*Configure & Use Aspire for Azure Emulators*

|  |  |
| --- | --- |
| Bert O’Neill | |
| Department: | Software Development |
| Document Title: | Use Aspire with Azure Emulators |
| Author: | Bert O’Neill |
| Date: | 25-Aug-2025 |
| Version | 1.0 |
| Email | Bert.ONeill@Yahoo.com |
| Document Goals | Provide an overview for developers, on how to quickly configure their .Net 9 Aspire (Hosting) project to consume (Docker) Azure Emulators. |

# Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Description | Author |
| 25-Aug-2025 | 1.0 | Initial draft | Bert O’Neill |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

[Revision History 1](#_Toc207024043)

[Introduction 3](#_Toc207024044)

[Purpose 3](#_Toc207024045)

[Scope 3](#_Toc207024046)

[Prerequisites 3](#_Toc207024047)

[Emulator Installation (Service Bus \ Storage \ SignalR) 4](#_Toc207024048)

[Microsoft Azure SignalR Emulator 4](#_Toc207024049)

[Start SignalR 4](#_Toc207024050)

[Azure Storage Emulator (Blob\Queue\Table) 4](#_Toc207024051)

[Service Bus Emulator 5](#_Toc207024052)

[Serverless Architecture – Why Use Emulators 7](#_Toc207024053)

[Project Structure 8](#_Toc207024054)

[Setting Up Aspire - NuGet Packages 9](#_Toc207024055)

[Create Azure Service Bus Emulator 9](#_Toc207024056)

[Create Azure Function Emulator 9](#_Toc207024057)

[Create Azure SignalR Emulator 9](#_Toc207024058)

[Create Azure Storage Emulator 9](#_Toc207024059)

[Miscellaneous 9](#_Toc207024060)

[Acronym 9](#_Toc207024061)

# Introduction

This document provides an overview on how to configure your Visual Studio (Aspire) environment to incorporate Azure Emulators, to design\test your serverless applications locally.

## Purpose

Your project maybe in a position where you now need to demo to a client the latest Azure feature, your team has developed or you have implemented a code fix, but you’re not able to spin up Azure resources, due to security or financial restrictions. To get around this, you can avail of the latest Serverless Emulators.

## Scope

The scope of this document is to convey the configuration steps needed to start using Azure Emulators with .Net Aspire.

# Prerequisites

* [Docker](https://docs.docker.com/desktop/) (desktop) installed.
* [Visual Studio 22 Community](https://visualstudio.microsoft.com/vs/community/) installed (updated to version 17.14.13 August 2025).
* [Clone GitHub Code Repo](https://github.com/Bert0Neill/AspireAzuriteDemo).
* [NPM](https://nodejs.org/en) installed.

# Emulator Installation (Service Bus \ Storage \ SignalR)

Within your development environment or on a server that your team has access to, install the components below:

## Microsoft Azure SignalR Emulator

From a command prompt, run the following .Net CLI command

dotnet tool **install** -g Microsoft.Azure.SignalR.Emulator

Or *update* if already installed:

dotnet tool **update** -g Microsoft.Azure.SignalR.Emulator

Or to *uninstall*:

dotnet tool **uninstall** -g Microsoft.Azure.SignalR.Emulator

A screenshot of a computer

AI-generated content may be incorrect.

For simplicities sake, I have created a folder on my C:\ drive called *Azurite* (this will host the storage files etc.).

### Start SignalR

Run the following command to start the SignalR emulator locally:

asrs-emulator start

Below, you will see the connection string to use within your code:

A screenshot of a computer

AI-generated content may be incorrect.

## Azure Storage Emulator (Blob\Queue\Table)

Run the following statement ***once*** from your command prompt and then *Ctrl C* to cancel the job (**NB** I am suppling the path to use, my folder above) – this is to setup the folder structure (you will then reference this folder in your code, each time):

azurite --location **C:\Azurite**\Azurite-Data --debug C:\Azurite\Azurite-Debug.log

You can see the URLs that will be allocated to each storage type, these are the URLs and port numbers that you will later use within your codebase:

A screenshot of a computer screen

AI-generated content may be incorrect.

This will create the appropriate files within the storage location, supplied in the prompt:

A screenshot of a computer

AI-generated content may be incorrect.

In our code, if you are using Azure Storage Emulator, you will reference the above standard setup:

A computer screen shot of a program

AI-generated content may be incorrect.

If you find that the executable (azurite.exe) doesn’t exist within your environment, ensure that you have Visual Studio 22 updated to the latest version – (time of blog 17.14.13):

A screenshot of a computer

AI-generated content may be incorrect.

## Service Bus Emulator

Ensure that you have Docker running:

A screen shot of a phone

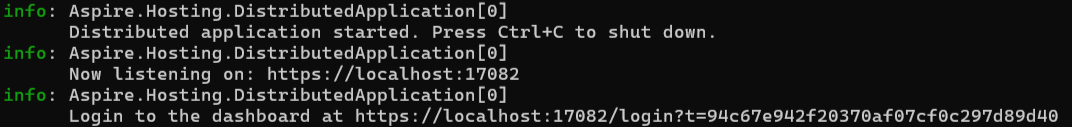
AI-generated content may be incorrect.

You won’t have to startup any tools or run any commands to have the Service Bus running, this will be taken care off by the .Net Aspire container project.

Example of an Aspire screen below, showing the Service Bus and SQL Edge images running in a container.

A screenshot of a computer

AI-generated content may be incorrect.



Docker will automatically host the same configuration (no need for any setup scripts) – configured within the code:

A screenshot of a computer

AI-generated content may be incorrect.

*Alternatively*, you can setup the Service Bus\SQL Edge images manually and connect to those instances instead of a runtime approach *(I’ve included the steps for completeness below, but within this blog, I will let the .Net code define the images for Docker)*:

Copy the following three files to the folder you created earlier (for e.g. C:\Azurite):



A screenshot of a computer

AI-generated content may be incorrect.

Edit the **.env** file, so that *CONFIG\_PATH* is pointing to the config.json file location:

A screenshot of a computer code

AI-generated content may be incorrect.

Run the following docker composer command to load Docker with the appropriate *Service Bus* emulator and the *SQL Edge* images:

docker compose -f "C:\Azurite\Docker\docker-compose.yaml" up -d

The first time you run the docker compose command it might take a couple of minutes to download the images, but subsequent calls will start immediately.

A screenshot of a computer

AI-generated content may be incorrect.

Then, within Docker you can see the Container with the two images loaded:

A screenshot of a computer

AI-generated content may be incorrect.

NB: That the two image names come from your Docker Composer file:

A screenshot of a computer

AI-generated content may be incorrect.

**NB:** That in the Service Bus logs, you can see the connection string:

A screenshot of a computer

AI-generated content may be incorrect.

You have now completed the environment setup for Azure emulators!

## CosmosDB Emulator

I’m not using CosmosDB in this particular example, but I’ll include how to install the emulator, and the code needed. Use the following [Microsoft page](https://learn.microsoft.com/en-us/azure/cosmos-db/how-to-develop-emulator?tabs=windows%2Ccsharp&pivots=api-mongodb) as a guide:

1. Download the [Azure Cosmos DB emulator](https://aka.ms/cosmosdb-emulator).
2. Run the installer on your local machine with **administrative privileges**.
3. The emulator automatically installs the appropriate developer certificates and configures firewall rules on your local machine.
4. Start the emulator by selecting the application in the Windows **Start menu**.
5. Alternatively, you can start the emulator's executable (Microsoft.Azure.Cosmos.Emulator.exe) at the *%ProgramFiles%\Azure Cosmos DB Emulator path*.

**NB**: You can add the port number to use, as a switch parameter

Microsoft.Azure.Cosmos.Emulator.exe **/Port=65000**

**Add the required NuGet package**

dotnet add package Microsoft.Azure.Cosmos

**2️⃣ Create a simple model**

public class MyItem

{

public string Id { get; set; }

public string Name { get; set; }

}

**3️⃣ Create a Cosmos DB service**

using Microsoft.Azure.Cosmos;

public class CosmosService

{

private readonly CosmosClient \_client;

private readonly Container \_container;

public CosmosService()

{

string connectionString = "AccountEndpoint=https://localhost:8081/;AccountKey=C2y6yDjf5/R+ob0N8A7Cgv30VR;";

\_client = new CosmosClient(connectionString);

// Create database and container if they don't exist

var database = \_client.CreateDatabaseIfNotExistsAsync("TestDatabase").Result;

\_container = database.Database.CreateContainerIfNotExistsAsync(

new ContainerProperties("TestContainer", "/id")).Result.Container;

}

public async Task AddItemAsync(MyItem item)

{

await \_container.CreateItemAsync(item, new PartitionKey(item.Id));

}

public async Task<List<MyItem>> GetItemsAsync()

{

var query = \_container.GetItemQueryIterator<MyItem>("SELECT \* FROM c");

var results = new List<MyItem>();

while (query.HasMoreResults)

{

var response = await query.ReadNextAsync();

results.AddRange(response.ToList());

}

return results;

}

}

**4️⃣ Register the service in Aspire**

builder.Services.AddSingleton<CosmosService>();

**5️⃣ Create a minimal API**

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Http;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<CosmosService>();

var app = builder.Build();

app.MapPost("/items", async (MyItem item, CosmosService cosmosService) =>

{

await cosmosService.AddItemAsync(item);

return Results.Ok(item);

});

app.MapGet("/items", async (CosmosService cosmosService) =>

{

var items = await cosmosService.GetItemsAsync();

return Results.Ok(items);

});

app.Run();

**6️⃣ Testing**

* Start the **Cosmos DB Emulator** first.
* Run your Aspire-hosted API.
* Use **Postman** or **curl** to test:

**Add an item:**

POST <https://localhost:5001/items>

Content-Type: application/json

{

"id": "1",

"name": "Test Item"

}

**Get all items:**

GET <https://localhost:5001/items>

You should see the items you added in the emulator.

# Serverless Architecture – Why Use Emulators

For years Amazon has had [LocalStack](https://www.localstack.cloud/), where developers could use Messaging (SQS\SNS), Storage (S3), Database (DynamoDB), Security (Secrets), Compute (Lambda) etc. This has put the Azure community at a disadvantage when it came to prototyping solutions without having to incur a cost or setup resources that needed to be removed upon demo completion.

In the past, I personally had to spin up Azure Resources to demo a new feature while onsite with a client – having to contend with security firewalls and passwords was troublesome but not a showstopper but then seeing how I could use LocalStack and demo on my laptop – was just so much easier!

The attached code is a scalable solution you can demo locally to your clients on-site, without any overheads or costs associated – architectural overview below:

Where multiple clients are updated by SignalR, to new messages posted to the Message Queue, which is in turn monitored by an Azure Function – this makes for a decoupled and scalable solution – all demonstrated locally.

A diagram of a software server

AI-generated content may be incorrect.

# Project Structure

The projects within the ([repo](https://github.com/Bert0Neill/AspireAzuriteDemo)) solution are individually simplistic in themselves, but put them together and you have a serverless solution that can scale-up and monitor\consume\process millions of messages in a Queue\Topic and inform clients of any new messages. By using SignalR, you can tailor what client see’s what message etc.

* Azurite.AppHost → Aspire Orchestration (Server)
* Azurite.API → Minimal .Net Web API (Server)
* Azurite.BlazorWasmApp → Browser (Client C#)
* Azurite.SignalR → Maintain Browser Sessions (Server)
* Azurite.Fnx\_MonitorServicebusQueue → Monitor Message Queue (Server)

A screenshot of a computer

AI-generated content may be incorrect.

# Aspire Service Bus NuGet Package

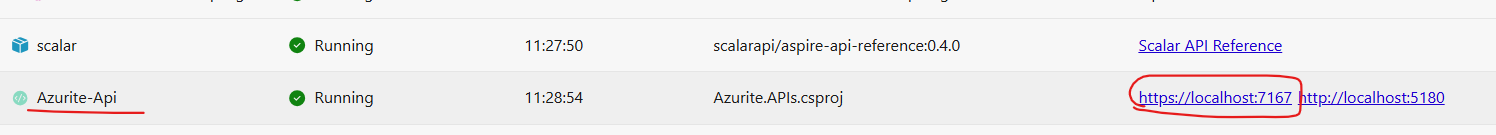
Ensure your *Aspire project* has a reference to the [Aspire.Azure.Messaging.ServiceBus](https://www.nuget.org/packages/Aspire.Azure.Messaging.ServiceBus/9.4.1#show-readme-container) NuGet package:A close up of a text

AI-generated content may be incorrect.

This allows Aspire to define a Service Bus as a resource (in our case an emulator service bus) and inject the connection string into the projects that actually need them (API, Azure Fnx. or a Worker Service for e.g.).

# Create Web API Function

In Aspire, click on the https link:



This will bring up a page with the URL <https://localhost:7167> – append /scalar/v1 to the end of the URL to display the Scalar testing interface: - <https://localhost:7167/scalar/v1>

If you scroll down, you will see a Send API method, click the “Test Request” button (right hand side), **NB:** that in Scalar you can copy the C#\JavaScript etc. example code to use inside your client method, to make the call\response.

A screenshot of a computer

AI-generated content may be incorrect.

Enter some text to be pushed onto the Service Bus Emulator and click Send

A screenshot of a computer

AI-generated content may be incorrect.

Api Send method, that will push the message onto the Service Bus Emulator:

A screenshot of a computer

AI-generated content may be incorrect.

# Create Azure Function

The Fnx Trigger code that is monitoring the Service Bus for messages, and then calls the SignalR extension to push the message onto all connected clients (to display in their browser).

A screen shot of a computer

AI-generated content may be incorrect.

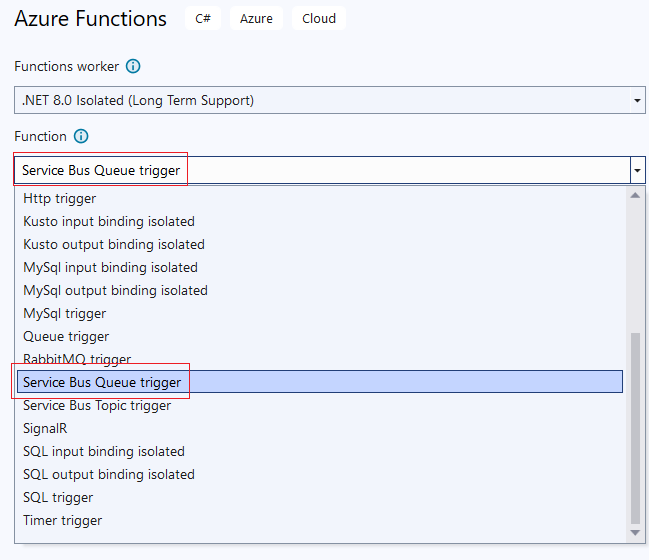
# Create Azure SignalR Emulator

# Create Azure Storage Emulator

# Miscellaneous

## When Adding Your Azure Function Project

Ensure that you Function is a Queue or Topic trigger type:



## Acronym

|  |  |
| --- | --- |
| Abbreviation | Meaning |
| EF | Entity Framework |
| EFC | Entity Framework Core |
| SQL | Structured Query Language |
| TSQL | Transact Structured Query Language |