

Now that the app has been provisioned and deployed, you can browse to the Azure portal. In the resource group where you deployed the app, you'll see the three container apps and other resources.



The screenshot shows the Azure portal's resource group overview page. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Resource visualizer, Events, Settings (Deployments, Security, Deployment stacks, Policies, Properties, Locks), Cost Management (Cost analysis, Cost alerts (preview), Budgets, Advisor recommendations), Monitoring (Insights (preview), Alerts, Metrics, Diagnostic settings), and a search bar. The main content area displays the 'Essentials' section with information about the subscription (Subscription ID, Location, Tags), deployment status (1 Succeeded), and resource filters. Below this, there are sections for Recommendations, Changed resources (7 items), and Unsecure resources (0 items). A table lists seven resources: apiservice (Container App), cache (Container App), Container Apps Environment (Container Apps Environment), Log Analytics workspace (Log Analytics workspace), Managed Identity (Managed Identity), and webfrontend (Container App). The table includes columns for Name, Type, and Location. At the bottom right, there is a 'Give feedback' button.

Click on the `web` Container App to open it up in the portal.

Application Url : <https://webfrontend.az...>

Resource group (move) :

Status :

Location (move) :

Subscription (move) :

Subscription ID :

Container Apps Environment :

Environment type :

Log Analytics :

View Cost JSON View

Click the Application URL link to open the front end in the browser.

Date	Temp. (C)	Temp. (F)	Summary
10/27/2023	20	67	Warm
10/28/2023	40	103	Scorching
10/29/2023	16	60	Balmy
10/30/2023	-11	13	Cool
10/31/2023	39	102	Bracing

When you click the "Weather" node in the navigation bar, the front end `web` container app makes a call to the `apiservice` container app to get data. The front end's output will be cached using the `redis` container app and the [.NET Aspire Redis Output Caching component](#). As you refresh the front end a few times, you'll notice that the weather data is cached. It will update after a few seconds.

Deploy the .NET Aspire Dashboard

You can deploy the .NET Aspire dashboard as part of your hosted app. This feature is currently in alpha support, so you must enable the `alpha.aspire.dashboard` feature flag. When enabled, the `azd` output logs print an additional URL to the deployed dashboard.

Azure Developer CLI

```
azd config set alpha.aspire.dashboard on
```

You can also run `azd monitor` to automatically launch the dashboard.

Azure Developer CLI

```
azd monitor
```

Clean up resources

Run the following Azure CLI command to delete the resource group when you no longer need the Azure resources you created. Deleting the resource group also deletes the resources contained inside of it.

Azure CLI

```
az group delete --name <your-resource-group-name>
```

For more information, see [Clean up resources in Azure](#).

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Deploy a .NET Aspire project to Azure Container Apps using Visual Studio

Article • 06/21/2024

.NET Aspire projects are designed to run in containerized environments. Azure Container Apps is a fully managed environment that enables you to run microservices and containerized applications on a serverless platform. This article will walk you through creating a new .NET Aspire solution and deploying it to Microsoft Azure Container Apps using the Visual Studio. You'll learn how to complete the following tasks:

- ✓ Provision an Azure resource group and Container Registry
- ✓ Publish the .NET Aspire projects as container images in Azure Container Registry
- ✓ Provision a Redis container in Azure
- ✓ Deploy the apps to an Azure Container Apps environment
- ✓ View application console logs to troubleshoot application issues

Prerequisites

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

- [.NET 8.0](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or [the .NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

Create a .NET Aspire project

As a starting point, this article assumes that you've created a .NET Aspire project from the [.NET Aspire Starter Application](#) template. For more information, see [Quickstart: Build your first .NET Aspire project](#).

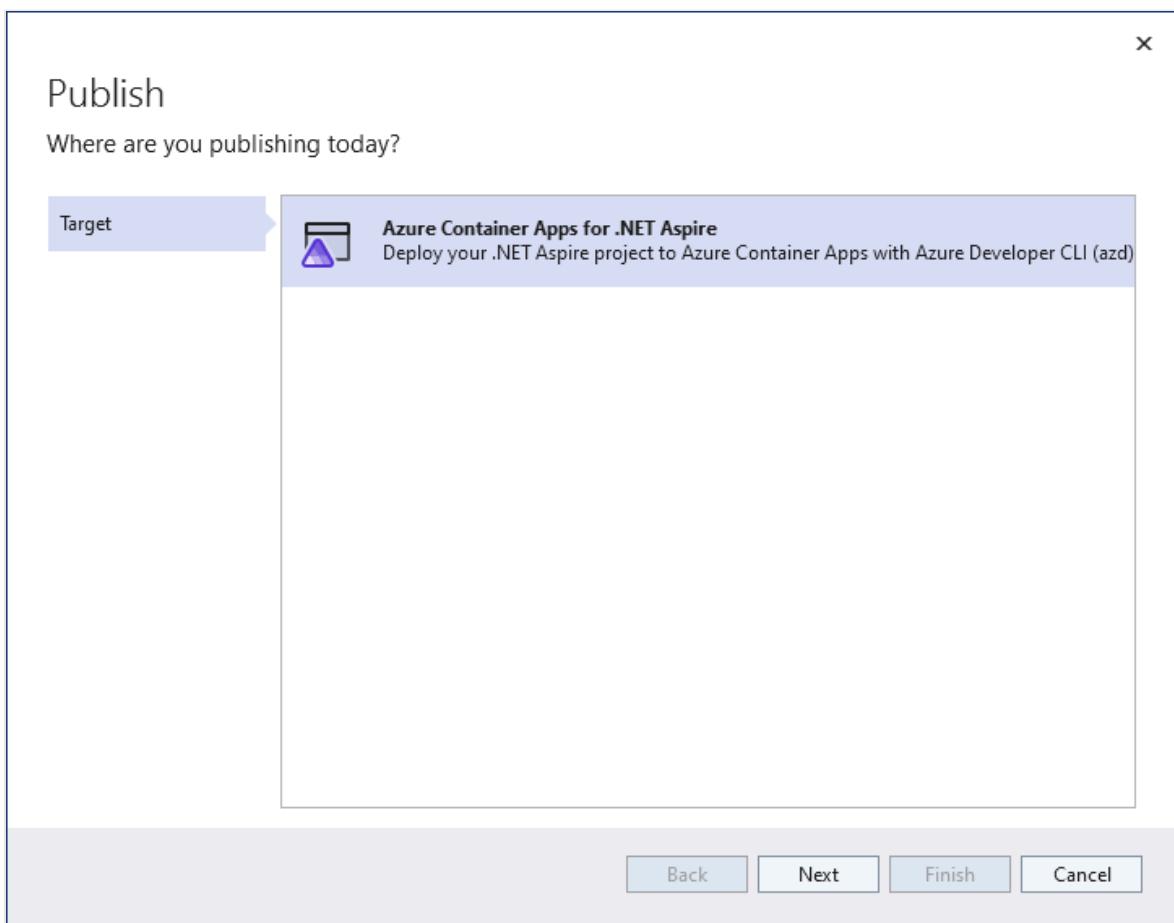
Resource naming

When you create new Azure resources, it's important to follow the naming requirements. For Azure Container Apps, the name must be 2-32 characters long and consist of lowercase letters, numbers, and hyphens. The name must start with a letter and end with an alphanumeric character.

For more information, see [Naming rules and restrictions for Azure resources](#).

Deploy the app

1. In the solution explorer, right-click on the **.AppHost** project and select **Publish** to open the **Publish** dialog.
2. Select **Azure Container Apps for .NET Aspire** as the publishing target.



3. On the **AzDev Environment** step, select your desired **Subscription** and **Location** values and then enter an **Environment name** such as *aspire-vs*. The environment name determines the naming of Azure Container Apps environment resources.
4. Select **Finish** to close the dialog workflow and view the deployment environment summary.
5. Select **Publish** to provision and deploy the resources on Azure. This process may take several minutes to complete. Visual Studio provides status updates on the

deployment progress.

- When the publish completes, Visual Studio displays the resource URLs at the bottom of the environment screen. Use these links to view the various deployed resources. Select the **webfrontend** URL to open a browser to the deployed app.

The screenshot shows the Azure Container Apps for .NET Aspire environment page. At the top, there's a navigation bar with a logo, the name "aspire-vs", a dropdown menu, and a "Publish" button. Below the navigation is a message box indicating "Deployment succeeded. Deployment ID: aspire-vs-1714076079 on 4/25/2024 4:16 PM." with a link to "Open in Azure portal". The main content area is divided into sections: "Environment" (Name: aspire-vs, Subscription: [redacted], Location: East US 2 (eastus2)), "Resources" (listing Container App, Container Apps Environment, Container registry, Managed Identity, Log Analytics workspace, and Resource group), and "Services" (listing apbservice and webfrontend with their respective endpoints).

Test the deployed app

Now that the app has been provisioned and deployed, you can browse to the Azure portal. In the resource group where you deployed the app, you'll see the three container apps and other resources.

Home >

Search

+ Create Manage view Delete resource group Refresh Export to CSV Group by none

View Cost JSON View

Overview

Activity log Access control (IAM) Tags Resource visualizer Events

Settings

Deployments Security Deployment stacks Policies Properties Locks

Cost Management

Cost analysis Cost alerts (preview) Budgets Advisor recommendations

Monitoring

Insights (preview) Alerts Metrics Diagnostic settings

Essentials

Subscription (move) Deployments 1 Succeeded

Subscription ID

Location

Tags (edit)

azd-env-name : david-pine-rg

Resources Recommendations

Filter for any field... Type equals all Location equals all Add filter

Recommendations 0 Changed resources 7 Unsecure resources 0

Name ↑	Type	Location
Container registry	Container registry	[redacted]
apiservice	Container App	[redacted]
cache	Container App	[redacted]
[redacted]	Container Apps Environment	[redacted]
[redacted]	Log Analytics workspace	[redacted]
[redacted]	Managed Identity	[redacted]
webfrontend	Container App	[redacted]

Showing 1 - 7 of 7 results.

Give feedback

Click on the `web` Container App to open it up in the portal.

Home > >

webfrontend Container App

Stop Refresh Delete Send us your feedback

View Cost JSON View

Essentials

Resource group (move) : [redacted]
Status : [redacted]
Location (move) : [redacted]
Subscription (move) : [redacted]
Subscription ID : [redacted]

Application Url : [https://webfrontend.\[redacted\].az...](https://webfrontend.[redacted].az...)

Container Apps Environment : [redacted]
Environment type : [redacted]
Log Analytics : [redacted]

Give feedback

Click the Application URL link to open the front end in the browser.

AspireToAca

Home Counter Weather

About

Weather

This component demonstrates showing data loaded from a backend API service.

Date	Temp. (C)	Temp. (F)	Summary
10/27/2023	20	67	Warm
10/28/2023	40	103	Scorching
10/29/2023	16	60	Balmy
10/30/2023	-11	13	Cool
10/31/2023	39	102	Bracing

Give feedback

When you click the "Weather" node in the navigation bar, the front end `web` container app makes a call to the `apiservice` container app to get data. The front end's output will be cached using the `redis` container app and the [.NET Aspire Redis Output Caching component](#). As you refresh the front end a few times, you'll notice that the weather data is cached. It will update after a few seconds.

Deploy the .NET Aspire Dashboard

You can deploy the .NET Aspire dashboard as part of your hosted app. This feature is currently in alpha support, so you must enable the `alpha.aspire.dashboard` [feature flag](#). When enabled, the `azd` output logs print an additional URL to the deployed dashboard.

```
Azure Developer CLI
```

```
azd config set alpha.aspire.dashboard on
```

You can also run `azd monitor` to automatically launch the dashboard.

```
Azure Developer CLI
```

```
azd monitor
```

Clean up resources

To delete the `azd` environment, the **More actions** dropdown and then choose **Delete environment**.

The screenshot shows the Azure Container Apps for .NET Aspire portal. At the top, there is a navigation bar with a logo, a dropdown menu, and a 'Publish' button. Below the navigation bar, there is a toolbar with a 'New environment' button, a 'More actions' dropdown, and a status message 'Ready to publish.' The 'More actions' dropdown is open, showing options: 'Refresh environment' (with a circular arrow icon) and 'Delete environment' (with a red X icon). The main content area displays the environment settings for 'another'. Under the 'Environment' section, it shows the 'Name' as 'another', 'Subscription' as 'Azure SDK Developer Playground', and 'Location' as 'West US 3 (westus3)'. There is also a '...' button. Under the 'Resources' section, it says 'Azure resources are not provisioned yet.'.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Deploy a .NET Aspire project to Azure Container Apps using the Azure Developer CLI (in-depth guide)

Article • 06/15/2024

The Azure Developer CLI (`azd`) has been extended to support deploying .NET Aspire projects. Use this guide to walk through the process of creating and deploying a .NET Aspire project to Azure Container Apps using the Azure Developer CLI. In this tutorial, you'll learn the following concepts:

- ✓ Explore how `azd` integration works with .NET Aspire projects
- ✓ Provision and deploy resources on Azure for a .NET Aspire project using `azd`
- ✓ Generate Bicep infrastructure and other template files using `azd`

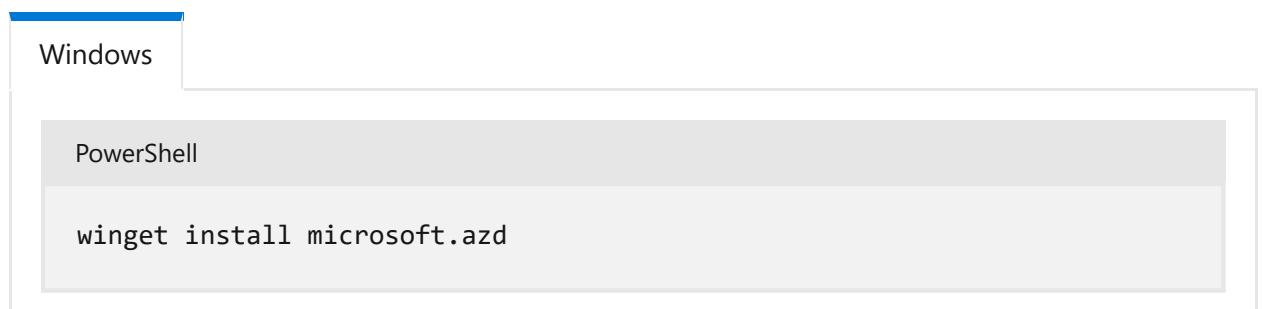
Prerequisites

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

- [.NET 8.0](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or [the .NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

You will also need to have the Azure Developer CLI [installed locally](#). Common install options include the following:

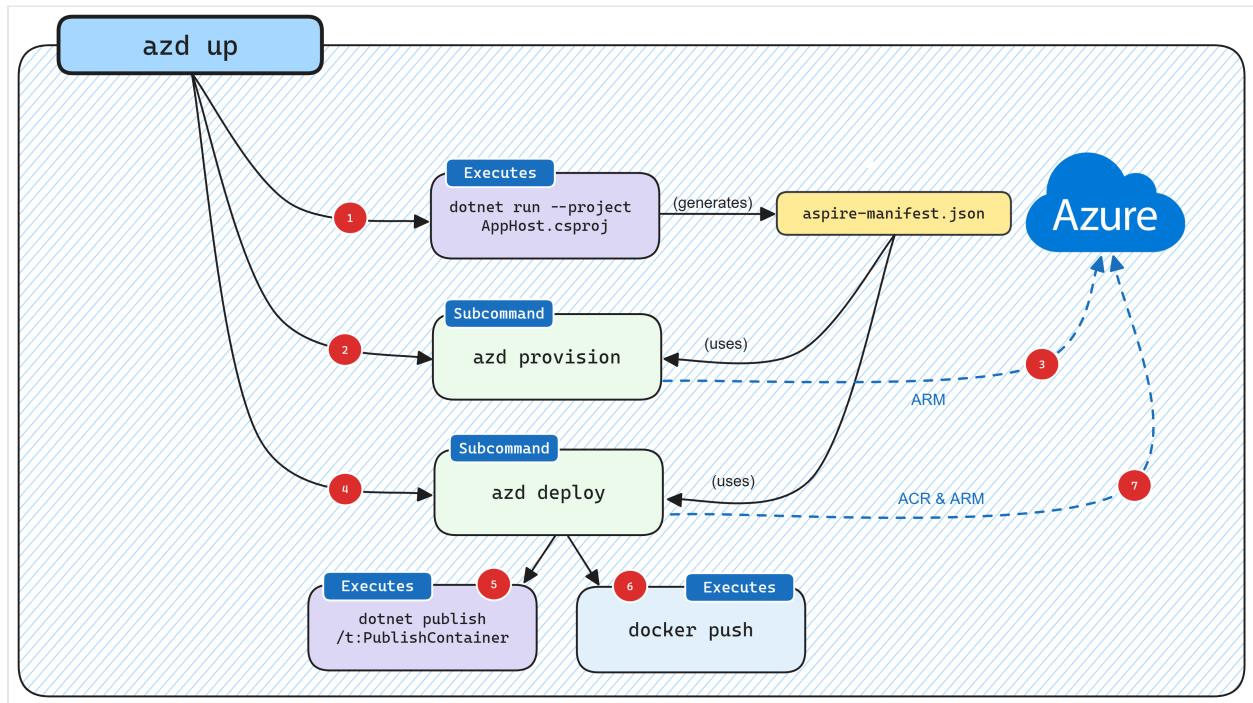


A screenshot of a Windows terminal window. The title bar says "Windows PowerShell". The command "winget install microsoft_azd" is typed into the terminal.

```
Windows PowerShell
winget install microsoft_azd
```

How Azure Developer CLI integration works

The `azd init` workflow provides customized support for .NET Aspire projects. The following diagram illustrates how this flow works conceptually and how `azd` and .NET Aspire are integrated:



1. When `azd` targets a .NET Aspire project it starts the `AppHost` with a special command (`dotnet run --project AppHost.csproj --output-path manifest.json --publisher manifest`), which produces the Aspire **manifest file**.
2. The manifest file is interrogated by the `azd provision` sub-command logic to generate Bicep files in-memory only (by default).
3. After generating the Bicep files, a deployment is triggered using Azure's ARM APIs targeting the subscription and resource group provided earlier.
4. Once the underlying Azure resources are configured, the `azd deploy` sub-command logic is executed which uses the same Aspire manifest file.
5. As part of deployment `azd` makes a call to `dotnet publish` using .NET's built in container publishing support to generate container images.
6. Once `azd` has built the container images it pushes them to the ACR registry that was created during the provisioning phase.
7. Finally, once the container image is in ACR, `azd` updates the resource using ARM to start using the new version of the container image.

! Note

`azd` also enables you to output the generated Bicep to an `infra` folder in your project, which you can read more about in the [Generating Bicep from .NET Aspire app model](#) section.

Provision and deploy a .NET Aspire starter app

The steps in this section demonstrate how to create a .NET Aspire start app and handle provisioning and deploying the app resources to Azure using `azd`.

Create the .NET Aspire starter app

Create a new .NET Aspire project using the `dotnet new` command. You can also create the project using Visual Studio.

.NET CLI

```
dotnet new aspire-starter --use-redis-cache -o AspireSample
cd AspireSample
dotnet run --project AspireSample.AppHost\AspireSample.AppHost.csproj
```

The previous commands create a new .NET Aspire project based on the `aspire-starter` template which includes a dependency on Redis cache. It runs the .NET Aspire project which verifies that everything is working correctly.

Initialize the template

1. Open a new terminal window and `cd` into the `AppHost` project directory of your .NET Aspire solution.
2. Execute the `azd init` command to initialize your project with `azd`, which will inspect the local directory structure and determine the type of app.

Azure Developer CLI

```
azd init
```

For more information on the `azd init` command, see [azd init](#).

3. Select **Use code in the current directory** when `azd` prompts you with two app initialization options.

Output

```
? How do you want to initialize your app? [Use arrows to move, type to filter]
> Use code in the current directory
  Select a template
```

4. After scanning the directory, `azd` prompts you to confirm that it found the correct .NET Aspire *AppHost* project. Select the **Confirm and continue initializing my app** option.

Output

```
Detected services:
```

```
.NET (Aspire)
Detected in:
D:\source\repos\AspireSample\AspireSample.AppHost\AspireSample.AppHost.csproj
```

```
azd will generate the files necessary to host your app on Azure using
Azure Container Apps.
```

```
? Select an option [Use arrows to move, type to filter]
> Confirm and continue initializing my app
  Cancel and exit
```

5. Enter an environment name, which is used to name provisioned resources in Azure and managing different environments such as `dev` and `prod`.

Output

```
Generating files to run your app on Azure:
```

```
(✓) Done: Generating ./azure.yaml
(✓) Done: Generating ./next-steps.md
```

```
SUCCESS: Your app is ready for the cloud!
```

```
You can provision and deploy your app to Azure by running the azd up
command in this directory. For more information on configuring your
app, see ./next-steps.md
```

`azd` generates a number of files and places them into the working directory. These files are:

- *azure.yaml*: Describes the services of the app, such as .NET Aspire AppHost project, and maps them to Azure resources.

- `.azure/config.json`: Configuration file that informs `azd` what the current active environment is.
- `.azure/aspireazddev/.env`: Contains environment specific overrides.

The `azure.yaml` file has the following contents:

```
yml

# yaml-language-server:
$schema=https://raw.githubusercontent.com/Azure/azure-
dev/main/schemas/v1.0/azure.yaml.json

name: AspireSample
services:
  app:
    language: dotnet
    project: .\AspireSample.AppHost\AspireSample.AppHost.csproj
    host: containerapp
```

Resource naming

When you create new Azure resources, it's important to follow the naming requirements. For Azure Container Apps, the name must be 2-32 characters long and consist of lowercase letters, numbers, and hyphens. The name must start with a letter and end with an alphanumeric character.

For more information, see [Naming rules and restrictions for Azure resources](#).

Initial deployment

1. In order to deploy the .NET Aspire project, authenticate to Azure AD to call the Azure resource management APIs.

```
Azure Developer CLI
```

```
azd auth login
```

The previous command will launch a browser to authenticate the command-line session.

2. Once authenticated, run the following command from the `AppHost` project directory to provision and deploy the application.

```
Azure Developer CLI
```

```
azd up
```

ⓘ Important

To push container images to the Azure Container Registry (ACR), you need to have `Microsoft.Authorization/roleAssignments/write` access. This can be achieved by enabling an **Admin user** on the registry. Open the Azure Portal, navigate to the ACR resource / Settings / Access keys, and then select the **Admin user** checkbox. For more information, see [Enable admin user](#).

3. When prompted, select the subscription and location the resources should be deployed to. Once these options are selected the .NET Aspire project will be deployed.

Output

By default, a service can only be reached from inside the Azure Container Apps environment it is running in. Selecting a service here will also allow it to be reached from the Internet.

- ? Select which services to expose to the Internet webfrontend
- ? Select an Azure Subscription to use: 1. <YOUR SUBSCRIPTION>
- ? Select an Azure location to use: 1. <YOUR LOCATION>

Packaging services (azd package)

SUCCESS: Your application was packaged for Azure in less than a second.

Provisioning Azure resources (azd provision)
Provisioning Azure resources can take some time.

Subscription: <YOUR SUBSCRIPTION>
Location: <YOUR LOCATION>

You can view detailed progress in the Azure Portal:
<LINK TO DEPLOYMENT>

- (✓) Done: Resource group: <YOUR RESOURCE GROUP>
- (✓) Done: Container Registry: <ID>
- (✓) Done: Log Analytics workspace: <ID>
- (✓) Done: Container Apps Environment: <ID>

SUCCESS: Your application was provisioned in Azure in 1 minute 13 seconds.

You can view the resources created under the resource group <YOUR RESOURCE GROUP> in Azure Portal:
<LINK TO RESOURCE GROUP OVERVIEW>

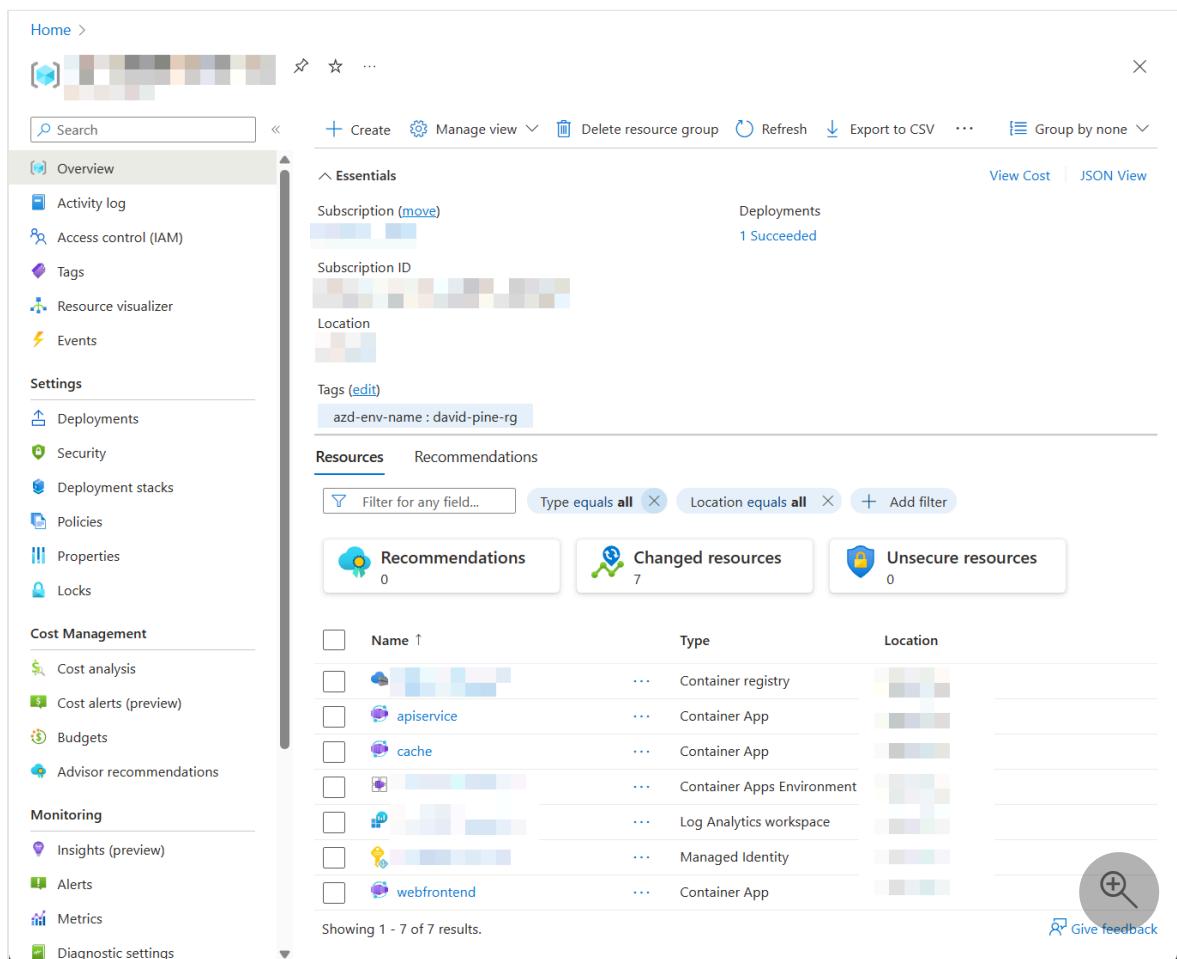
Deploying services (azd deploy)

```
(✓) Done: Deploying service apiservice
- Endpoint: <YOUR UNIQUE apiservice APP>.azurecontainerapps.io/
(✓) Done: Deploying service webfrontend
- Endpoint: <YOUR UNIQUE webfrontend APP>.azurecontainerapps.io/
```

SUCCESS: Your application was deployed to Azure in 1 minute 39 seconds.
You can view the resources created under the resource group <YOUR RESOURCE GROUP> in Azure Portal:
[<LINK TO RESOURCE GROUP OVERVIEW>](#)

SUCCESS: Your up workflow to provision and deploy to Azure completed in 3 minutes 50 seconds.

The final line of output from the `azd` command is a link to the Azure Portal that shows all of the Azure resources that were deployed:



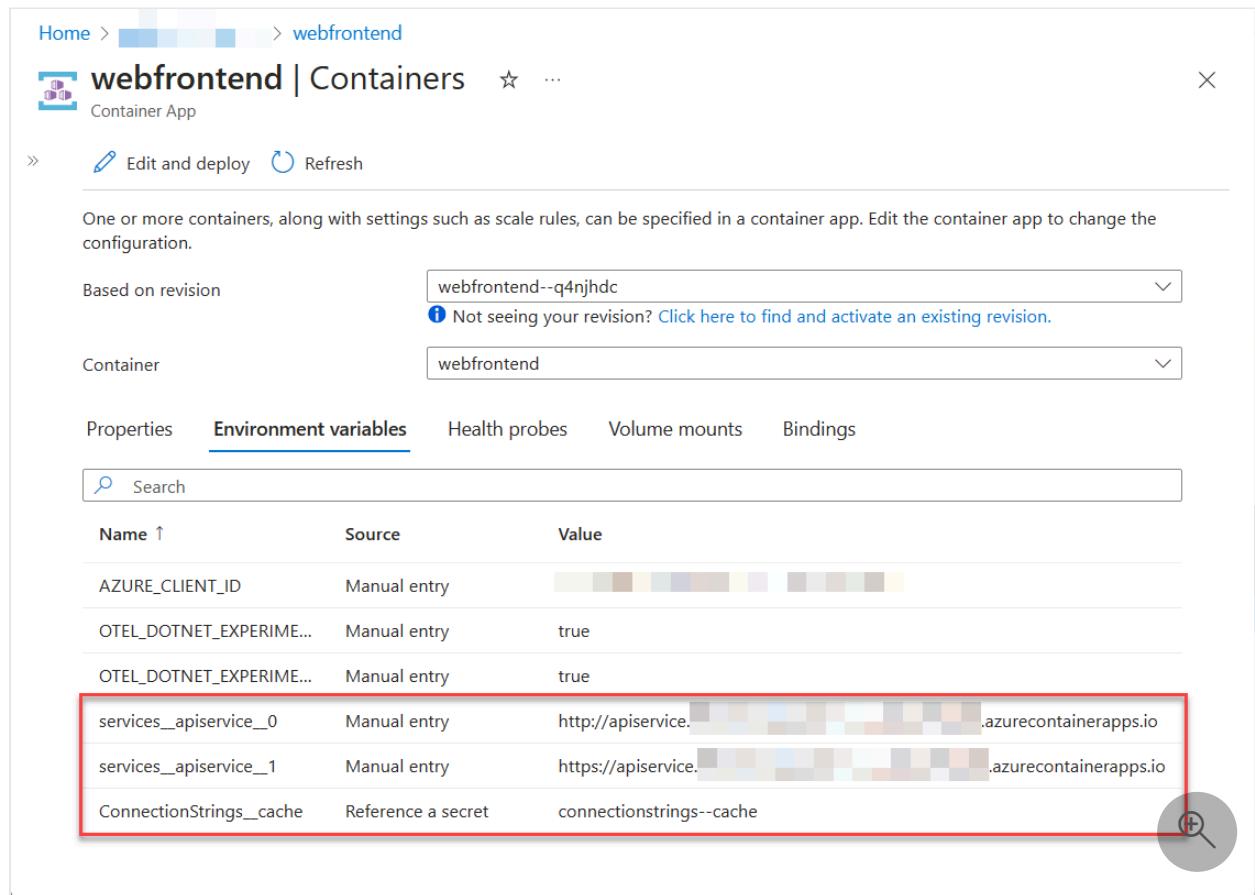
The screenshot shows the Azure Portal's Resource Groups page. The left sidebar lists various categories like Overview, Settings, Cost Management, Monitoring, etc. The main area displays the 'Essentials' section with details such as Subscription (move), Subscription ID, Location, and Tags. Below this, the 'Resources' section shows recommendations, changed resources (7 items), and unsecure resources (0 items). A table lists the deployed resources: apiservice (Container App), cache (Container App), and webfrontend (Container App). The table includes columns for Name, Type, and Location. At the bottom right, there are 'Give feedback' and search icons.

Name	Type	Location
apiservice	Container App	[Redacted]
cache	Container App	[Redacted]
webfrontend	Container App	[Redacted]

Three containers are deployed within this application:

- `webfrontend`: Contains code from the web project in the starter template.
- `apiservice`: Contains code from the API service project in the starter template.
- `cache`: A Redis container image to supply a cache to the front-end.

Just like in local development, the configuration of connection strings has been handled automatically. In this case, `azd` was responsible for interpreting the application model and translating it to the appropriate deployment steps. As an example, consider the connection string and service discovery variables that are injected into the `webfrontend` container so that it knows how to connect to the Redis cache and `apiservice`.



Home > [REDACTED] > webfrontend

webfrontend | Containers

Container App

Edit and deploy Refresh

One or more containers, along with settings such as scale rules, can be specified in a container app. Edit the container app to change the configuration.

Based on revision: webfrontend--q4njhdc
Not seeing your revision? [Click here to find and activate an existing revision.](#)

Container: webfrontend

Properties Environment variables Health probes Volume mounts Bindings

Name ↑	Source	Value
AZURE_CLIENT_ID	Manual entry	[REDACTED]
OTEL_DOTNET_EXPERIME...	Manual entry	true
OTEL_DOTNET_EXPERIME...	Manual entry	true
services_apiservice_0	Manual entry	http://apiservice.[REDACTED].azurecontainerapps.io
services_apiservice_1	Manual entry	https://apiservice.[REDACTED].azurecontainerapps.io
ConnectionStrings_cache	Reference a secret	connectionstrings--cache

For more information on how .NET Aspire projects handle connection strings and service discovery, see [.NET Aspire orchestration overview](#).

Deploy application updates

When the `azd up` command is executed the underlying Azure resources are *provisioned* and a container image is built and *deployed* to the container apps hosting the .NET Aspire project. Typically once development is underway and Azure resources are deployed it won't be necessary to provision Azure resources every time code is updated —this is especially true for the developer inner loop.

To speed up deployment of code changes, `azd` supports deploying code updates in the container image. This is done using the `azd deploy` command:

```
azd deploy
```

Output

Deploying services (azd deploy)

(✓) Done: Deploying service apiservice

- Endpoint: <YOUR UNIQUE apiservice APP>.azurecontainerapps.io/

(✓) Done: Deploying service webfrontend

- Endpoint: <YOUR UNIQUE webfrontend APP>.azurecontainerapps.io/

SUCCESS: Your application was deployed to Azure in 1 minute 54 seconds.
You can view the resources created under the resource group <YOUR RESOURCE GROUP> in Azure Portal:
[<LINK TO RESOURCE GROUP OVERVIEW>](#)

It's not necessary to deploy all services each time. `azd` understands the .NET Aspire project model, it's possible to deploy just one of the services specified using the following command:

Azure Developer CLI

```
azd deploy webfrontend
```

For more information, see [Azure Developer CLI reference: azd deploy](#).

Deploy infrastructure updates

Whenever the dependency structure within a .NET Aspire project changes, `azd` must re-provision the underlying Azure resources. The `azd provision` command is used to apply these changes to the infrastructure.

To see this in action, update the `Program.cs` file in the `AppHost` project to the following:

C#

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

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

// Add the locations database.
var locationsdb = builder.AddPostgres("db").AddDatabase("locations");

// Add the locations database reference to the API service.
```

```
var apiservice = builder.AddProject<Projects.AspireSample_ApiService>("apiservice")
    .WithReference(locationsdb);

builder.AddProject<Projects.AspireSample_Web>("webfrontend")
    .WithReference(cache)
    .WithReference(apiservice);

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

Save the file and issue the following command:

Azure Developer CLI

```
azd provision
```

The `azd provision` command updates the infrastructure by creating a container app to host the Postgres database. The `azd provision` command didn't update the connection strings for the `apiservice` container. In order to have connection strings updated to point to the newly provisioned Postgres database the `azd deploy` command needs to be invoked again. When in doubt, use `azd up` to both provision and deploy.

Clean up resources

Remember to clean up the Azure resources that you've created during this walkthrough. Because `azd` knows the resource group in which it created the resources it can be used to spin down the environment using the following command:

Azure Developer CLI

```
azd down
```

The previous command may take some time to execute, but when completed the resource group and all its resources should be deleted.

Output

Deleting all resources and deployed code on Azure (azd down)
Local application code is not deleted when running 'azd down'.

Resource group(s) to be deleted:

- <YOUR RESOURCE GROUP>: <LINK TO RESOURCE GROUP OVERVIEW>

? Total resources to delete: 7, are you sure you want to continue? Yes
Deleting your resources can take some time.

(✓) Done: Deleting resource group: <YOUR RESOURCE GROUP>

SUCCESS: Your application was removed from Azure in 9 minutes 59 seconds.

Generate Bicep from .NET Aspire project model

Although development teams are free to use `azd up` (or `azd provision` and `azd deploy`) commands for their deployments both for development and production purposes, some teams may choose to generate Bicep files that they can review and manage as part of version control (this also allows these Bicep files to be referenced as part of a larger more complex Azure deployment).

`azd` includes the ability to output the Bicep it uses for provisioning via following command:

Azure Developer CLI

```
azd config set alpha.infraSynth on  
azd infra synth
```

After this command is executed in the starter template example used in this guide, the following files are created in the *AppHost* project directory:

- *infra/main.bicep*: Represents the main entry point for the deployment.
- *infra/main.parameters.json*: Used as the parameters for main Bicep (maps to environment variables defined in *.azure* folder).
- *infra/resources.bicep*: Defines the Azure resources required to support the .NET Aspire project model.
- *AspireSample.Web/manifests/containerApp.tpl.yaml*: The container app definition for `webfrontend`.
- *AspireSample.ApiService/manifests/containerApp.tpl.yaml*: The container app definition for `apiservice`.

The *infra\resources.bicep* file doesn't contain any definition of the container apps themselves (with the exception of container apps which are dependencies such as Redis and Postgres):

Bicep

```
@description('The location used for all deployed resources')  
param location string = resourceGroup().location
```

```

@description('Tags that will be applied to all resources')
param tags object = {}

var resourceToken = uniqueString(resourceGroup().id)

resource managedIdentity
'Microsoft.ManagedIdentity/userAssignedIdentities@2023-01-31' = {
  name: 'mi-${resourceToken}'
  location: location
  tags: tags
}

resource containerRegistry 'Microsoft.ContainerRegistry/registries@2023-07-01' = {
  name: replace('acr-${resourceToken}', '-', '')
  location: location
  sku: {
    name: 'Basic'
  }
  tags: tags
}

resource caeMiRoleAssignment 'Microsoft.Authorization/roleAssignments@2022-04-01' = {
  name: guid(containerRegistry.id, managedIdentity.id,
  subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '7f951dda-4ed3-4680-a7ca-43fe172d538d'))
  scope: containerRegistry
  properties: {
    principalId: managedIdentity.properties.principalId
    principalType: 'ServicePrincipal'
    roleDefinitionId:
  subscriptionResourceId('Microsoft.Authorization/roleDefinitions', '7f951dda-4ed3-4680-a7ca-43fe172d538d')
  }
}

resource logAnalyticsWorkspace
'Microsoft.OperationalInsights/workspaces@2022-10-01' = {
  name: 'law-${resourceToken}'
  location: location
  properties: {
    sku: {
      name: 'PerGB2018'
    }
  }
  tags: tags
}

resource containerAppEnvironment 'Microsoft.App/managedEnvironments@2023-05-01' = {
  name: 'cae-${resourceToken}'
  location: location
  properties: {

```

```
    appLogsConfiguration: {
        destination: 'log-analytics'
        logAnalyticsConfiguration: {
            customerId: logAnalyticsWorkspace.properties.customerId
            sharedKey: logAnalyticsWorkspace.listKeys().primarySharedKey
        }
    }
}
tags: tags
}

resource cache 'Microsoft.App/containerApps@2023-05-02-preview' = {
    name: 'cache'
    location: location
    properties: {
        environmentId: containerAppEnvironment.id
        configuration: {
            service: {
                type: 'redis'
            }
        }
        template: {
            containers: [
                {
                    image: 'redis'
                    name: 'redis'
                }
            ]
        }
    }
    tags: union(tags, {'aspire-resource-name': 'cache'})
}

resource locations 'Microsoft.App/containerApps@2023-05-02-preview' = {
    name: 'locations'
    location: location
    properties: {
        environmentId: containerAppEnvironment.id
        configuration: {
            service: {
                type: 'postgres'
            }
        }
        template: {
            containers: [
                {
                    image: 'postgres'
                    name: 'postgres'
                }
            ]
        }
    }
    tags: union(tags, {'aspire-resource-name': 'locations'})
}

output MANAGED_IDENTITY_CLIENT_ID string =
```

```
managedIdentity.properties.clientId
output AZURE_CONTAINER_REGISTRY_ENDPOINT string =
containerRegistry.properties.loginServer
output AZURE_CONTAINER_REGISTRY_MANAGED_IDENTITY_ID string =
managedIdentity.id
output AZURE_CONTAINER_APPS_ENVIRONMENT_ID string =
containerAppEnvironment.id
output AZURE_CONTAINER_APPS_ENVIRONMENT_DEFAULT_DOMAIN string =
containerAppEnvironment.properties.defaultDomain
```

For more information on using Bicep to automate deployments to Azure see, [What is Bicep?](#)

The definition of the container apps from the .NET service projects is contained within the *containerApp/tmplyaml* files in the `manifests` directory in each project respectively. Here is an example from the `webfrontend` project:

```
yml

location: {{ .Env.AZURE_LOCATION }}
identity:
  type: UserAssigned
  userAssignedIdentities:
    ? "{{ .Env.AZURE_CONTAINER_REGISTRY_MANAGED_IDENTITY_ID }}"
    : {}
properties:
  environmentId: {{ .Env.AZURE_CONTAINER_APPS_ENVIRONMENT_ID }}
  configuration:
    activeRevisionsMode: single
  ingress:
    external: true
    targetPort: 8080
    transport: http
    allowInsecure: false
  registries:
    - server: {{ .Env.AZURE_CONTAINER_REGISTRY_ENDPOINT }}
      identity: {{ .Env.AZURE_CONTAINER_REGISTRY_MANAGED_IDENTITY_ID }}
template:
  containers:
    - image: {{ .Env.SERVICE_WEBFRONTEND_IMAGE_NAME }}
      name: webfrontend
      env:
        - name: AZURE_CLIENT_ID
          value: {{ .Env.MANAGED_IDENTITY_CLIENT_ID }}
        - name: ConnectionStrings__cache
          value: {{ connectionString "cache" }}
        - name: OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EVENT_LOG_ATTRIBUTES
          value: "true"
        - name: OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EXCEPTION_LOG_ATTRIBUTES
          value: "true"
        - name: services__apiservice_0
          value: http://apiservice.internal.{{}}
```

```
.Env.AZURE_CONTAINER_APPS_ENVIRONMENT_DEFAULT_DOMAIN }}
  - name: services_apiservice_1
    value: https://apiservice.internal.{{
.Env.AZURE_CONTAINER_APPS_ENVIRONMENT_DEFAULT_DOMAIN }}}
tags:
  azd-service-name: webfrontend
  aspire-resource-name: webfrontend
```

After executing the `azd infra synth` command, when `azd provision` and `azd deploy` are called they use the Bicep and supporting generated files.

ⓘ Important

If `azd infra synth` is called again, it replaces any modified files with freshly generated ones and prompts you for confirmation before doing so.

Isolated environments for debugging

Because `azd` makes it easy to provision new environments, it's possible for each team member to have an isolated cloud-hosted environment for debugging code in a setting that closely matches production. When doing this each team member should create their own environment using the following command:

Azure Developer CLI

```
azd env new
```

This will prompt the user for subscription and resource group information again and subsequent `azd up`, `azd provision`, and `azd deploy` invocations will use this new environment by default. The `--environment` switch can be applied to these commands to switch between environments.

Clean up resources

Run the following Azure CLI command to delete the resource group when you no longer need the Azure resources you created. Deleting the resource group also deletes the resources contained inside of it.

Azure CLI

```
az group delete --name <your-resource-group-name>
```

For more information, see [Clean up resources in Azure](#).

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Tutorial: Deploy a .NET Aspire project using the Azure Developer CLI and GitHub Actions

Article • 07/10/2024

The Azure Developer CLI (`azd`) enables you to deploy .NET Aspire projects using GitHub Actions by automatically configuring the required authentication and environment settings. This article walks you through the process of creating and deploying a .NET Aspire project on Azure Container Apps using `azd` and GitHub Actions. You learn the following concepts:

- ✓ Explore how `azd` integration works with .NET Aspire projects and GitHub Actions
- ✓ Create and configure a GitHub repository for a .NET Aspire project using `azd`
- ✓ Add a GitHub Actions workflow file to your .NET Aspire solution
- ✓ Monitor and explore GitHub Actions workflow executions and Azure deployments

Prerequisites

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

- [.NET 8.0](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or [the .NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

You also need to have the Azure Developer CLI [installed locally](#) (version 1.5.1 or higher). Common install options include the following:

Windows

PowerShell

```
winget install microsoft.azd
```

Create a .NET Aspire solution

As a starting point, this article assumes that you've created a .NET Aspire solution from the **.NET Aspire Starter Application** template. For more information, see [Quickstart: Build your first .NET Aspire app.](#)

Initialize the template

1. Open a new terminal window and `cd` into the *AppHost* project directory of your .NET Aspire solution.
2. Execute the `azd init` command to initialize your project with `azd`, which will inspect the local directory structure and determine the type of app.

Azure Developer CLI

```
azd init
```

For more information on the `azd init` command, see [azd init](#).

3. Select **Use code in the current directory** when `azd` prompts you with two app initialization options.

Output

```
? How do you want to initialize your app? [Use arrows to move, type to filter]
> Use code in the current directory
Select a template
```

4. After scanning the directory, `azd` prompts you to confirm that it found the correct .NET Aspire *AppHost* project. Select the **Confirm and continue initializing my app** option.

Output

Detected services:

.NET (Aspire)

```
Detected in:  
D:\source\repos\AspireSample\AspireSample.AppHost\AspireSample.AppHost.csproj  
  
azd will generate the files necessary to host your app on Azure using  
Azure Container Apps.  
  
? Select an option [Use arrows to move, type to filter]  
> Confirm and continue initializing my app  
Cancel and exit
```

5. Enter an environment name, which is used to name provisioned resources in Azure and managing different environments such as `dev` and `prod`.

Output

```
Generating files to run your app on Azure:
```

```
(✓) Done: Generating ./azure.yaml  
(✓) Done: Generating ./next-steps.md
```

```
SUCCESS: Your app is ready for the cloud!
```

```
You can provision and deploy your app to Azure by running the azd up  
command in this directory. For more information on configuring your  
app, see ./next-steps.md
```

`azd` generates a number of files and places them into the working directory. These files are:

- `azure.yaml`: Describes the services of the app, such as .NET Aspire AppHost project, and maps them to Azure resources.
- `.azure/config.json`: Configuration file that informs `azd` what the current active environment is.
- `.azure/aspireazddev/.env`: Contains environment specific overrides.

Add the GitHub Actions workflow file

Although `azd` generated some essential template files for you, the project still needs a GitHub Actions workflow file to support provisioning and deployments using CI/CD.

1. Create an empty `.github` folder at the root of your project. `azd` uses this directory by default to discover GitHub Actions workflow files.

 Tip

If you're on macOS user and you're struggling to create a folder with a leading `.`, you can use the terminal to create the folder. Open the terminal and navigate to the root of your project. Run the following command to create the folder:

Bash

```
mkdir .github
```

2. Inside the new `.github` folder, create another folder called `workflows` (you'll end up with `.github/workflows`).
3. Add a new GitHub Actions workflow file into the new folder named `azure-dev.yml`. The `azd` starter template provides a [Sample GitHub Actions workflow file](#) that you can copy into your project.
4. Update the sample GitHub Actions workflow to include a step to install the .NET Aspire workload. This ensures the .NET Aspire tooling and commands are available to the job running your GitHub Actions. The completed workflow file should match the following:

yml

```
on:  
  workflow_dispatch:  
  push:  
    # Run when commits are pushed to mainline branch (main or master)  
    # Set this to the mainline branch you are using  
    branches:  
      - main  
  
permissions:  
  id-token: write  
  contents: read  
  
jobs:  
  build:  
    runs-on: ubuntu-latest  
    env:  
      AZURE_CLIENT_ID: ${{ vars.AZURE_CLIENT_ID }}  
      AZURE_TENANT_ID: ${{ vars.AZURE_TENANT_ID }}  
      AZURE_SUBSCRIPTION_ID: ${{ vars.AZURE_SUBSCRIPTION_ID }}  
      AZURE_CREDENTIALS: ${{ secrets.AZURE_CREDENTIALS }}  
      AZURE_ENV_NAME: ${{ vars.AZURE_ENV_NAME }}  
      AZURE_LOCATION: ${{ vars.AZURE_LOCATION }}  
    steps:  
      - name: Checkout  
        uses: actions/checkout@v4
```

```

- name: Install azd
  uses: Azure/setup-azd@v1.0.0

- name: Install .NET Aspire workload
  run: dotnet workload install aspire

- name: Log in with Azure (Federated Credentials)
  if: ${{ env.AZURE_CLIENT_ID != '' }}
  run: |
    azd auth login ` 
      --client-id "$Env:AZURE_CLIENT_ID" ` 
      --federated-credential-provider "github" ` 
      --tenant-id "$Env:AZURE_TENANT_ID"
  shell: pwsh

- name: Log in with Azure (Client Credentials)
  if: ${{ env.AZURE_CREDENTIALS != '' }}
  run: |
    $info = $Env:AZURE_CREDENTIALS | ConvertFrom-Json - 
AsHashtable;
    Write-Host "::add-mask:: $($info.clientSecret)"

    azd auth login ` 
      --client-id "$($info.clientId)" ` 
      --client-secret "$($info.clientSecret)" ` 
      --tenant-id "$($info.tenantId)"
  shell: pwsh

- name: Provision Infrastructure
  run: azd provision --no-prompt
  # Required when
  # env:
  #   AZD_INITIAL_ENVIRONMENT_CONFIG: ${{ 
secrets.AZD_INITIAL_ENVIRONMENT_CONFIG }}

  # Required when provisioning and deploying are defined in
  # separate jobs.
  # - name: Refresh azd env (pulls latest infrastructure provision)
  #   run: azd env refresh
  #   env:
  #     AZURE_LOCATION: ${{ env.AZURE_LOCATION }}

- name: Deploy Application
  run: azd deploy --no-prompt

```

Additionally, you may notice that the provisioning and deployment steps are combined into a single job. If you prefer to separate these steps into different jobs, you can do so by creating two separate jobs in the workflow file. The provisioning job should run first, followed by the deployment job. The deployment job should include the `AZD_INITIAL_ENVIRONMENT_CONFIG` secret to ensure the deployment job has access to the environment configuration. You'd also need to uncomment the `azd env refresh` step in

the deployment job to ensure the deployment job has access to the latest infrastructure provision.

Create the GitHub repository and pipeline

The Azure Developer CLI enables you to automatically create CI/CD pipelines with the correct configurations and permissions to provision and deploy resources to Azure. `azd` can also create a GitHub repository for your app if it doesn't exist already.

1. Run the `azd pipeline config` command to configure your deployment pipeline and securely connect it to Azure:

```
Azure Developer CLI
```

```
azd pipeline config
```

2. Select the subscription to provision and deploy the app resources to.
3. Select the Azure location to use for the resources.
4. When prompted whether to create a new Git repository in the directory, enter `y` and press `Enter`.

ⓘ Note

Creating a GitHub repository required you being logged into GitHub. There are a few selections that vary based on your preferences. After logging in, you will be prompted to create a new repository in the current directory.

5. Select **Create a new private GitHub repository** to configure the git remote.
6. Enter a name of your choice for the new GitHub repository or press enter to use the default name. `azd` creates a new repository in GitHub and configures it with the necessary secrets required to authenticate to Azure.

```
C:\Projects\azdsamples\cicd>azd pipeline config
? Select an Azure Subscription to use: 18. Azure Dev Platform Services Content Team (2123e5ff-23f5-4a99-a6c9-19e619433f)
  (V) Done: Retrieving locations...
? Select an Azure location to use: 43. (US) East US 2 (eastus2)

Configure your GitHub pipeline

(1) Warning: Checking current directory for Git repository
No GitHub repository detected.

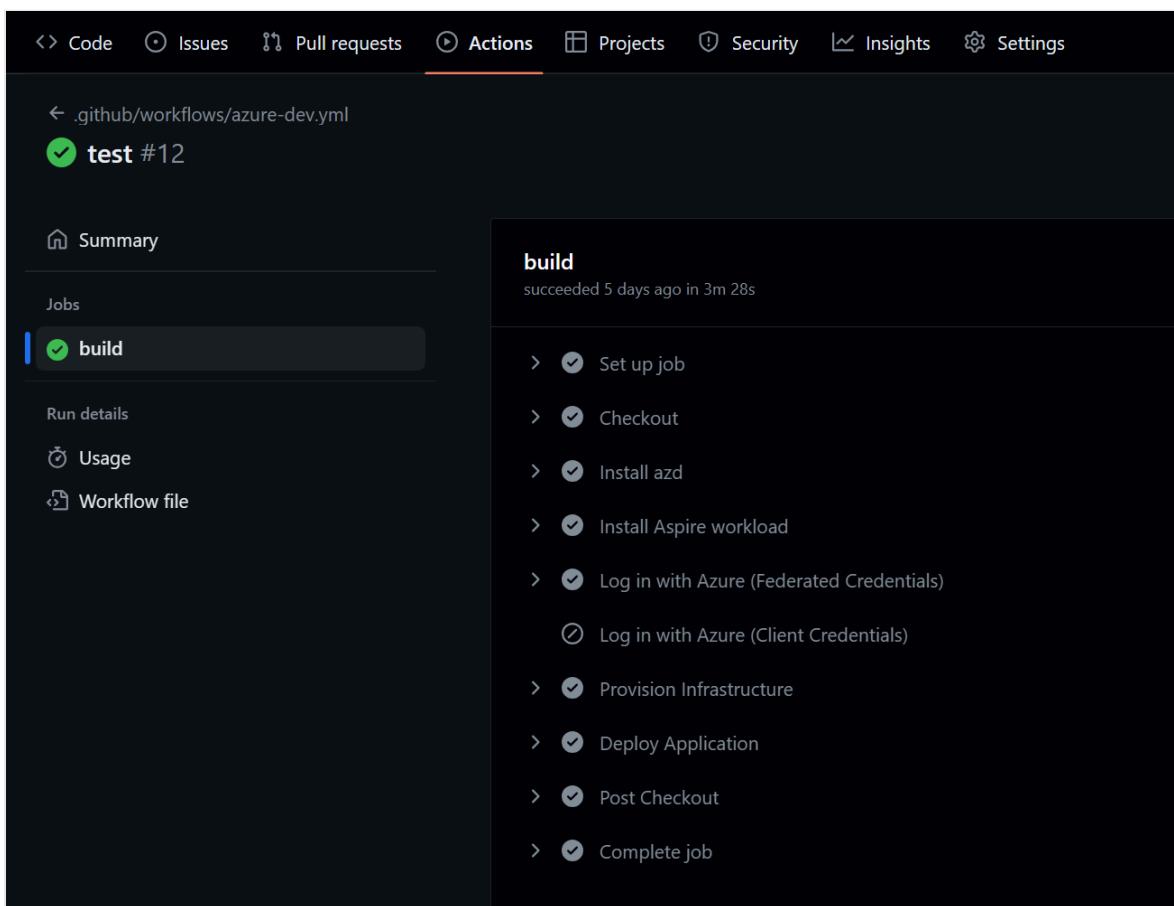
? Do you want to initialize a new Git repository in this directory? Yes
  (V) Done: Creating Git repository locally.

? How would you like to configure your git remote to GitHub? Create a new private GitHub repository
? Enter the name for your new repository OR Hit enter to use this name: (cicd)
```

7. Enter `y` to proceed when `azd` prompts you to commit and push your local changes to start the configured pipeline.

Explore the GitHub Actions workflow and deployment

1. Navigate to your new GitHub repository using the link output by `azd`.
2. Select the **Actions** tab to view the repository workflows. You should see the new workflow either running or already completed. Select the workflow to view the job steps and details in the logs of the run. For example, you can expand steps such as **Install .NET Aspire Workload** or **Deploy application** to see the details of the completed action.



3. Select **Deploy Application** to expand the logs for that step. You should see two endpoint urls printed out for the `apiservice` and `webfrontend`. Select either of these links to open them in another browser tab and explore the deployed application.

✓ Deploy Application

```
1 ► Run azd deploy --no-prompt
11
12 Deploying services (azd deploy)
13
14 |     |
15 |=    |
16 |==   |
17 |==  |
18 |==== |
19 |=====|
20 |=====|
21 |=====|
22 | =====|
23 | =====|
24 | === |
25 | == |
26 (✓) Done: Deploying service apiservice
27 - Endpoint: https://apiservice.internal.mangomeadow-1660bbdc.eastus.azurecontainerapps.io/
28
29 |     |
30 |=    |
31 |==   |
32 |==  |
33 |==== |
34 |=====|
35 |=====|
36 |=====|
37 | =====|
38 | =====|
39 | === |
40 | == |
41 (✓) Done: Deploying service webfrontend
42 - Endpoint: https://webfrontend.internal.mangomeadow-1660bbdc.eastus.azurecontainerapps.io/
43
```

Congratulations! You successfully deployed a .NET Aspire project using the Azure Developer CLI and GitHub Actions.

Clean up resources

Run the following Azure CLI command to delete the resource group when you no longer need the Azure resources you created. Deleting the resource group also deletes the resources contained inside of it.

Azure CLI

```
az group delete --name <your-resource-group-name>
```

For more information, see [Clean up resources in Azure](#).



Collaborate with us on



.NET Aspire feedback

GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET Aspire is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Use Application Insights for .NET Aspire telemetry

Article • 04/12/2024

Azure Application Insights, a feature of Azure Monitor, excels in Application Performance Management (APM) for live web applications. .NET Aspire projects are designed to use OpenTelemetry for application telemetry. OpenTelemetry supports an extension model to support sending data to different APMs. .NET Aspire uses OTLP by default for telemetry export, which is used by the dashboard during development. Azure Monitor doesn't (yet) support OTLP, so the applications need to be modified to use the Azure Monitor exporter, and configured with the connection string.

To use Application insights, you specify its configuration in the app host project *and* use the [Azure Monitor distro in the service defaults project](#).

Choosing how Application Insights is provisioned

.NET Aspire has the capability to provision cloud resources as part of cloud deployment, including Application Insights. In your .NET Aspire project, you can decide if you want .NET Aspire to provision an Application Insights resource when deploying to Azure. You can also select to use an existing Application Insights resource by providing its connection string. The connection information is managed by the resource configuration in the app host project.

Provisioning Application insights during Azure deployment

With this option, an instance of Application Insights will be created for you when the application is deployed using the Azure Developer CLI (`azd`).

To use automatic provisioning, you specify a dependency in the app host project, and reference it in each project/resource that needs to send telemetry to Application Insights. The steps include:

- Add a Nuget package reference to [Aspire.Hosting.Azure.ApplicationInsights](#) in the app host project.

- Update the app host code to use the Application Insights resource, and reference it from each project:

```
C#  
  
var builder = DistributedApplication.CreateBuilder(args);  
  
// Automatically provision an Application Insights resource  
var insights = builder.AddAzureApplicationInsights("MyApplicationInsights");  
  
// Reference the resource from each project  
var apiService = builder.AddProject<Projects.ApiService>("apiservice")  
.WithReference(insights);  
  
builder.AddProject<Projects.Web>("webfrontend")  
.WithReference(apiService)  
.WithReference(insights);  
  
builder.Build().Run();
```

Follow the steps in [Deploy a .NET Aspire project to Azure Container Apps using the Azure Developer CLI \(in-depth guide\)](#) to deploy the application to Azure Container Apps. `azd` will create an Application Insights resource as part of the same resource group, and configure the connection string for each container.

Manual provisioning of Application Insights resource

Application Insights uses a connection string to tell the OpenTelemetry exporter where to send the telemetry data. The connection string is specific to the instance of Application Insights you want to send the telemetry to. It can be found in the Overview page for the application insights instance.

Setting	Value
Instrumentation Key	12345678-1234-abcd-1234-abcd1234abcd
Connection String	InstrumentationKey=12345678-1234-abcd-1234-abcd1234abcd
Workspace	12345678-1234-abcd-1234-abcd1234abcd-aspire-US

If you wish to use an instance of Application Insights that you have provisioned manually, then you should use the `AddConnectionString` API in the app host project to tell the projects/containers where to send the telemetry data. The Azure Monitor distro expects the environment variable to be `APPLICATIONINSIGHTS_CONNECTION_STRING`, so that needs to be explicitly set when defining the connection string.

C#

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

var insights = builder.AddConnectionString(
    "myInsightsResource",
    "APPLICATIONINSIGHTS_CONNECTION_STRING");

var apiService = builder.AddProject<Projects.ApiService>("apiservice")
    .WithReference(insights);

builder.AddProject<Projects.Web>("webfrontend")
    .WithReference(apiService)
    .WithReference(insights);

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

Resource usage during development

When running the .NET Aspire project locally, the preceding code reads the connection string from configuration. As this is a secret, you should store the value in [app secrets](#). Right click on the app host project and choose **Manage Secrets** from the context menu to open the secrets file for the app host project. In the file add the key and your specific connection string, the example below is for illustration purposes.

JSON

```
{
  "ConnectionStrings": {
    "myInsightsResource": "InstrumentationKey=12345678-abcd-1234-abcd-
1234abcd5678;IngestionEndpoint=https://westus3-
1.in.applicationinsights.azure.com"
  }
}
```

⚠ Note

The `name` specified in the app host code needs to match a key inside the `ConnectionStrings` section in the settings file.

Resource usage during deployment

When [deploying an Aspire application with Azure Developer CLI \(azd\)](#), it will recognize the connection string resource and prompt for a value. This enables a different resource

to be used for the deployment from the value used for local development.

Mixed deployment

If you wish to use a different deployment mechanism per execution context, use the appropriate API conditionally. For example, the following code uses a pre-supplied connection at development time, and an automatically provisioned resource at deployment time.

C#

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

var insights = builder.ExecutionContext.IsPublishMode
    ? builder.AddAzureApplicationInsights("myInsightsResource")
    : builder.AddConnectionString("myInsightsResource",
"APPLICATIONINSIGHTS_CONNECTION_STRING");

var apiService = builder.AddProject<Projects.ApiService>("apiservice")
    .WithReference(insights);

builder.AddProject<Projects.Web>("webfrontend")
    .WithReference(apiService)
    .WithReference(insights);

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

💡 Tip

The preceding code requires you to supply the connection string information in app secrets for development time usage, and will be prompted for the connection string by `azd` at deployment time.

Use the Azure Monitor distro

To make exporting to Azure Monitor simpler, this example uses the Azure Monitor Exporter Repo. This is a wrapper package around the Azure Monitor OpenTelemetry Exporter package that makes it easier to export to Azure Monitor with a set of common defaults.

Add the following package to the `ServiceDefaults` project, so that it will be included in each of the .NET Aspire services. For more information, see [.NET Aspire service defaults](#).

XML

```
<PackageReference Include="Azure.Monitor.OpenTelemetry.AspNetCore"
    Version="[SelectVersion]" />
```

Add a using statement to the top of the project.

```
C#
```



```
using Azure.Monitor.OpenTelemetry.AspNetCore;
```

Uncomment the line in `AddOpenTelemetryExporters` to use the Azure Monitor exporter:

```
C#
```



```
private static IHostApplicationBuilder AddOpenTelemetryExporters(
    this IHostApplicationBuilder builder)
{
    // Omitted for brevity...

    // Uncomment the following lines to enable the Azure Monitor exporter
    // (requires the Azure.Monitor.OpenTelemetry.AspNetCore package)
    if
        (!string.IsNullOrEmpty(builder.Configuration["APPLICATIONINSIGHTS_CONNECTION_STRING"]))
    {
        builder.Services.AddOpenTelemetry().UseAzureMonitor();
    }
    return builder;
}
```

It's possible to further customize the Azure Monitor exporter, including customizing the resource name and changing the sampling. For more information, see [Customize the Azure Monitor exporter](#). Using the parameterless version of `UseAzureMonitor()`, will pickup the connection string from the `APPLICATIONINSIGHTS_CONNECTION_STRING` environment variable, we configured via the app host project.

Tutorial: Deploy a .NET Aspire project with a SQL Server Database to Azure

Article • 08/29/2024

In this tutorial, you learn to configure an ASP.NET Core app with a SQL Server Database for deployment to Azure. .NET Aspire provides multiple SQL Server integration configurations that provision different database services in Azure. You'll learn how to:

- ✓ Create a basic ASP.NET Core app that is configured to use the .NET Aspire SQL Server integration
- ✓ Configure the app to provision an Azure SQL Database
- ✓ Configure the app to provision a containerized SQL Server database

ⓘ Note

This document focuses specifically on .NET Aspire configurations to provision and deploy SQL Server resources in Azure. Visit the [Azure Container Apps deployment](#) tutorial to learn more about the full .NET Aspire deployment process.

Prerequisites

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

- [.NET 8.0](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or [the .NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

Create the sample solution

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 Application**. Choose **Next**.
3. On the **Configure your new project** screen:

- Enter a **Project Name of AspireSQL**.
- Leave the rest of the values at their defaults and select **Next**.

4. On the **Additional information** screen:

- Verify that **.NET 8.0** is selected and choose **Create**.

Visual Studio creates a new ASP.NET Core solution that is structured to use .NET Aspire. The solution consists of the following projects:

- **AspireSQL.Web**: A Blazor project that depends on service defaults.
- **AspireSQL.ApiService**: An API project that depends on service defaults.
- **AspireSQL.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.
- **AspireSQL.ServiceDefaults**: A shared class library to hold configurations that can be reused across the projects in your solution.

Configure the app for SQL Server deployment

.NET Aspire provides two built-in configuration options to streamline SQL Server deployment on Azure:

- Provision a containerized SQL Server database using Azure Container Apps
- Provision an Azure SQL Database instance

Add the .NET Aspire integration to the app

Add the appropriate .NET Aspire integration to the *AspireSQL.AppHost* project for your desired hosting service.

Azure SQL Database

Add the [Aspire.Hosting.Azure.Sql](#) package to the *AspireSQL.AppHost* project:

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Sql
```

Configure the AppHost project

Configure the *AspireSQL.AppHost* project for your desired SQL database service.

Azure SQL Database

Replace the contents of the *Program.cs* file in the *AspireSQL.AppHost* project with the following code:

C#

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

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

// Provisions an Azure SQL Database when published
var sqlServer = builder.AddSqlServer("sqlserver")
    .PublishAsAzureSqlDatabase()
    .AddDatabase("sqldb");

builder.AddProject<Projects.AspireSql_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(apiService);

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

The preceding code adds a SQL Server Container resource to your app and configures a connection to a database called `sqldata`. The `PublishAsAzureSqlDatabase` method ensures that tools such as the Azure Developer CLI or Visual Studio create an Azure SQL Database resource during the deployment process.

Deploy the app

Tools such as the [Azure Developer CLI](#) (`azd`) support .NET Aspire SQL Server integration configurations to streamline deployments. `azd` consumes these settings and provisions properly configured resources for you.

Initialize the template

1. Open a new terminal window and `cd` into the *AppHost* project directory of your .NET Aspire solution.
2. Execute the `azd init` command to initialize your project with `azd`, which will inspect the local directory structure and determine the type of app.

```
Azure Developer CLI
```

```
azd init
```

For more information on the `azd init` command, see [azd init](#).

3. Select **Use code in the current directory** when `azd` prompts you with two app initialization options.

```
Output
```

```
? How do you want to initialize your app? [Use arrows to move, type to filter]
> Use code in the current directory
    Select a template
```

4. After scanning the directory, `azd` prompts you to confirm that it found the correct .NET Aspire *AppHost* project. Select the **Confirm and continue initializing my app** option.

```
Output
```

```
Detected services:
```

```
.NET (Aspire)
Detected in:
D:\source\repos\AspireSample\AspireSample.AppHost\AspireSample.AppHost.csproj
```

```
azd will generate the files necessary to host your app on Azure using
Azure Container Apps.
```

```
? Select an option [Use arrows to move, type to filter]
> Confirm and continue initializing my app
    Cancel and exit
```

5. Enter an environment name, which is used to name provisioned resources in Azure and managing different environments such as `dev` and `prod`.

Output

Generating files to run your app on Azure:

```
(✓) Done: Generating ./azure.yaml  
(✓) Done: Generating ./next-steps.md
```

SUCCESS: Your app is ready for the cloud!

You can provision and deploy your app to Azure by running the `azd up` command in this directory. For more information on configuring your app, see `./next-steps.md`

`azd` generates a number of files and places them into the working directory. These files are:

- `azure.yaml`: Describes the services of the app, such as .NET Aspire AppHost project, and maps them to Azure resources.
- `.azure/config.json`: Configuration file that informs `azd` what the current active environment is.
- `.azure/aspireazddev/.env`: Contains environment specific overrides.

Deploy the template

1. Once an `azd` template is initialized, the provisioning and deployment process can be executed as a single command from the `AppHost` project directory using `azd up`:

Azure Developer CLI

```
azd up
```

2. Select the subscription you'd like to deploy to from the list of available options:

Output

Select an Azure Subscription to use: [Use arrows to move, type to filter]

1. SampleSubscription01 (xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx)
2. SamepleSubscription02 (xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxx)

3. Select the desired Azure location to use from the list of available options:

Output

```
Select an Azure location to use: [Use arrows to move, type to filter]
  42. (US) Central US (centralus)
  43. (US) East US (eastus)
> 44. (US) East US 2 (eastus2)
  46. (US) North Central US (northcentralus)
  47. (US) South Central US (southcentralus)
```

After you make your selections, `azd` executes the provisioning and deployment process.

Output

By default, a service can only be reached from inside the Azure Container Apps environment it is running in. Selecting a service here will also allow it to be reached from the Internet.

```
? Select which services to expose to the Internet webfrontend
? Select an Azure Subscription to use: 1. <YOUR SUBSCRIPTION>
? Select an Azure location to use: 1. <YOUR LOCATION>
```

Packaging services (azd package)

SUCCESS: Your application was packaged for Azure in less than a second.

Provisioning Azure resources (azd provision)

Provisioning Azure resources can take some time.

Subscription: <YOUR SUBSCRIPTION>

Location: <YOUR LOCATION>

You can view detailed progress in the Azure Portal:
[<LINK TO DEPLOYMENT>](#)

- (✓) Done: Resource group: <YOUR RESOURCE GROUP>
- (✓) Done: Container Registry: <ID>
- (✓) Done: Log Analytics workspace: <ID>
- (✓) Done: Container Apps Environment: <ID>

SUCCESS: Your application was provisioned in Azure in 1 minute 13 seconds.
You can view the resources created under the resource group <YOUR RESOURCE GROUP> in Azure Portal:

[<LINK TO RESOURCE GROUP OVERVIEW>](#)

Deploying services (azd deploy)

- (✓) Done: Deploying service apiservice
 - Endpoint: <YOUR UNIQUE apiservice APP>.azurecontainerapps.io/
- (✓) Done: Deploying service webfrontend
 - Endpoint: <YOUR UNIQUE webfrontend APP>.azurecontainerapps.io/

SUCCESS: Your application was deployed to Azure in 1 minute 39 seconds.

You can view the resources created under the resource group <YOUR RESOURCE GROUP> in Azure Portal:

<LINK TO RESOURCE GROUP OVERVIEW>

SUCCESS: Your up workflow to provision and deploy to Azure completed in 3 minutes 50 seconds.

The `azd up` command acts as wrapper for the following individual `azd` commands to provision and deploy your resources in a single step:

1. **azd package**: The app projects and their dependencies are packaged into containers.
2. **azd provision**: The Azure resources the app will need are provisioned.
3. **azd deploy**: The projects are pushed as containers into an Azure Container Registry instance, and then used to create new revisions of Azure Container Apps in which the code will be hosted.

When the `azd up` stages complete, your app will be available on Azure, and you can open the Azure portal to explore the resources. `azd` also outputs URLs to access the deployed apps directly.

Azure SQL Database

Name	Type
acrwmem23cepmjpy	Container registry
apiservice	Container App
cae-wmem23cepmjpy	Container Apps Environment
law-wmem23cepmjpy	Log Analytics workspace
mi-wmem23cepmjpy	Managed Identity
sqldb (sqlserver-wmem23cepmjpy/sqldb)	SQL database
sqlserver-wmem23cepmjpy	SQL server
webfrontend	Container App

Clean up resources

Run the following Azure CLI command to delete the resource group when you no longer need the Azure resources you created. Deleting the resource group also deletes the resources contained inside of it.

```
az group delete --name <your-resource-group-name>
```

For more information, see [Clean up resources in Azure](#).

See also

- [.NET Aspire deployment via Azure Container Apps](#)
- [.NET Aspire Azure Container Apps deployment deep dive](#)
- [Deploy a .NET Aspire project using GitHub Actions](#)

Tutorial: Deploy a .NET Aspire project with a Redis Cache to Azure

Article • 08/29/2024

In this tutorial, you learn to configure a .NET Aspire project with a Redis Cache for deployment to Azure. .NET Aspire provides multiple caching integration configurations that provision different Redis services in Azure. You'll learn how to:

- ✓ Configure the app to provision an Azure Cache for Redis
- ✓ Configure the app to provision a containerized Redis Cache

(!) Note

This document focuses specifically on .NET Aspire configurations to provision and deploy Redis Cache resources in Azure. For more information and to learn more about the full .NET Aspire deployment process, see the [Azure Container Apps deployment](#) tutorial.

Prerequisites

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

- [.NET 8.0](#)
- .NET Aspire workload:
 - Installed with the [Visual Studio installer](#) or the [.NET CLI workload](#).
- An OCI compliant container runtime, such as:
 - [Docker Desktop](#) or [Podman](#).
- An Integrated Developer Environment (IDE) or code editor, such as:
 - [Visual Studio 2022](#) version 17.10 or higher (Optional)
 - [Visual Studio Code](#) (Optional)
 - [C# Dev Kit: Extension](#) (Optional)

For more information, see [.NET Aspire setup and tooling](#).

Create the sample solution

Follow the [Tutorial: Implement caching with .NET Aspire integrations](#) to create the sample project.

Configure the app for Redis cache deployment

.NET Aspire provides two built-in configuration options to streamline Redis Cache deployment on Azure:

- Provision a containerized Redis Cache using Azure Container Apps
- Provision an Azure Cache for Redis instance

Add the .NET Aspire integration to the app

Add the appropriate .NET Aspire integration to the *AspireRedis.AppHost* project for your desired hosting service.

Azure Cache for Redis

Add the [Aspire.Hosting.Azure.Redis](#) package to the *AspireRedis.AppHost* project:

.NET CLI

```
dotnet add package Aspire.Hosting.Azure.Redis
```

Configure the AppHost project

Configure the *AspireRedis.AppHost* project for your desired Redis service.

Azure Cache for Redis

Replace the contents of the *Program.cs* file in the *AspireRedis.AppHost* project with the following code:

C#

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

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

var apiService = builder.AddProject<Projects.AspireRedis_ApiService>
("apiservice")
    .WithReference(cache);

builder.AddProject<Projects.AspireRedis_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithReference(cache)
```

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

The preceding code adds an Azure Cache for Redis resource to your app and configures a connection called `cache`. The `PublishAsAzureRedis` method ensures that tools such as the Azure Developer CLI or Visual Studio create an Azure Cache for Redis resource during the deployment process.

Deploy the app

Tools such as the [Azure Developer CLI](#) (`azd`) support .NET Aspire Redis integration configurations to streamline deployments. `azd` consumes these settings and provisions properly configured resources for you.

(!) Note

You can also use the [Azure CLI](#) or [Bicep](#) to provision and deploy .NET Aspire project resources. These options require more manual steps, but provide more granular control over your deployments. .NET Aspire projects can also connect to an existing Redis instance through manual configurations.

1. Open a terminal window in the root of your .NET Aspire project.
2. Run the `azd init` command to initialize the project with `azd`.

```
Azure Developer CLI
```

```
azd init
```

3. When prompted for an environment name, enter `docs-aspireredis`.
4. Run the `azd up` command to begin the deployment process:

```
Azure Developer CLI
```

```
azd up
```

5. Select the Azure subscription that should host your app resources.
6. Select the Azure location to use.

The Azure Developer CLI provisions and deploys your app resources. The process may take a few minutes to complete.

- When the deployment finishes, click the resource group link in the output to view the created resources in the Azure portal.

Azure Cache for Redis

The deployment process provisioned an Azure Cache for Redis resource due to the **.AppHost** configuration you provided.

Name	Type
acrjzqx7narm4mzm	Container registry
apiservice	Container App
cachejzqx7narm4mzm	Azure Cache for Redis
cae-jzqx7narm4mzm	Container Apps Environment
kv265dafe5jzqx7narm4mzm	Key vault
law-jzqx7narm4mzm	Log Analytics workspace

Clean up resources

Run the following Azure CLI command to delete the resource group when you no longer need the Azure resources you created. Deleting the resource group also deletes the resources contained inside of it.

Azure CLI

```
az group delete --name <your-resource-group-name>
```

For more information, see [Clean up resources in Azure](#).

See also

- [.NET Aspire deployment via Azure Container Apps](#)
- [.NET Aspire Azure Container Apps deployment deep dive](#)
- [Deploy a .NET Aspire project using GitHub Actions](#)

.NET Aspire manifest format for deployment tool builders

Article • 03/29/2024

In this article, you learn about the .NET Aspire manifest format. This article serves as a reference guide for deployment tool builders, aiding in the creation of tooling to deploy .NET Aspire projects on specific hosting platforms, whether on-premises or in the cloud.

.NET Aspire [simplifies the local development experience](#) by helping to manage interdependencies between application integrations. To help simplify the deployment of applications, .NET Aspire projects can generate a manifest of all the resources defined as a JSON formatted file.

Generate a manifest

A valid .NET Aspire project is required to generate a manifest. To get started, create a .NET Aspire project using the `aspire-starter` .NET template:

.NET CLI

```
dotnet new aspire-starter --use-redis-cache `  
  -o AspireApp && `  
  cd AspireApp
```

Manifest generation is achieved by running `dotnet build` with a special target:

.NET CLI

```
dotnet run --project AspireApp.AppHost\AspireApp.AppHost.csproj `  
  -- `  
  --publisher manifest `  
  --output-path ../aspire-manifest.json
```

💡 Tip

The `--output-path` supports relative paths. The previous command uses `../aspire-manifest.json` to place the manifest file in the root of the project directory.

For more information, see [dotnet run](#). The previous command produces the following output:

Output

```
Building...
info: Aspire.Hosting.Publishing.ManifestPublisher[0]
      Published manifest to: .\AspireApp.AppHost\aspire-manifest.json
```

The file generated is the .NET Aspire manifest and is used by tools to support deploying into target cloud environments.

⚠ Note

You can also generate a manifest as part of the launch profile. Consider the following *launchSettings.json*:

JSON

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "generate-manifest": {
      "commandName": "Project",
      "launchBrowser": false,
      "dotnetRunMessages": true,
      "commandLineArgs": "--publisher manifest --output-path aspire-
manifest.json"
    }
  }
}
```

Basic manifest format

Publishing the manifest from the default starter template for .NET Aspire produces the following JSON output:

JSON

```
{
  "resources": {
    "cache": {
      "type": "container.v0",
      "connectionString": "{cache.bindings.tcp.host}:
{cache.bindings.tcp.port}",
      "image": "redis:7.2.4",
      "bindings": {
        "tcp": {
          "scheme": "tcp",
          "host": "127.0.0.1",
          "port": 11211
        }
      }
    }
  }
}
```

```
        "protocol": "tcp",
        "transport": "tcp",
        "containerPort": 6379
    }
}
},
"apiservice": {
    "type": "project.v0",
    "path": "../AspireApp.ApiService/AspireApp.ApiService.csproj",
    "env": {
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EXCEPTION_LOG_ATTRIBUTES": "true",
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EVENT_LOG_ATTRIBUTES": "true"
    },
    "bindings": {
        "http": {
            "scheme": "http",
            "protocol": "tcp",
            "transport": "http"
        },
        "https": {
            "scheme": "https",
            "protocol": "tcp",
            "transport": "http"
        }
    }
},
"webfrontend": {
    "type": "project.v0",
    "path": "../AspireApp.Web/AspireApp.Web.csproj",
    "env": {
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EXCEPTION_LOG_ATTRIBUTES": "true",
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EVENT_LOG_ATTRIBUTES": "true",
        "ConnectionStrings__cache": "{cache.connectionString}",
        "services_apiservice_0": "{apiservice.bindings.http.url}",
        "services_apiservice_1": "{apiservice.bindings.https.url}"
    },
    "bindings": {
        "http": {
            "scheme": "http",
            "protocol": "tcp",
            "transport": "http"
        },
        "https": {
            "scheme": "https",
            "protocol": "tcp",
            "transport": "http"
        }
    }
}
}
```

The manifest format JSON consists of a single object called `resources`, which contains a property for each resource specified in `Program.cs` (the `name` argument for each name is used as the property for each of the child resource objects in JSON).

Connection string and binding references

In the previous example, there are two project resources and one Redis cache resource. The `webfrontend` depends on both the `apiservice` (project) and `cache` (Redis) resources.

This dependency is known because the environment variables for the `webfrontend` contain placeholders that reference the two other resources:

JSON

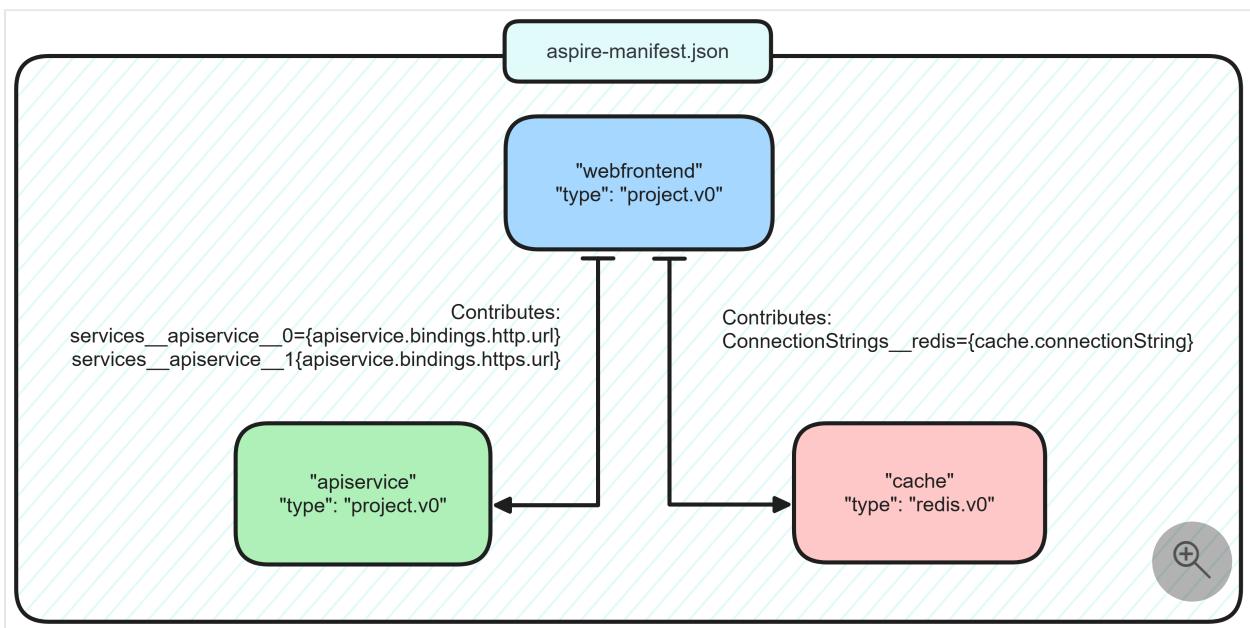
```
"env": {  
    // ... other environment variables omitted for clarity  
    "ConnectionString_cache": "{cache.connectionString}",  
    "services_apiservice_0": "{apiservice.bindings.http.url}",  
    "services_apiservice_1": "{apiservice.bindings.https.url}"  
},
```

The `apiservice` resource is referenced by `webfrontend` using the call `WithReference(apiservice)` in the app host `Program.cs` file and `redis` is referenced using the call `WithReference(cache)`:

C#

```
var builder = DistributedApplication.CreateBuilder(args);  
  
var cache = builder.AddRedis("cache");  
  
var apiService = builder.AddProject<Projects.AspireApp_ApiService>(  
    "apiservice");  
  
builder.AddProject<Projects.AspireApp_Web>("webfrontend")  
    .WithReference(cache)  
    .WithReference(apiService);  
  
builder.Build().Run();
```

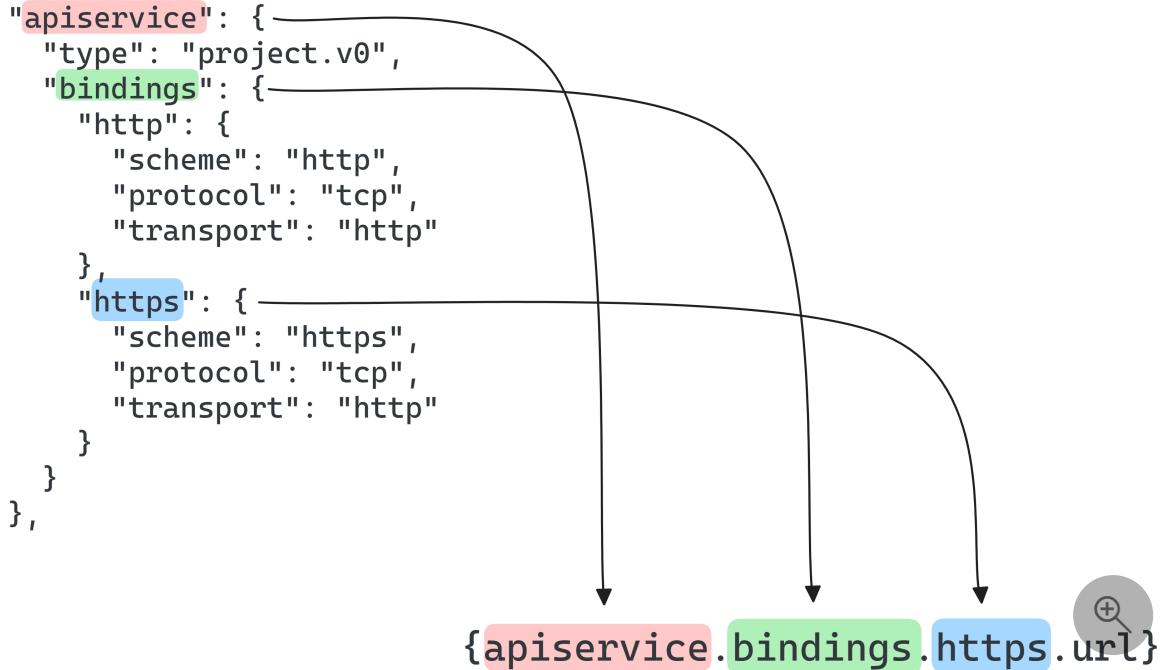
References between project resource types result in [service discovery](#) variables being injected into the referencing project. References to well known reference types such as Redis result in connection strings being injected.



For more information on how resources in the app model and references between them work, see, [.NET Aspire orchestration overview](#).

Placeholder string structure

Placeholder strings reference the structure of the .NET Aspire manifest:



The final segment of the placeholder string (`url` in this case) is generated by the tool processing the manifest. There are several suffixes that could be used on the placeholder string:

- `connectionString`: For well-known resource types such as Redis. Deployment tools translate the resource in the most appropriate infrastructure for the target cloud

environment and then produce a .NET Aspire compatible connection string for the consuming application to use. On `container.v0` resources the `connectionString` field may be present and specified explicitly. This is to support scenarios where a container resource type is referenced using the [WithReference](#) extension but is desired to be hosted explicitly as a container.

- `url`: For service-to-service references where a well-formed URL is required. The deployment tool produces the `url` based on the scheme, protocol, and transport defined in the manifest and the underlying compute/networking topology that was deployed.
- `host`: The host segment of the URL.
- `port`: The port segment of the URL.

Resource types

Each resource has a `type` field. When a deployment tool reads the manifest, it should read the type to verify whether it can correctly process the manifest. During the .NET Aspire preview period, all resource types have a `v0` suffix to indicate that they're subject to change. As .NET Aspire approaches release a `v1` suffix will be used to signify that the structure of the manifest for that resource type should be considered stable (subsequent updates increment the version number accordingly).

Common resource fields

The `type` field is the only field that is common across all resource types, however, the `project.v0`, `container.v0`, and `executable.v0` resource types also share the `env` and `bindings` fields.

⚠ Note

The `executable.v0` resource type isn't fully implemented in the manifest due to its lack of utility in deployment scenarios. For more information on containerizing executables, see [Dockerfile resource types](#).

The `env` field type is a basic key/value mapping where the values might contain [placeholder strings](#).

Bindings are specified in the `bindings` field with each binding contained within its own field under the `bindings` JSON object. The fields omitted by the .NET Aspire manifest in the `bindings` node include:

- `scheme`: One of the following values `tcp`, `udp`, `http`, or `https`.
- `protocol`: One of the following values `tcp` or `udp`
- `transport`: Same as `scheme`, but used to disambiguate between `http` and `http2`.
- `containerPort`: Optional, if omitted defaults to port 80.

The `inputs` field

Some resources generate an `inputs` field. This field is used to specify input parameters for the resource. The `inputs` field is a JSON object where each property is an input parameter that's used in placeholder structure resolution. Resources that have a `connectionString`, for example, might use the `inputs` field to specify a `password` for the connection string:

JSON

```
"connectionString": "Host={<resourceName>.bindings.tcp.host};Port=
{<resourceName>.bindings.tcp.port};Username=admin;Password=
{<resourceName>.inputs.password};"
```

The connection string placeholder references the `password` input parameter from the `inputs` field:

JSON

```
"inputs": {
  "password": {
    "type": "string",
    "secret": true,
    "default": {
      "generate": {
        "minLength": 10
      }
    }
  }
}
```

The preceding JSON snippet shows the `inputs` field for a resource that has a `connectionString` field. The `password` input parameter is a string type and is marked as a secret. The `default` field is used to specify a default value for the input parameter. In this case, the default value is generated using the `generate` field, with random string of a minimum length.

Built-in resources

The following table is a list of resource types that are explicitly generated by .NET Aspire and extensions developed by the .NET Aspire team:

Cloud-agnostic resource types

These resources are available in the [Aspire.Hosting](#) NuGet package.

 Expand table

App model usage	Manifest resource type	Heading link
AddContainer	container.v0	Container resource type
PublishAsDockerFile	dockerfile.v0	Dockerfile resource types
AddDatabase	value.v0	MongoDB Server resource types
AddMongoDB	container.v0	MongoDB resource types
AddDatabase	value.v0	MySQL Server resource types
AddMySql	container.v0	MySQL resource types
AddDatabase	value.v0	Postgres resource types
AddPostgres	container.v0	Postgres resource types
AddProject	project.v0	Project resource type
AddRabbitMQ	container.v0	RabbitMQ resource types
AddRedis	container.v0	Redis resource type
AddDatabase	value.v0	SQL Server resource types
AddSqlServer	container.v0	SQL Server resource types

Project resource type

Example code:

C#

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

Example manifest:

JSON

```
"apiservice": {
    "type": "project.v0",
    "path": "../AspireApp.ApiService/AspireApp.ApiService.csproj",
    "env": {
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EXCEPTION_LOG_ATTRIBUTES": "true",
        "OTEL_DOTNET_EXPERIMENTAL_OTLP_EMIT_EVENT_LOG_ATTRIBUTES": "true"
    },
    "bindings": {
        "http": {
            "scheme": "http",
            "protocol": "tcp",
            "transport": "http"
        },
        "https": {
            "scheme": "https",
            "protocol": "tcp",
            "transport": "http"
        }
    }
}
```

Container resource type

Example code:

C#

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

builder.AddContainer("mycontainer", "myimage")
    .WithEnvironment("LOG_LEVEL", "WARN")
    .WithHttpEndpoint(3000);
```

Example manifest:

JSON

```
{
    "resources": {
        "mycontainer": {
            "type": "container.v0",
            "image": "myimage:latest",
            "env": {
                "LOG_LEVEL": "WARN"
            },
            "bindings": {
                "http": {
                    "scheme": "http",
                    "port": 3000
                }
            }
        }
    }
}
```

```
        "protocol": "tcp",
        "transport": "http",
        "containerPort": 3000
    }
}
}
}
```

Dockerfile resource types

Example code:

```
C#  
  
var builder = DistributedApplication.CreateBuilder(args);  
  
builder.AddNodeApp("nodeapp", "../nodeapp/app.js")
    .WithHttpEndpoint(hostPort: 5031, env: "PORT")
    .PublishAsDockerFile();
```

💡 Tip

The `PublishAsDockerFile` call is required to generate the Dockerfile resource type in the manifest, and this extension method is only available on the [ExecutableResource](#) type.

Example manifest:

JSON

```
{  
  "resources": {  
    "nodeapp": {  
      "type": "dockerfile.v0",  
      "path": "../nodeapp/Dockerfile",  
      "context": "../nodeapp",  
      "env": {  
        "NODE_ENV": "development",  
        "PORT": "{nodeapp.bindings.http.port}"  
      },  
      "bindings": {  
        "http": {  
          "scheme": "http",  
          "protocol": "tcp",  
          "transport": "http",  
          "containerPort": 5031  
        }  
      }  
    }  
  }  
}
```

```
        }
    }
}
}
```

Postgres resource types

Example code:

```
C#  
  
var builder = DistributedApplication.CreateBuilder(args);  
  
builder.AddPostgres("postgres1")  
    .AddDatabase("shipping");
```

Example manifest:

JSON

```
{  
  "resources": {  
    "postgres1": {  
      "type": "container.v0",  
      "connectionString": "Host={postgres1.bindings.tcp.host};Port={postgres1.bindings.tcp.port};Username=postgres;Password={postgres1.inputs.password}",  
      "image": "postgres:16.2",  
      "env": {  
        "POSTGRES_HOST_AUTH_METHOD": "scram-sha-256",  
        "POSTGRES_INITDB_ARGS": "--auth-host=scram-sha-256 --auth-local=scram-sha-256",  
        "POSTGRES_PASSWORD": "{postgres1.inputs.password}"  
      },  
      "bindings": {  
        "tcp": {  
          "scheme": "tcp",  
          "protocol": "tcp",  
          "transport": "tcp",  
          "containerPort": 5432  
        }  
      },  
      "inputs": {  
        "password": {  
          "type": "string",  
          "secret": true,  
          "default": {  
            "generate": {  
              "minLength": 10
```

```
        }
    }
}
},
"shipping": {
    "type": "value.v0",
    "connectionString": "{postgres1.connectionString};Database=shipping"
}
}
}
```

RabbitMQ resource types

RabbitMQ is modeled as a container resource `container.v0`. The following sample shows how they're added to the app model.

C#

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

builder.AddRabbitMQ("rabbitmq1");
```

The previous code produces the following manifest:

JSON

```
{
  "resources": {
    "rabbitmq1": {
      "type": "container.v0",
      "connectionString": "amqp://guest:{rabbitmq1.inputs.password}@{rabbitmq1.bindings.tcp.host}:{rabbitmq1.bindings.tcp.port}",
      "image": "rabbitmq:3",
      "env": {
        "RABBITMQ_DEFAULT_USER": "guest",
        "RABBITMQ_DEFAULT_PASS": "{rabbitmq1.inputs.password}"
      },
      "bindings": {
        "tcp": {
          "scheme": "tcp",
          "protocol": "tcp",
          "transport": "tcp",
          "containerPort": 5672
        }
      },
      "inputs": {
        "password": {
          "type": "string",

```

```
        "secret": true,
        "default": {
            "generate": {
                "minLength": 10
            }
        }
    }
}
```

Redis resource type

Example code:

```
C#  
  
var builder = DistributedApplication.CreateBuilder(args);  
  
builder.AddRedis("redis1");
```

Example manifest:

```
JSON  
  
{  
    "resources": {  
        "redis1": {  
            "type": "container.v0",  
            "connectionString": "{redis1.bindings.tcp.host}:  
{redis1.bindings.tcp.port}",  
            "image": "redis:7.2.4",  
            "bindings": {  
                "tcp": {  
                    "scheme": "tcp",  
                    "protocol": "tcp",  
                    "transport": "tcp",  
                    "containerPort": 6379  
                }  
            }  
        }  
    }  
}
```

SQL Server resource types

Example code:

C#

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

builder.AddSqlServer("sql1")
    .AddDatabase("shipping");
```

Example manifest:

JSON

```
{
  "resources": {
    "sql1": {
      "type": "container.v0",
      "connectionString": "Server={sql1.bindings.tcp.host}, {sql1.bindings.tcp.port};User ID=sa;Password={sql1.inputs.password};TrustServerCertificate=true",
      "image": "mcr.microsoft.com/mssql/server:2022-latest",
      "env": {
        "ACCEPT_EULA": "Y",
        "MSSQL_SA_PASSWORD": "{sql1.inputs.password}"
      },
      "bindings": {
        "tcp": {
          "scheme": "tcp",
          "protocol": "tcp",
          "transport": "tcp",
          "containerPort": 1433
        }
      },
      "inputs": {
        "password": {
          "type": "string",
          "secret": true,
          "default": {
            "generate": {
              "minLength": 10
            }
          }
        }
      }
    },
    "shipping": {
      "type": "value.v0",
      "connectionString": "{sql1.connectionString};Database=shipping"
    }
  }
}
```

MongoDB resource types

Example code:

```
C#  
  
var builder = DistributedApplication.CreateBuilder(args);  
  
builder.AddMongoDB("mongodb1")  
    .AddDatabase("shipping");
```

Example manifest:

```
JSON  
  
{  
    "resources": {  
        "mongodb1": {  
            "type": "container.v0",  
            "connectionString": "mongodb://{{mongodb1.bindings.tcp.host}}:  
{{mongodb1.bindings.tcp.port}}",  
            "image": "mongo:7.0.5",  
            "bindings": {  
                "tcp": {  
                    "scheme": "tcp",  
                    "protocol": "tcp",  
                    "transport": "tcp",  
                    "containerPort": 27017  
                }  
            }  
        },  
        "shipping": {  
            "type": "value.v0",  
            "connectionString": "{{mongodb1.connectionString}}/shipping"  
        }  
    }  
}
```

MySQL resource types

Example code:

```
C#  
  
var builder = DistributedApplication.CreateBuilder(args);  
  
builder.AddMySql("mysql1")  
    .AddDatabase("shipping");
```

Example manifest:

JSON

```
{  
  "resources": {  
    "mysql1": {  
      "type": "container.v0",  
      "connectionString": "Server={mysql1.bindings.tcp.host};Port={mysql1.bindings.tcp.port};User ID=root;Password={mysql1.inputs.password}",  
      "image": "mysql:8.3.0",  
      "env": {  
        "MYSQL_ROOT_PASSWORD": "{mysql1.inputs.password}"  
      },  
      "bindings": {  
        "tcp": {  
          "scheme": "tcp",  
          "protocol": "tcp",  
          "transport": "tcp",  
          "containerPort": 3306  
        }  
      },  
      "inputs": {  
        "password": {  
          "type": "string",  
          "secret": true,  
          "default": {  
            "generate": {  
              "minLength": 10  
            }  
          }  
        }  
      },  
      "shipping": {  
        "type": "value.v0",  
        "connectionString": "{mysql1.connectionString};Database=shipping"  
      }  
    }  
  }  
}
```

Azure-specific resource types

The following resources are available in the [Aspire.Hosting.Azure](#) NuGet package.

 Expand table

App Model usage	Manifest resource type	Heading link
AddAzureAppConfiguration	azure.bicep.v0	Azure App Configuration resource types
AddAzureKeyVault	azure.bicep.v0	Azure Key Vault resource type
AddRedis .AsAzureRedis()	azure.bicep.v0	Azure Redis resource types
AddAzureServiceBus	azure.bicep.v0	Azure Service Bus resource type
AddSqlServer .AsAzureSqlDatabase()	azure.bicep.v0	Azure SQL resource types
AddSqlServer .AsAzureSqlDatabase().AddDatabase(...)	value.v0	Azure SQL resource types
AddPostgres .AsAzurePostgresFlexibleServer(...)	azure.bicep.v0	Azure Postgres resource types
AddPostgres .AsAzurePostgresFlexibleServer(...).AddDatabase(...)	value.v0	Azure Postgres resource types
AddAzureStorage	azure.storage.v0	Azure Storage resource types
AddBlobs	value.v0	Azure Storage resource types
AddQueues	value.v0	Azure Storage

App Model usage	Manifest resource type	Heading link
		resource types
AddTables	value.v0	Azure Storage resource types

Azure Key Vault resource type

Example code:

```
C#  
  
var builder = DistributedApplication.CreateBuilder(args);  
  
builder.AddAzureKeyVault("keyvault1");
```

Example manifest:

```
JSON  
  
{  
  "resources": {  
    "keyvault1": {  
      "type": "azure.bicep.v0",  
      "connectionString": "{keyvault1.outputs.vaultUri}",  
      "path": "aspire.hosting.azure.bicep.keyvault.bicep",  
      "params": {  
        "principalId": "",  
        "principalType": "",  
        "vaultName": "keyvault1"  
      }  
    }  
  }  
}
```

Azure Service Bus resource type

Example code:

```
C#
```

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

builder.AddAzureServiceBus("sb1")
    .AddTopic("topic1", [])
    .AddTopic("topic2", [])
    .AddQueue("queue1")
    .AddQueue("queue2");
```

Example manifest:

JSON

```
{
  "resources": {
    "sb1": {
      "type": "azure.bicep.v0",
      "connectionString": "{sb1.outputs.serviceBusEndpoint}",
      "path": "aspire.hosting.azure.bicep.servicebus.bicep",
      "params": {
        "serviceBusNamespaceName": "sb1",
        "principalId": "",
        "principalType": "",
        "queues": [
          "queue1",
          "queue2"
        ],
        "topics": [
          {
            "name": "topic1",
            "subscriptions": []
          },
          {
            "name": "topic2",
            "subscriptions": []
          }
        ]
      }
    }
  }
}
```

Azure Storage resource types

Example code:

C#

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

```
var storage = builder.AddAzureStorage("images");

storage.AddBlobs("blobs");
storage.AddQueues("queues");
storage.AddTables("tables");
```

Example manifest:

JSON

```
{
  "resources": {
    "images": {
      "type": "azure.bicep.v0",
      "path": "aspire.hosting.azure.bicep.storage.bicep",
      "params": {
        "principalId": "",
        "principalType": "",
        "storageName": "images"
      }
    },
    "blobs": {
      "type": "value.v0",
      "connectionString": "{images.outputs.blobEndpoint}"
    },
    "queues": {
      "type": "value.v0",
      "connectionString": "{images.outputs.queueEndpoint}"
    },
    "tables": {
      "type": "value.v0",
      "connectionString": "{images.outputs.tableEndpoint}"
    }
  }
}
```

Azure Redis resource type

Example code:

C#

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

builder.AddRedis("azredis1")
  .PublishAsAzureRedis();
```

Example manifest:

JSON

```
{  
  "resources": {  
    "azredis1": {  
      "type": "azure.bicep.v0",  
      "connectionString": "{azredis1.secretOutputs.connectionString}",  
      "path": "aspire.hosting.azure.bicep.redis.bicep",  
      "params": {  
        "redisCacheName": "azredis1",  
        "keyVaultName": ""  
      }  
    }  
  }  
}
```

Azure App Configuration resource type

Example code:

C#

```
var builder = DistributedApplication.CreateBuilder(args);  
  
builder.AddAzureAppConfiguration("appconfig1");
```

Example manifest:

JSON

```
{  
  "resources": {  
    "appconfig1": {  
      "type": "azure.bicep.v0",  
      "connectionString": "{appconfig1.outputs.appConfigEndpoint}",  
      "path": "aspire.hosting.azure.bicep.appconfig.bicep",  
      "params": {  
        "configName": "appconfig1",  
        "principalId": "",  
        "principalType": ""  
      }  
    }  
  }  
}
```

Azure SQL resource types

Example code:

C#

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

builder.AddSqlServer("sql1")
    .PublishAsAzureSqlDatabase()
    .AddDatabase("inventory");
```

Example manifest:

JSON

```
{
  "resources": {
    "sql1": {
      "type": "azure.bicep.v0",
      "connectionString": "Server=tcp:{sql1.outputs.sqlServerFqdn},1433;Encrypt=True;Authentication=\u0022Active Directory Default\u0022",
      "path": "aspire.hosting.azure.bicep.sql.bicep",
      "params": {
        "serverName": "sql1",
        "principalId": "",
        "principalName": "",
        "databases": [
          "inventory"
        ]
      }
    },
    "inventory": {
      "type": "value.v0",
      "connectionString": "{sql1.connectionString};Database=inventory"
    }
  }
}
```

Azure Postgres resource types

Example code:

C#

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

var administratorLogin = builder.AddParameter("administratorLogin");
var administratorLoginPassword = builder.AddParameter(
    "administratorLoginPassword", secret: true);

var pg = builder.AddPostgres("postgres")
    .PublishAsAzurePostgresFlexibleServer()
```

```
    administratorLogin, administratorLoginPassword)
    .AddDatabase("db");
```

Example manifest:

JSON

```
{
  "resources": {
    "administratorLogin": {
      "type": "parameter.v0",
      "value": "{administratorLogin.inputs.value}",
      "inputs": {
        "value": {
          "type": "string"
        }
      }
    },
    "administratorLoginPassword": {
      "type": "parameter.v0",
      "value": "{administratorLoginPassword.inputs.value}",
      "inputs": {
        "value": {
          "type": "string",
          "secret": true
        }
      }
    },
    "postgres": {
      "type": "azure.bicep.v0",
      "connectionString": "{postgres.secretOutputs.connectionString}",
      "path": "aspire.hosting.azure.bicep.postgres.bicep",
      "params": {
        "serverName": "postgres",
        "keyVaultName": "",
        "administratorLogin": "{administratorLogin.value}",
        "administratorLoginPassword": "{administratorLoginPassword.value}",
        "databases": [
          "db"
        ]
      }
    },
    "db": {
      "type": "value.v0",
      "connectionString": "{postgres.connectionString};Database=db"
    }
  }
}
```

Resource types supported in the Azure Developer CLI

The [Azure Developer CLI](#) (azd) is a tool that can be used to deploy .NET Aspire projects to Azure Container Apps. With the `azure.bicep.v0` resource type, cloud-agnostic resource container types can be mapped to Azure-specific resources. The following table lists the resource types that are supported in the Azure Developer CLI:

 [Expand table](#)

Name	Cloud-agnostic API	Configure as Azure resource
Redis	AddRedis	<code>PublishAsAzureRedis</code>
Postgres	AddPostgres	<code>PublishAsAzurePostgresFlexibleServer</code>
SQL Server	AddSqlServer	<code>PublishAsAzureSqlDatabase</code>

When resources are configured as Azure resources, the `azure.bicep.v0` resource type is generated in the manifest. For more information, see [Deploy a .NET Aspire project to Azure Container Apps using the Azure Developer CLI \(in-depth guide\)](#).

See also

- [.NET Aspire overview](#)
- [.NET Aspire orchestration overview](#)
- [.NET Aspire integrations overview](#)
- [Service discovery in .NET Aspire](#)

Local Azure provisioning

Article • 09/23/2024

.NET Aspire simplifies local cloud-native app development with its compelling app host model. This model allows you to run your app locally with the same configuration and services as in Azure.

In this article you learn how to provision Azure resources from your local development environment through the [.NET Aspire app host](#). All of this is possible with the help of the `Azure.Provisioning.*` libraries, which provide a set of APIs to provision Azure resources. These packages are transitive dependencies of the .NET Aspire Azure hosting libraries you use in your app host, so you don't need to install them separately.

Requirements

This article assumes that you have an Azure account and subscription. If you don't have an Azure account, you can create a free one at [Azure Free Account](#). For provisioning functionality to work correctly, you'll need to be authenticated with Azure. Additionally, you'll need to provide some configuration values so that the provisioning logic can create resources on your behalf.

App host provisioning APIs

The app host provides a set of APIs to express Azure resources. These APIs are available as extension methods in .NET Aspire Azure hosting libraries, extending the `IDistributedApplicationBuilder` interface. When you add Azure resources to your app host, they'll add the appropriate provisioning functionality implicitly. In other words, you don't need to call any provisioning APIs directly.

When the app host starts, the following provisioning logic is executed:

1. The `Azure` configuration section is validated.
2. When invalid the dashboard and app host output provides hints as to what's missing. For more information, see [Missing configuration value hints](#).
3. When valid Azure resources are conditionally provisioned:
 - a. If an Azure deployment for a given resource doesn't exist, it's created and configured as a deployment.
 - b. The configuration of said deployment is stamped with a checksum as a means to support only provisioning resources as necessary.

Use existing Azure resources

The app host automatically manages provisioning of Azure resources. The first time the app host runs, it provisions the resources specified in the app host. Subsequent runs don't provision the resources again unless the app host configuration changes.

If you've already provisioned Azure resources outside of the app host and want to use them, you can provide the connection string with the `AddConnectionString` API as shown in the following Azure Key Vault example:

```
C#  
  
// Service registration  
var secrets = builder.ExecutionContext.IsPublishMode  
    ? builder.AddAzureKeyVault("secrets")  
    : builder.AddConnectionString("secrets");  
  
// Service consumption  
builder.AddProject<Projects.ExampleProject>()  
    .WithReference(secrets)
```

The preceding code snippet shows how to add an Azure Key Vault to the app host. The `AddAzureKeyVault` API is used to add the Azure Key Vault to the app host. The `AddConnectionString` API is used to provide the connection string to the app host.

Alternatively, for some Azure resources, you can opt-in to running them as an emulator with the `RunAsEmulator` API. This API is available for Azure Cosmos DB and Azure Storage. For example, to run Azure Cosmos DB as an emulator, you can use the following code snippet:

```
C#  
  
var cosmos = builder.AddAzureCosmosDB("cosmos")  
    .RunAsEmulator();
```

The `RunAsEmulator` API configures an Azure Cosmos DB resource to be emulated using the Azure Cosmos DB emulator with the NoSQL API.

.NET Aspire Azure hosting integrations

If you're using Azure resources in your app host, you're using one or more of the .NET Aspire Azure hosting integrations. These hosting libraries provide extension methods to the `IDistributedApplicationBuilder` interface to add Azure resources to your app host.

Azure provisioning integrations

The following Azure provisioning libraries are available:

- [!\[\]\(027b3efe6549ced83ee3603e87e8840d_img.jpg\) Azure.Provisioning.AppConfiguration ↗](#)
- [!\[\]\(a723972a0c0fe6651551a1ad115892f2_img.jpg\) Azure.Provisioning.ApplicationInsights ↗](#)
- [!\[\]\(eb0f0de920f2b308a76ae3117dbc93c0_img.jpg\) Azure.Provisioning.CognitiveServices ↗](#)
- [!\[\]\(04d3df2257e9b915469af5a5393e87c4_img.jpg\) Azure.Provisioning.CosmosDB ↗](#)
- [!\[\]\(7609b65a451384a4ff9b62a03c9c1f01_img.jpg\) Azure.Provisioning.EventHubs ↗](#)
- [!\[\]\(559b1bd4a78edeee0161dfb8f44dd982_img.jpg\) Azure.Provisioning.KeyVault ↗](#)
- [!\[\]\(d7e07789e3554504c0aa6d83b438bfcb_img.jpg\) Azure.Provisioning.OperationalInsights ↗](#)
- [!\[\]\(6470fe1b2968cf4300cf22856d3519ac_img.jpg\) Azure.Provisioning.PostgreSQL ↗](#)
- [!\[\]\(33ad9c6e0669e4133b2a52d8d79d63ec_img.jpg\) Azure.Provisioning.Redis ↗](#)
- [!\[\]\(d2df2e840bf104cf04de65299ce87fc5_img.jpg\) Azure.Provisioning.Resources ↗](#)
- [!\[\]\(b502468b10326d0343399315d33cfbce_img.jpg\) Azure.Provisioning.Search ↗](#)
- [!\[\]\(71485d5d49d519f62048f178140b738b_img.jpg\) Azure.Provisioning.ServiceBus ↗](#)
- [!\[\]\(019d460a963a2c9a4d545dae0f373d11_img.jpg\) Azure.Provisioning.SignalR ↗](#)
- [!\[\]\(f02fd2a57ab2117eacbf2e5e80c547cf_img.jpg\) Azure.Provisioning.Sql ↗](#)
- [!\[\]\(9ae4087991333d95c424ec8903df2f64_img.jpg\) Azure.Provisioning.Storage ↗](#)
- [!\[\]\(731c3165e5c809816992c9b7d12484d3_img.jpg\) Azure.Provisioning ↗](#)

Tip

You shouldn't need to install these packages manually in your projects, as they're transitive dependencies of the corresponding .NET Aspire Azure hosting libraries.

Configuration

When utilizing Azure resources in your local development environment, you need to provide the necessary configuration values. Configuration values are specified under the `Azure` section:

- `SubscriptionId`: The Azure subscription ID.
- `AllowResourceGroupCreation`: A boolean value that indicates whether to create a new resource group.
- `ResourceGroup`: The name of the resource group to use.
- `Location`: The Azure region to use.

Consider the following example `appsettings.json` configuration:

JSON

```
{  
  "Azure": {  
    "SubscriptionId": "<Your subscription id>",  
    "AllowResourceGroupCreation": true,  
    "ResourceGroup": "<Valid resource group name>",  
    "Location": "<Valid Azure location>"  
  }  
}
```

ⓘ Important

It's recommended to store these values as app secrets. For more information, see [Manage app secrets](#).

After you've configured the necessary values, you can start provisioning Azure resources in your local development environment.

Azure provisioning credential store

The .NET Aspire app host uses a credential store for Azure resource authentication and authorization. Depending on your subscription, the correct credential store may be needed for multi-tenant provisioning scenarios.

With the [Aspire.Hosting.Azure](#) NuGet package installed, and if your app host depends on Azure resources, the default Azure credential store relies on the [DefaultAzureCredential](#). To change this behavior, you can set the credential store value in the *appsettings.json* file, as shown in the following example:

JSON

```
{  
  "Azure": {  
    "CredentialSource": "AzureCli"  
  }  
}
```

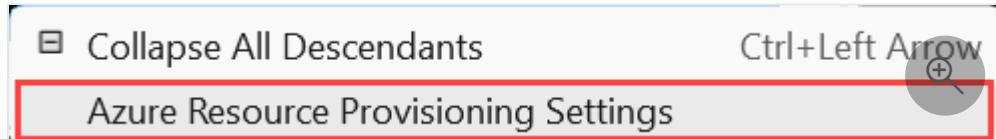
As with all [configuration-based settings](#), you can configure these with alternative providers, such as [user secrets](#) or [environment variables](#). The `Azure:CredentialSource` value can be set to one of the following values:

- `AzureCli`: Delegates to the [AzureCliCredential](#).
- `AzurePowerShell`: Delegates to the [AzurePowerShellCredential](#).
- `VisualStudio`: Delegates to the [VisualStudioCredential](#).

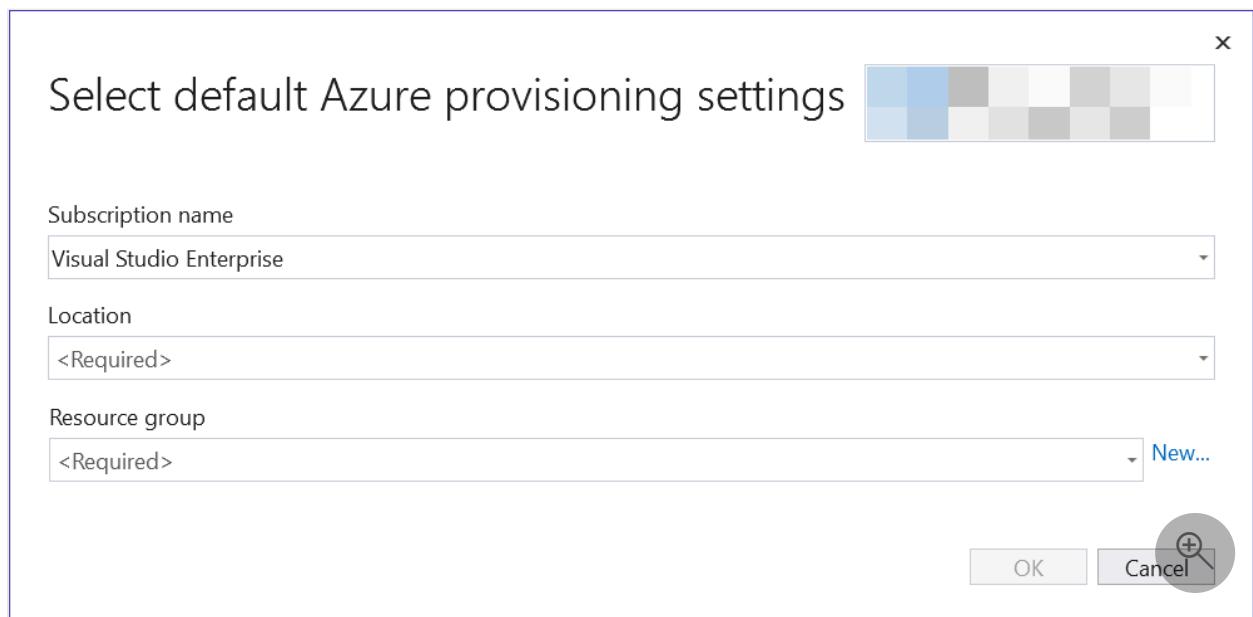
- `VisualStudioCode`: Delegates to the [VisualStudioCodeCredential](#).
- `AzureDeveloperCli`: Delegates to the [AzureDeveloperCliCredential](#).
- `InteractiveBrowser`: Delegates to the [InteractiveBrowserCredential](#).

Tooling support

In Visual Studio, you can use Connected Services to configure the default Azure provisioning settings. Select the app host project, right-click on the **Connected Services** node, and select **Azure Resource Provisioning Settings**:



This will open a dialog where you can configure the Azure provisioning settings, as shown in the following screenshot:



Missing configuration value hints

When the `Azure` configuration section is missing, has missing values, or is invalid, the [.NET Aspire dashboard](#) provides useful hints. For example, consider an app host that's missing the `SubscriptionId` configuration value that's attempting to use an Azure Key Vault resource. The **Resources** page indicates the **State** as **Missing subscription configuration**:

Resources

Type	Name	State
AzureKeyVaultResource	kv1	! Missing subscription configuration
Project	apiservice	✓ Running

Additionally, the **Console logs** display this information as well, consider the following screenshot:

Console logs

kv1 (Missing subscription configuration) ▾ Watching logs...

```
1 Provisioning kv1...
2 stderr Resource could not be provisioned because Azure subscription, location, and
   oning for more details.
```



Known limitations

After provisioning Azure resources in this way, you must manually clean up the resources in the Azure portal as .NET Aspire doesn't provide a built-in mechanism to delete Azure resources. The easiest way to achieve this is by deleting the configured resource group. This can be done in the [Azure portal](#) or by using the Azure CLI:

Azure CLI

```
az group delete --name <ResourceGroupName>
```

Replace `<ResourceGroupName>` with the name of the resource group you want to delete. For more information, see [az group delete](#).

Use custom Bicep templates

Article • 06/13/2024

When you're targeting Azure as your desired cloud provider, you can use Bicep to define your infrastructure as code. [Bicep is a domain-specific language \(DSL\)](#) for deploying Azure resources declaratively. It aims to drastically simplify the authoring experience with a cleaner syntax and better support for modularity and code reuse.

While .NET Aspire provides a set of pre-built Bicep templates so that you don't need to write them, there might be times when you either want to customize the templates or create your own. This article explains the concepts and corresponding APIs that you can use to customize the Bicep templates.

Important

This article is not intended to teach Bicep, but rather to provide guidance on how to create customize Bicep templates for use with .NET Aspire.

As part of the Azure deployment story for .NET Aspire, the Azure Developer CLI (`azd`) provides an understanding of your .NET Aspire project and the ability to deploy it to Azure. The `azd` CLI uses the Bicep templates to deploy the application to Azure.

Install App Host package

To use any of this functionality, you must install the [Aspire.Hosting.Azure](#) NuGet package:

.NET CLI

.NET CLI

```
dotnet add package Aspire.Hosting.Azure
```

For more information, see [dotnet add package](#) or [Manage package dependencies in .NET applications](#).

All of the examples in this article assume that you've installed the `Aspire.Hosting.Azure` package and imported the `Aspire.Hosting.Azure` namespace. Additionally, the examples assume you've created an `IDistributedApplicationBuilder` instance:

C#

```
using Aspire.Hosting.Azure;
```

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

// Examples go here...

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

💡 Tip

By default, when you call any of the Bicep-related APIs, a call is also made to [AddAzureProvisioning](#) that adds support for generating Azure resources dynamically during application startup.

Reference Bicep files

Imagine that you've defined a Bicep template in a file named `storage.bicep` that provisions an Azure Storage Account:

```
Bicep

param location string = resourceGroup().location
param storageAccountName string = 'toylaunch${uniqueString(resourceGroup().id)}'

resource storageAccount 'Microsoft.Storage/storageAccounts@2021-06-01' = {
    name: storageAccountName
    location: location
    sku: {
        name: 'Standard_LRS'
    }
    kind: 'StorageV2'
    properties: {
        accessTier: 'Hot'
    }
}
```

To add a reference to the Bicep file on disk, call the [AddBicepTemplate](#) method. Consider the following example:

```
C#

builder.AddBicepTemplate(
    name: "storage",
    bicepFile: "../infra/storage.bicep");
```

The preceding code adds a reference to a Bicep file located at `../infra/storage.bicep`. The file paths should be relative to the *app host* project. This reference results in an [AzureBicepResource](#) being added to the application's resources collection with the `"storage"` name, and the API returns an `IResourceBuilder<AzureBicepResource>` instance that can be used to further customize the resource.

Reference Bicep inline

While having a Bicep file on disk is the most common scenario, you can also add Bicep templates inline. Inline templates can be useful when you want to define a template in code or when you want to generate the template dynamically. To add an inline Bicep template, call the [AddBicepTemplateString](#) method with the Bicep template as a `string`. Consider the following example:

```
C#  
  
builder.AddBicepTemplateString(  
    name: "ai",  
    bicepContent: """  
        @description('That name is the name of our application.')  
        param cognitiveServiceName string =  
            'CognitiveService-${uniqueString(resourceGroup().id)}'  
  
        @description('Location for all resources.')  
        param location string = resourceGroup().location  
  
        @allowed([  
            'S0'  
        ])  
        param sku string = 'S0'  
  
        resource cognitiveService 'Microsoft.CognitiveServices/accounts@2021-10-01'  
        = {  
            name: cognitiveServiceName  
            location: location  
            sku: {  
                name: sku  
            }  
            kind: 'CognitiveServices'  
            properties: {  
                apiProperties: {  
                    statisticsEnabled: false  
                }  
            }  
        }  
    """;  
);
```

In this example, the Bicep template is defined as an inline `string` and added to the application's resources collection with the name `"ai"`. This example provisions an Azure AI resource.

Pass parameters to Bicep templates

Bicep supports accepting parameters, which can be used to customize the behavior of the template. To pass parameters to a Bicep template from .NET Aspire, chain calls to the [WithParameter](#) method as shown in the following example:

C#

```
var region = builder.AddParameter("region");

builder.AddBicepTemplate("storage", "../infra/storage.bicep")
    .WithParameter("region", region)
    .WithParameter("storageName", "app-storage")
    .WithParameter("tags", ["latest", "dev"]);
```

The preceding code:

- Adds a parameter named `"region"` to the `builder` instance.
- Adds a reference to a Bicep file located at `../infra/storage.bicep`.
- Passes the `"region"` parameter to the Bicep template, which is resolved using the standard parameter resolution.
- Passes the `"storageName"` parameter to the Bicep template with a *hardcoded* value.
- Passes the `"tags"` parameter to the Bicep template with an array of strings.

For more information, see [External parameters](#).

Well-known parameters

.NET Aspire provides a set of well-known parameters that can be passed to Bicep templates. These parameters are used to provide information about the application and the environment to the Bicep templates. The following well-known parameters are available:

 [Expand table](#)

Field	Description	Value
<code>AzureBicepResource.KnownParameters.KeyVaultName</code>	The name of the key vault resource used to store secret outputs.	<code>"keyVaultName"</code>
<code>AzureBicepResource.KnownParameters.Location</code>	The location of the resource. This is required for all resources.	<code>"location"</code>
<code>AzureBicepResource.KnownParameters.LogAnalyticsWorkspaceId</code>	The resource ID of the log analytics workspace.	<code>"logAnalyticsWorkspaceId"</code>
<code>AzureBicepResource.KnownParameters.PrincipalId</code>	The principal ID of the current user or managed identity.	<code>"principalId"</code>

Field	Description	Value
AzureBicepResource.KnownParameters.PrincipalName	The principal name of the current user or managed identity.	"principalName"
AzureBicepResource.KnownParameters.PrincipalType	The principal type of the current user or managed identity. Either User or ServicePrincipal.	"principalType"

To use a well-known parameter, pass the parameter name to the [WithParameter](#) method, such as `WithParameter(AzureBicepResource.KnownParameters.KeyVaultName)`. You don't pass values for well-known parameters, as they're resolved automatically by .NET Aspire.

Consider an example where you want to setup an Azure Event Grid webhook. You might define the Bicep template as follows:

Bicep

```
param topicName string
param webHookEndpoint string
param principalId string
param principalType string
param location string = resourceGroup().location

// The topic name must be unique because it's represented by a DNS entry.
// must be between 3-50 characters and contain only values a-z, A-Z, 0-9, and "-".

resource topic 'Microsoft.EventGrid/topics@2023-12-15-preview' = {
    name: toLower(take('${topicName}${uniqueString(resourceGroup().id)}', 50))
    location: location

    resource eventSubscription 'eventSubscriptions' = {
        name: 'customSub'
        properties: {
            destination: {
                endpointType: 'WebHook'
                properties: {
                    endpointUrl: webHookEndpoint
                }
            }
        }
    }
}
```

```

resource EventGridRoleAssignment 'Microsoft.Authorization/roleAssignments@2022-04-01' = {
    name: guid(topic.id, principalId,
    subscriptionResourceId('Microsoft.Authorization/roleDefinitions', 'd5a91429-5739-47e2-a06b-3470a27159e7'))
    scope: topic
    properties: {
        principalId: principalId
        principalType: principalType
        roleDefinitionId:
    subscriptionResourceId('Microsoft.Authorization/roleDefinitions', 'd5a91429-5739-47e2-a06b-3470a27159e7')
    }
}

output endpoint string = topic.properties.endpoint

```

This Bicep template defines several parameters, including the `topicName`, `webHookEndpoint`, `principalId`, `principalType`, and the optional `location`. To pass these parameters to the Bicep template, you can use the following code snippet:

C#

```

var webHookApi = builder.AddProject<Projects.WebHook_Api>("webhook-api");

var webHookEndpointExpression = ReferenceExpression.Create(
    $"{webHookApi.GetEndpoint("https")}/hook");

builder.AddBicepTemplate("event-grid-webhook", "../infra/event-grid-webhook.bicep")
    .WithParameter("topicName", "events")
    .WithParameter(AzureBicepResource.KnownParameters.PrincipalId)
    .WithParameter(AzureBicepResource.KnownParameters.PrincipalType)
    .WithParameter("webHookEndpoint", () => webHookEndpointExpression);

```

- The `webHookApi` project is added as a reference to the `builder`.
- The `topicName` parameter is passed a hardcoded name value.
- The `webHookEndpoint` parameter is passed as an expression that resolves to the URL from the `api` project references' "https" endpoint with the `/hook` route.
- The `principalId` and `principalType` parameters are passed as well-known parameters.

The well-known parameters are convention-based and shouldn't be accompanied with a corresponding value when passed using the `WithParameter` API. Well-known parameters simplify some common functionality, such as *role assignments*, when added to the Bicep templates, as shown in the preceding example. Role assignments are required for the Event Grid webhook to send events to the specified endpoint. For more information, see [EventGrid Data Sender role assignment](#).

Get outputs from Bicep references

In addition to passing parameters to Bicep templates, you can also get outputs from the Bicep templates. Consider the following Bicep template, as it defines an `output` named `endpoint`:

```
Bicep

param storageName string
param location string = resourceGroup().location

resource myStorageAccount 'Microsoft.Storage/storageAccounts@2019-06-01' = {
    name: storageName
    location: location
    kind: 'StorageV2'
    sku: {
        name: 'Standard_LRS'
        tier: 'Standard'
    }
    properties: {
        accessTier: 'Hot'
    }
}

output endpoint string = myStorageAccount.properties.primaryEndpoints.blob
```

The Bicep defines an output named `endpoint`. To get the output from the Bicep template, call the `GetOutput` method on an `IResourceBuilder<AzureBicepResource>` instance as demonstrated in following C# code snippet:

```
C#

var storage = builder.AddBicepTemplate(
    name: "storage",
    bicepFile: "../infra/storage.bicep"
);

var endpoint = storage.GetOutput("endpoint");
```

In this example, the output from the Bicep template is retrieved and stored in an `endpoint` variable. Typically, you would pass this output as an environment variable to another resource that relies on it. For instance, if you had an ASP.NET Core Minimal API project that depended on this endpoint, you could pass the output as an environment variable to the project using the following code snippet:

```
C#

var storage = builder.AddBicepTemplate(
    name: "storage",
    bicepFile: "../infra/storage.bicep"
);

var endpoint = storage.GetOutput("endpoint");

var apiService = builder.AddProject<Projects.AspireSample_ApiService>(
    name: "apiservice"
```

```
)  
.WithEnvironment("STORAGE_ENDPOINT", endpoint);
```

For more information, see [Bicep outputs](#).

Get secret outputs from Bicep references

It's important to [avoid outputs for secrets](#) when working with Bicep. If an output is considered a *secret*, meaning it shouldn't be exposed in logs or other places, you can treat it as such. This can be achieved by storing the secret in Azure Key Vault and referencing it in the Bicep template. .NET Aspire's Azure integration provides a pattern for securely storing outputs from the Bicep template by allows resources to use the `keyVaultName` parameter to store secrets in Azure Key Vault.

Consider the following Bicep template as an example the helps to demonstrate this concept of securing secret outputs:

Bicep

```
param databaseAccountName string  
param keyVaultName string  
  
param databases array = []  
  
@description('Tags that will be applied to all resources')  
param tags object = {}  
  
param location string = resourceGroup().location  
  
var resourceToken = uniqueString(resourceGroup().id)  
  
resource cosmosDb 'Microsoft.DocumentDB/databaseAccounts@2023-04-15' = {  
    name: replace('${databaseAccountName}-${resourceToken}', '-', '')  
    location: location  
    kind: 'GlobalDocumentDB'  
    tags: tags  
    properties: {  
        consistencyPolicy: { defaultConsistencyLevel: 'Session' }  
        locations: [  
            {  
                locationName: location  
                failoverPriority: 0  
            }  
        ]  
        databaseAccountOfferType: 'Standard'  
    }  
  
    resource db 'sqlDatabases@2023-04-15' = [for name in databases: {  
        name: '${name}'  
        location: location  
        tags: tags  
        properties: {  
            resource: {
```

```

        id: '${name}'
    }
}
}

var primaryMasterKey = cosmosDb.listKeys(cosmosDb.apiVersion).primaryMasterKey

resource vault 'Microsoft.KeyVault/vaults@2023-07-01' existing = {
    name: keyVaultName

    resource secret 'secrets@2023-07-01' = {
        name: 'connectionString'
        properties: {
            value:
'AccountEndpoint=${cosmosDb.properties.documentEndpoint};AccountKey=${primaryMasterK
ey}'
        }
    }
}

```

The preceding Bicep template expects a `keyVaultName` parameter, among several other parameters. It then defines an Azure Cosmos DB resource and stashes a secret into Azure Key Vault, named `connectionString` which represents the fully qualified connection string to the Cosmos DB instance. To access this secret connection string value, you can use the following code snippet:

```

C#

var cosmos = builder.AddBicepTemplate("cosmos", "../infra/cosmosdb.bicep")
    .WithParameter("databaseAccountName", "fallout-db")
    .WithParameter(AzureBicepResource.KnownParameters.KeyVaultName)
    .WithParameter("databases", ["vault-33", "vault-111"]);

var connectionString =
    cosmos.GetSecretOutput("connectionString");

builder.AddProject<Projects.WebHook_Api>("api")
    .WithEnvironment(
        "ConnectionStrings__cosmos",
        connectionString);

```

In the preceding code snippet, the `cosmos` Bicep template is added as a reference to the `builder`. The `connectionString` secret output is retrieved from the Bicep template and stored in a variable. The secret output is then passed as an environment variable (`ConnectionStrings__cosmos`) to the `api` project. This environment variable is used to connect to the Cosmos DB instance.

When this resource is deployed, the underlying deployment mechanism will automatically [Reference secrets from Azure Key Vault](#). To guarantee secret isolation, .NET Aspire creates a Key Vault per source.

Note

In *local provisioning* mode, the secret is extracted from Key Vault and set it in an environment variable. For more information, see [Local Azure provisioning](#).

See also

For continued learning, see the following resources as they relate to .NET Aspire and Azure deployment:

- [Deploy a .NET Aspire project to Azure Container Apps](#)
- [Deploy a .NET Aspire project to Azure Container Apps using the Azure Developer CLI \(in-depth guide\)](#)
- [.NET Aspire manifest format for deployment tool builders](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Allow unsecure transport in .NET Aspire

Article • 06/15/2024

Starting with .NET Aspire preview 5, the app host will crash if an `applicationUrl` is configured with an unsecure transport (non-TLS `http`) protocol. This is a security feature to prevent accidental exposure of sensitive data. However, there are scenarios where you might need to allow unsecure transport. This article explains how to allow unsecure transport in .NET Aspire projects.

Symptoms

When you run a .NET Aspire project with an `applicationUrl` configured with an unsecure transport protocol, you might see the following error message:

Output

```
The 'applicationUrl' setting must be an https address unless the  
'ASPIRE_ALLOW_UNSECURED_TRANSPORT' environment variable is set to true.
```

This configuration is commonly set in the launch profile.

How to allow unsecure transport

To allow an unsecure transport in .NET Aspire, set the `ASPIRE_ALLOW_UNSECURED_TRANSPORT` environment variable to `true`. This environment variable is used to control the behavior of the app host when an `applicationUrl` is configured with an insecure transport protocol:

Windows

PowerShell

```
$env:ASPIRE_ALLOW_UNSECURED_TRANSPORT = "true"
```

Alternatively, you can control this via the launch profile as it exposes the ability to configure environment variables per profile. To do this, consider the following example settings in the `launchSettings.json` file:

JSON

```
{  
    "$schema": "http://json.schemastore.org/launchsettings.json",  
    "profiles": {  
        "https": {  
            "commandName": "Project",  
            "dotnetRunMessages": true,  
            "launchBrowser": true,  
            "applicationUrl": "https://localhost:15015;http://localhost:15016",  
            "environmentVariables": {  
                "ASPNETCORE_ENVIRONMENT": "Development",  
                "DOTNET_ENVIRONMENT": "Development",  
                "DOTNET_DASHBOARD_OTLP_ENDPOINT_URL": "https://localhost:16099",  
                "DOTNET_RESOURCE_SERVICE_ENDPOINT_URL": "https://localhost:17037"  
            }  
        },  
        "http": {  
            "commandName": "Project",  
            "dotnetRunMessages": true,  
            "launchBrowser": true,  
            "applicationUrl": "http://localhost:15016",  
            "environmentVariables": {  
                "ASPNETCORE_ENVIRONMENT": "Development",  
                "DOTNET_ENVIRONMENT": "Development",  
                "DOTNET_DASHBOARD_OTLP_ENDPOINT_URL": "http://localhost:16099",  
                "DOTNET_RESOURCE_SERVICE_ENDPOINT_URL": "http://localhost:17038",  
                "ASPIRE_ALLOW_UNSECURED_TRANSPORT": "true"  
            }  
        }  
    }  
}
```

The preceding example shows two profiles, `https` and `http`. The `https` profile is configured with a secure transport protocol, while the `http` profile is configured with an insecure transport protocol. The `ASPIRE_ALLOW_UNSECURED_TRANSPORT` environment variable is set to `true` in the `http` profile to allow unsecure transport.

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Troubleshoot untrusted localhost certificate in .NET Aspire

Article • 06/13/2024

This article provides guidance on how to troubleshoot issues that you might encounter when working with untrusted localhost certificates in .NET Aspire.

Symptoms

Several .NET Aspire templates include ASP.NET Core projects that are configured to use HTTPS by default. If this is the first time you're running the project, and you're using Visual Studio, you're prompted to install a localhost certificate.

- There are situations in which you trust/install the development certificate, but you don't close all your browser windows. In these cases, your browser might indicate that the certificate isn't trusted.
- There are also situations where you don't trust the certificate at all. In these cases, your browser might indicate that the certificate isn't trusted.

Additionally, there are warning messages from Kestrel written to the console that indicate that the certificate is not trusted.

Possible solutions

Close all browser windows and try again. If you're still experiencing the issue, then attempt to resolve this by trusting the self-signed development certificate with the .NET CLI. To trust the certificate, run the following commands. First, remove the existing certificates.

Note

This will remove all existing development certificates on the local machine.

.NET CLI

```
dotnet dev-certs https --clean
```

To trust the certificate:

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

For more troubleshooting, see [Troubleshoot certificate problems such as certificate not trusted](#).

See also

- [Trust the ASP.NET Core HTTPS development certificate on Windows and macOS](#)
- [Trust HTTPS certificate on Linux](#)
- [.NET CLI: dotnet dev-certs](#)
- [Trust localhost certificate on Linux ↗](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Troubleshoot installing the .NET Aspire workload

Article • 05/21/2024

This article provides guidance on how to troubleshoot issues that you might encounter when installing the .NET Aspire workload from the .NET CLI.

Symptoms

When you install the .NET Aspire workload, you might encounter an installation error. The error message might indicate that the installation failed, or that the workload couldn't be installed. The error message might also indicate that a package source is unavailable, or that a package source isn't found often similar to:

Output

```
Workload update failed: One or more errors occurred: (Version X.Y.00Z of package A.B.C is not found in NuGet feeds.)
```

One common issue is that your SDK is aware of some workload manifest or workload pack versions that are not present in any of the feeds configured when you are trying to run the `dotnet workload` commands. This can happen if the SDK, during its daily check for updates, finds a new version of a workload manifest in a feed that isn't used when running `dotnet workload` commands. This discrepancy can cause errors during installation.

A less common issue, even when using the correct feeds, is that a workload manifest may have a dependency on a workload pack that is not published on the same feed. This can also lead to errors during installation as the required pack cannot be found.

Possible solution

Ensure that any recursive *Nuget.config* files are configured to specify the correct package sources and NuGet feeds. For example, if you have a *Nuget.config* file in your user profile directory, ensure that it doesn't specify a package source that is no longer available.

If you encounter errors related to the SDK being aware of workload manifest or workload pack versions not present in your configured feeds, you may need to adjust

your feeds or find the feed where the new version of the manifest or required pack is located.

In the case where a workload manifest has a dependency on a workload pack not published on the same feed, you will need to find and add the feed where that pack is located to your NuGet configuration.

Important

Some development environments may depend on private feeds that provide newer versions of the workload manifest or workload pack. In these situations, you may want to disable the daily SDK check for updates to avoid encountering errors during installation.

To disable the daily SDK check for updates, set the

`DOTNET_CLI_WORKLOAD_UPDATE_NOTIFY_DISABLE` environment variable to `true`.

See also

- [.NET SDK: Diagnosing issues with .NET SDK Workloads ↗](#)
- [.NET CLI: dotnet workload install](#)
- [NuGet: nuget.config reference](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

The specified name is already in use

Article • 06/15/2024

When deploying to Azure initial deployments may fail with an error similar to the following:

"The specified name is already in use"

This article describes several techniques to avoid this problem.

Symptoms

When deploying a .NET Aspire project to Azure, the resources in the [app model](#) are transformed into Azure resources. Some Azure resources have globally scoped names, such as Azure App Configuration, where all instances are in the `[name].azconfig.io` global namespace.

The value of `[name]` is derived from the .NET Aspire resource name, along with random characters based on the resource group name. However, the generated string may exceed the allowable length for the resource name in App Configuration. As a result, some characters are truncated to ensure compliance.

When a conflict occurs in the global namespace, the resource fails to deploy because the combination of `[name]+[truncated hash]` isn't unique enough.

Possible solutions

One workaround is to avoid using common names like `appconfig` or `storage` for resources. Instead, choose a more meaningful and specific name. This reduces the potential for conflict, but does not completely eliminate it. In such cases, you can use callback methods to set a specific name and avoid using the computed string altogether.

Consider the following example:

C#

```
var appConfig = builder.AddAzureAppConfiguration(
    "appConfig",
    (resource, construct, store) =>
{
```

```
    store.AssignProperty(p => p.Name, "'noncalculatedname'");  
});
```

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Container runtime appears to be unhealthy

Article • 07/08/2024

.NET Aspire requires Docker (or Podman) to be running and healthy. This topic describes a possible symptom you may see if Docker isn't in a healthy state.

Symptoms

When starting the AppHost the dashboard doesn't show up and an exception stack trace similar to this example is displayed in the console:

```
Output

info: Aspire.Hosting.DistributedApplication[0]
      Aspire version: 8.1.0-dev
info: Aspire.Hosting.DistributedApplication[0]
      Distributed application starting.
info: Aspire.Hosting.DistributedApplication[0]
      Application host directory is:
D:\aspire\playground\PostgresEndToEnd\PostgresEndToEnd.AppHost
fail: Microsoft.Extensions.Hosting.Internal.Host[11]
      Hosting failed to start
      Aspire.Hosting.DistributedApplicationException: Container runtime
'docker' was found but appears to be unhealthy. The error from the container
runtime check was error during connect: this error may indicate that the
docker daemon is not running: Get
"http://%2F%2Fpipe%2Fdocker_engine/v1.45/containers/json?limit=1": open
//./pipe/docker_engine: The system cannot find the file specified..
```

Possible solutions

Confirm that Docker is installed and running:

- On Windows, check that in the system tray the Docker icon is present and marked as "Running".
- On Linux, check that `docker ps -a` returns success.

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET Aspire is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Connection string is missing

Article • 08/29/2024

In .NET Aspire, code identifies resources with an arbitrary string, such as "database". Code that is consuming the resource elsewhere must use the same string or it will fail to correctly configure their relationships.

Symptoms

When your app accesses a service that needs one of the integrations in your app, it may fail with an exception similar to the following:

```
| "InvalidOperationException: ConnectionString is missing."
```

Possible solutions

Verify that the name of the resource, for instance a database resource, is the same in the AppHost and the Service that fails.

For example, if the AppHost defines a PostgreSQL resource with the name `db1` like this:

```
C#  
  
var db1 = builder.AddPostgres("pg1").AddDatabase("db1");
```

Then the service needs to resolve the resource with the same name `db1`.

```
C#  
  
var builder = WebApplication.CreateBuilder(args);  
  
builder.AddNpgsqlDbContext<MyDb1Context>("db1");
```

Any other value than the one provided in the AppHost will result in the exception message described above.

.NET Aspire diagnostics overview

Article • 04/23/2024

Several APIs of .NET Aspire are decorated with the [ExperimentalAttribute](#). This attribute indicates that the API is experimental and may be removed or changed in future versions of .NET Aspire. The attribute is used to identify APIs that aren't yet stable and may not be suitable for production use.

AZPROVISION001

.NET Aspire provides various overloads for Azure Provisioning resource types (from the `Azure.Provisioning` package). The overloads are used to create resources with different configurations. The overloads are experimental and may be removed or changed in future versions of .NET Aspire.

To suppress this diagnostic with the `SuppressMessageAttribute`, add the following code to your project:

```
C#  
  
using System.Diagnostics.CodeAnalysis;  
  
[assembly: SuppressMessage("AZPROVISION001", "Justification")]
```

Alternatively, you can suppress this diagnostic with preprocessor directive by adding the following code to your project:

```
C#  
  
#pragma warning disable AZPROVISION001  
    // API that is causing the warning.  
#pragma warning restore AZPROVISION001
```

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For



.NET Aspire feedback

.NET Aspire is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

Frequently asked questions about .NET Aspire

FAQ

This article lists frequently asked questions about .NET Aspire. For a more comprehensive overview, see [.NET Aspire overview](#).

Why use .NET Aspire for orchestration when I can use Docker Compose?

Docker Compose is excellent but is unproductive when all you want to do is run several projects or executables. Docker Compose requires developers to build container images and to run apps inside of containers. That's a barrier when you just want to run your front end, back end, workers, and a database. With .NET Aspire, you don't need to learn anything beyond what you already know.

Configuration through declarative code is better than through YAML. Docker Compose gets complex once you attempt to do any form of abstraction or composition (for example, see the old [eshopOnContainers app](#)). In addition, there are environment variable replacements (and includes) and no types or IntelliSense, and it's hard to reason about what exactly is running. Debugging is also difficult. .NET Aspire produces a better experience that's easy to get started and scales up to an orchestrator like Compose using a real programming language.

How can I add other projects to my .NET Aspire solution?

You can manually add projects to your .NET Aspire solution by using the

```
builder.AddProject("<name>", "<path/to/project.csproj>") API.
```

How can I deploy my .NET Aspire solution when tooling doesn't exist for my target cloud-provider?

.NET Aspire doesn't constrain deployment of any existing project or solution. .NET Aspire exposes a [deployment manifest](#) that's used by tool authors to produce artifacts for deployment to any cloud provider. However, unfortunately, not all cloud providers offer tooling for deployments based on this manifest. The manifest is a simple JSON file that describes the resources of your app and the dependencies between them. The manifest is used by the Azure Developer CLI to deploy to Azure. Likewise, Aspir8 uses the manifest to deploy to Kubernetes. You can use the manifest to deploy to any cloud provider that supports the resources you're using.

Can we build .NET Aspire apps without using any Azure-proprietary dependencies? What if we build an Aspire web app and choose not to deploy it on Azure or choose later to move it from Azure?

Yes, you can build .NET Aspire apps without using any Azure-proprietary dependencies. While .NET Aspire does offer a first-party solution to deploying to Azure, it's not a requirement. .NET Aspire is a cloud-native stack that can be used to build applications that run anywhere. All Azure-specific offerings are explicitly called out as such.

Why would I use .NET Aspire service discovery when Docker Compose has it built in and works with Kubernetes?

.NET Aspire service discovery APIs are an abstraction that works with various providers (like Kubernetes and Consul). One of the big advantages is that it works locally and is backed by .NET's `IConfiguration` abstraction. This means you can implement service discovery across your compute fabric in a way that doesn't result in code changes. If you have multiple Kubernetes clusters or services on Azure App Service or Azure Functions, you don't have to fundamentally change your application code to make it work locally, either in a single cluster or across multiple clusters. That's the benefit of the abstraction.

OpenTelemetry is something that can already be used in .NET. Why would I use .NET Aspire?

.NET Aspire takes a big bet on .NET's integration with OpenTelemetry. The .NET Aspire dashboard is a standard OTLP server that visualizes various telemetry data. Leaning on these open standards makes it easy to build these things without breaking compatibility with the broader ecosystem.

Observability tools such as Grafana, Jaeger, and Prometheus work with .NET. Why bother with .NET Aspire?

.NET Aspire isn't a replacement for these tools, but rather a complementary technology. .NET Aspire is a set of libraries and tools that make it easy to build applications that are observable. For more information, see the [Metrics example in the .NET Aspire sample repository](#) that shows Grafana and Prometheus.

Why is there a need for yet another framework to do what's already being done very well by everyone else?

.NET Aspire isn't a framework, it's an [opinionated stack](#). Perhaps the most controversial parts of it are the `DistributedApplication` APIs that you can use to build up the orchestration model in any .NET-based language. While everything is possible today, it's not easy. Using the Unix philosophy, the entire cloud-native ecosystem is built around tying various pieces of CNCF software together to build a stack. .NET Aspire tries to do the same thing using learnings from the cloud-native space and picks some opinions (in ways that use the same building blocks). One novel thing about how .NET Aspire builds various pieces of the stack is that it doesn't restrict the access or compatibility of other applications, frameworks, or services. As people play with it more, they realize how composable and extensible it is.

What's the difference between .NET Aspire and Microsoft Orleans?

Microsoft Orleans and .NET Aspire are complementary technologies.

[Orleans](#) is a distributed actor-based framework. .NET Aspire is a cloud-ready stack for building observable, production-ready, distributed applications. It includes local orchestration capabilities to simplify the developer inner loop and reusable opinionated components for integrating with commonly used application dependencies. An Orleans-based solution will still have external dependencies such as data stores and caches for which .NET Aspire can be used for orchestration purposes.

For more information, see [Use Orleans with .NET Aspire](#) and the corresponding [Orleans voting app sample](#).

What's the difference between .NET Aspire and Dapr?

Dapr and .NET Aspire are complementary technologies.

Where Dapr abstracts some of the underlying cloud platform, .NET Aspire provides opinionated configuration around the underlying cloud technologies without abstracting them. A .NET-based application that uses Dapr can use .NET Aspire to orchestrate the local developer inner loop and streamline deployment. .NET Aspire includes extensions that support the launching of Dapr side-car processes during the inner loop.

For more information, see [Use Dapr with .NET Aspire](#) and the corresponding [Dapr sample app ↗](#) in the .NET Aspire sample repository.

What's the difference between .NET Aspire and Project Tye?

Project Tye was an experiment which explored the launching and orchestration of micro-services and support deployment into orchestrators such as Kubernetes. .NET Aspire is a superset of Tye which includes the orchestration and deployment capabilities along with opinionated components for integrating common cloud-native dependencies. .NET Aspire can be considered the evolution of the Project Tye experiment.

What's the relationship between .NET Aspire and the Azure SDK for .NET?

.NET Aspire provides components that rely on the [Azure SDK for .NET](#), to expose common functionality for storage ([Azure Blob Storage](#), [Azure Storage Queues](#), and [Azure Table Storage](#)), databases ([Azure Cosmos DB](#) and [Azure Cosmos DB with Entity Framework Core](#)), [messaging](#), and [security](#).

What's the relationship between .NET Aspire and Kubernetes?

.NET Aspire makes it easy to develop distributed applications that can be orchestrated on your local development environment as executables and containers. Kubernetes is a technology that orchestrates and manages containers across multiple machines. .NET Aspire projects can produce a [manifest](#) that tool authors can use to produce artifacts for deployment to Kubernetes. In essence, Kubernetes is a deployment target for .NET Aspire projects.

Are worker services supported in .NET Aspire?

Yes, worker services are fully supported and there are docs and samples available to help you get started. Worker services are a great way to run background tasks, scheduled tasks, or long-running tasks in .NET Aspire. For more information, see [Database migrations with Entity Framework Core sample app](#).

Are Azure Functions supported in .NET Aspire?

We have no specific support for Azure Functions thus far in .NET Aspire, however it's a target execution environment for deployment that we are planning to support in future.

I want to run my web apps locally on IIS or IIS Express, does .NET Aspire support

this?

No. .NET Aspire doesn't support running web apps on IIS or IIS Express.

I want to deploy my apps to IIS, does .NET Aspire support this?

No. .NET Aspire doesn't support deploying apps to IIS. However, it doesn't prevent you from deploying your apps to IIS in the same way that you always have.

When deploying my existing apps, that are now referenced within a .NET Aspire solution, to IIS some integrations and Service Discovery isn't working as expected. What can I do?

.NET Aspire integrations require specific configuration that must be provided manually. The same is true for [Service Discovery](#), ideally, you should deploy to something other than IIS.

Next steps

To learn more about networking and functions:

- [.NET Aspire overview](#)
- [Build your first .NET Aspire project](#)
- [.NET Aspire components](#)