BY BRYAN HOGAN • 2022-10-17

TWITTER FACEBOOK LINKEDIN

How to containerize your ASP.NET Core application and SQL Server with Docker

Docker + ASP.NET Core + SQL Server
How to containerize your ASP.NET Core application along your SQL Server container with Docker

In my previous blog post, I showed how to deploy SQL Server in a Docker container, then connect to it from a .NET Web API application using Entity Framework Core.

In this post, you're going to build on the previous post and learn how to deploy a .NET Web API application in a Docker container and connect it to SQL Server running in a container also.

Prerequisites

You will need the following things in this tutorial:

- A Windows, Linux, or Mac machine
- .NET 6 SDK (or newer)
- A .NET code editor or IDE (e.g. VS Code with the C# plugin, Visual Studio, or JetBrains Rider)
- Docker Engine (you can install the engine using Docker Desktop (Windows, macOS, and Linux), Colima (macOS and Linux), or manually on any OS.
- Some experience with ASP.NET Core MVC or Web API and EF is recommended, but not required.

Get Started

To get started, clone the repository from the previous post, and open the project in your favorite IDE.

```
git clone https://github.com/bryanjhogan/WebApiEntityFrameworkDockerSqlServer.git
cd WebApiEntityFrameworkDockerSqlServer
```

It has a working .NET Web API application that connects to a SQL Server database, seeds, and queries the database for a list of products.

To deploy the Web API application in a Docker container and connect it to the SQL Server container, you'll need to do a few things.

1. Add a Dockerfile to the Web API project
2. Change the connection string to the database
3. Build the Docker image for the Web API application
4. Create a Docker Compose file to start the Web API application container and the SQL Server container
5. Start the containers

Step 1: Add a Dockerfile to the Web API project

In the *WebApiEntityFrameworkDockerSqlServer* directory, create a file named *Dockerfile* and add the following content to it:

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app
EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY ["WebApiEntityFrameworkDockerSqlServer.csproj", "."]
RUN dotnet restore "./WebApiEntityFrameworkDockerSqlServer.csproj"
COPY . .
WORKDIR "/src/."
RUN dotnet build "WebApiEntityFrameworkDockerSqlServer.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "WebApiEntityFrameworkDockerSqlServer.csproj" -c Release -o /app/publish /p:UseAppHost=f

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "WebApiEntityFrameworkDockerSqlServer.dll"]
```

This *Dockerfile* performs a multi-stage build. The content of the above file is what Visual Studio would give you if right-clicked on the project and selected *Add > Docker Support...*, and chose Target OS as Linux.

Briefly (and somewhat simplified), it pulls down two Docker images, one to build the .NET application (that image includes the .NET SDK), and one to run it (that image has .NET ASP.NET runtime only, not the SDK).

Once the application is built using the image with the SDK, it is copied to the image with the runtime. This second image is the one you will use to run the application inside the container.

The application will run on port 80, this will be exposed to your computer via the *docker-compose.yml* file.

To learn more about .NET on Docker, I recommend the excellent series by Steve Gordon.

Step 2: Change the connection string to the database

In the previous post, the connection string pointed to *localhost* because the SQL Server container was running on the same machine as the Web API application, and had port 1433 exposed to your machine. This lets the Web API application connect to the SQL Server container via *localhost:1433*.

But now, both the Web API application and the SQL Server container are running in their own containers, but they will be on the same Docker network. Connecting from one container to another on the same Docker network is done by using the container name as the hostname.

Open the *appsettings.Development.json* and cut the text from there, and paste it into *appsettings.json*. Within the connection string, change the server name from `localhost` to `sql_server2022` .

You should now have an *appsettings.json* file that looks like this:

```
{
  "ConnectionStrings": {
    "SalesDb": "Server=sql_server2022;Database=SalesDb;User Id=SA;Password=A&VeryComplex123Password;Multipl
  }
}
```

Step 3: Build the Docker image for the Web API application

Open a terminal and navigate to the folder where the *Dockerfile* is located.

Run the following command to build the Docker image for the Web API application:

```
docker build -t web_api .
```

The `-t` flag tags the image with a name. In this case, the name is **web_api**. You will use this name in the *docker-compose.yml* file to reference the image.

This will take a little while to build. Once complete, you can run the following command to see the image:

```
docker image ls
```

The output should look like this:

```
REPOSITORY                                         TAG          IMAGE ID       CREATED              SIZE
web_api                                            latest       b8a84144c908   About a minute ago   222MB
```

4. Create a Docker Compose file to start the Web API application container, and the SQL Server container

Create a file named *docker-compose.yml* (it doesn't matter where the file is located, but I suggest placing it in the same directory as your *Dockerfile)* and add the following content to it:

```yaml
version: "3.9"  # optional since v1.27.0
services:
  web_api:
    image: web_api
    container_name: web_api_application
    ports:
      - "5000:80"
  sql:
    image: "mcr.microsoft.com/mssql/server:2022-latest"
    container_name: sql_server2022
    ports: # not actually needed, because the two services are on the same network
      - "1433:1433"
    environment:
      - ACCEPT_EULA=y
      - SA_PASSWORD=A&VeryComplex123Password
```

This file defines two services.

The first service `web_api` , uses the image named `web_api` to create a container named `web_api_application` . It maps port 80 inside the container to port 5000 on your computer.

The second service `sql` , uses the image `mcr.microsoft.com/mssql/server:2022-latest` to create a container named `sql_server2022` . It maps port 1433 inside the container to port 1433 on your computer. As mentioned above, this is not needed, but it is useful if you want to connect to the database from your computer.

It also sets the environment variables `ACCEPT_EULA` and `SA_PASSWORD` to `y` and `A&VeryComplex123Password` respectively.

Remember, you changed the connection string to point to a server named `sql_server2022` , that is the name of the SQL Server container defined in this file.

> By default, ASP.NET Core also loads environment variables into its configuration. This allows you to override configuration baked into the Docker image using environment variables.

You don't need to define any network, `docker-compose` will do this for you and place both containers in the same network.

In steps 3 and 4, you separated out building the Web API application image from starting the container via the *docker-compse.yaml* file. You can also use the *docker-compse.yaml* file to build the image and start the container for you in a single step:

```yaml
version: "3.9"  # optional since v1.27.0
services:
  web_api:
    build: . # build the Docker image
    container_name: web_api_application
    ports:
      - "5000:80"
  sql:
    image: "mcr.microsoft.com/mssql/server:2022-latest"
    container_name: sql_server2022
    ports: # not actually needed, because the two services are on the same network.
      - "1433:1433"
    environment:
      - ACCEPT_EULA=y
      - SA_PASSWORD=A&VeryComplex123Password
```

5. Start the containers

To start the two containers, run the following command:

```
docker-compose up
```

It will take a moment for them to start, especially if you need to pull the referenced images from Docker Hub.

In your terminal, you will see output from both containers as they start up. Watch for any errors or exceptions, but if all goes well, you can open *http://localhost:5000/products* in your browser, and you should see the same output as in the previous post.

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

Conclusion

In this post, you learned how to run a .NET Web API application and your SQL Server in Docker containers using the docker-compose tool.

You could upload the image of your Web API application to Docker Hub, or another image repository to share with your peers.

Bryan Hogan is a blogger, podcaster, Microsoft MVP, and Pluralsight author. He has been working on .NET for almost 20 years. You can reach him on Twitter @bryanjhogan.

Rate this post

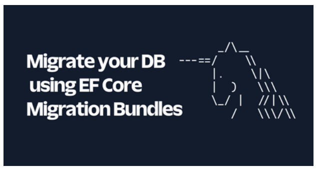
AUTHORS Bryan Hogan

REVIEWERS Niels Swimberghe

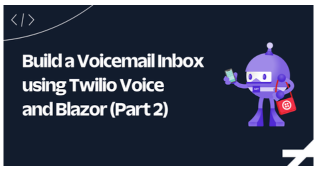
Related Posts

Review and catalog text messages with Cohere Classify and .NET
cohere

Review and catalog text messages with Cohere Classify and .NET
Jun 13, 2023
Learn how to classify incoming text messages via WhatsApp using Cohere Classify and ASP.NET Core.



Migrate your database using Entity Framework Core Migration Bundles
Jun 12, 2023
Learn how to use EF Core's new Migration Bundles feature, how to generate bundles, and how to execute them to migrate your databases.



Build a Voicemail Inbox using Twilio Voice and Blazor (Part 2)
Jun 05, 2023
Learn how to create a Blazor WebAssembly application to manage voicemail recordings made to your Twilio phone number.

.NET TWILIOVOICES DOCKER C# SQL ASP.NET CORE

Developer stories to your inbox.

Subscribe to the Developer Digest, a monthly dose of all things code.

Enter your email…

You may unsubscribe at any time using the unsubscribe link in the digest email. See our privacy policy for more information.

NEW!

Tutorials

Sample applications that cover common use cases in a variety of languages. Download, test drive, and tweak them yourself.

Get started

START FOR FREE

Not ready yet? Talk to an expert.