ASP.NET Core Docker HTTPS With Multiple Projects and Postgres

Published 2020-06-08

Photo by Irene Kredenets on Unsplash

It's officially summer in Wisconsin! OK, not quite yet but we did just have our first day that broke 90 F. And there's no better time to stay inside and create Docker containers!

This is going to be a very specific tutorial on getting docker running your ASP.NET Core application using https.

It will be more of a walkthrough on how I configured my docker files to support my .NET solution that has multiple projects as well as Postgres database running in separate container. I also used EF Core as my ORM.

Project Structure 💼

The is the basic structure I have for my solution. Notice the `docker-compose*` files are up under `src/`, while the API project has it's own `Dockerfile`.

```
RootFolder
  - src
    - API (project) # this is the WebAPI that gets launched
      - Dockerfile
    - Common (project)
      - Common.csproj
    - Core (project)
      - Core.csproj
    - Infrastructure (project)
      - Infrastructure.csproj
    docker-compose.yml
    docker-compose.debug.yml
    solutionfile.sln
  - tests
    - API.Tests (project)
    - Core.Tests (project)
```

Enable HTTPS in Startup.cs

First let's look at `Startup.cs` in the WebAPI. A couple things need configured.

`UseHttpsRedirection` tells something to use https

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment
{
    app.UseHttpsRedirection();
```

Add the `DbContext`s services to the container. In my case, I have one context for users, and one for everything else.

Note the server and port. This will be specified in our docker-compose files later.

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddDbContext<ApplicationContext>(options =>
      options.UseNpgsql("Server=db;Port=5432;Database=petcrm;User I

  services.AddDbContext<IdentityContext>(options =>
                    options.UseNpgsql("Server=db;Port=5432;Database=p
```

Setup EF Core Migrations □

If you haven't already, create your migrations. I wanted my migrations in my Infrastructure project, since's that's where the responsibility resides.

First I had to add an inner class to my DbContexts. The IdentityContext looks the same, just replace `Application` with `Identity`.

```
public class ApplicationContext : DbContext
{
```

```
public class ApplicationContextDesignFactory : IDesignTimeDbConte
{
    public ApplicationContext CreateDbContext(string[] args)
    {
        var optionsBuilder = new DbContextOptionsBuilder<Applicat
            .UseNpgsql("Server=db;Port=5432;Database=petcrm;User
        return new ApplicationContext(optionsBuilder.Options);
    }
}
```

Since I had two contexts, I had to add both separately while referencing a startup project (API) so the dotnet cli could create instances of the contexts.

```
# src/Infrastructure
$ dotnet ef migrations add --context IdentityContext InitialIdentit
$ dotnet ef migrations add --context ApplicationContext InitialAppl
```

I wanted to apply the migrations at runtime since Postgres would be running in a container that didn't exist yet. `DbContext.Database` has a `Migrate` method and it needs to be run after the webhost is built but before it runs, making `Program.cs` a suitable place to do this.

I created an extension method to not clutter up Program with too much code.

```
public static class MigrationHelper
{
    public static IHost MigrateDatabase<T>(this IHost host) where T
    {
        using (var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;
            try
            {
                var db = services.GetRequiredService<T>();
                db.Database.Migrate();
            }
            catch (Exception ex)
            {
                var logger = services.GetRequiredService<ILogger<Pr
                logger.LogError(ex, $"An error occurred while migra
            }
        }
        return host;
    }
}
```

Then in `Program.cs` after the host is built, run the extension method with each context.

```
public class Program
{
    public async static Task Main(string[] args)
    {
        var host = CreateHostBuilder(args).Build();

        host.MigrateDatabase<ApplicationContext>();
        host.MigrateDatabase<IdentityContext>();

        host.Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
}
```

Oof, more work than expected, but it turned out OK :)

Docker setup 🐳

Let's look at the API project's `Dockerfile` first. With Visual Studio, add docker support to the project to build out the shell then make some adjustments as needed.

Since I have multiple projects, I needed copy commands for each before restoring and building. Here's what I ended up with.

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
WORKDIR /src
COPY ["API/API.csproj", "API/"]
COPY ["Core/Core.csproj", "Core/"]
COPY ["Infrastructure/Infrastructure.csproj", "Infrastructure/"]
RUN dotnet restore "API/API.csproj"
COPY . .
WORKDIR "/src/API"
RUN dotnet build "API.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "API.csproj" -c Release -o /app/publish

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

Docker Compose Setup for HTTPS 🔒

First let's put some certs in place. Here is a refernence on it:

https://docs.microsoft.com/en-us/aspnet/core/security/docker-compose-https?view=aspnetcore-3.1#windows-using-linux-containers

The docs say you can use .Net CLI:

```
$ dotnet dev-certs https -ep %USERPROFILE%\.aspnet\https\aspnetapp.
$ dotnet dev-certs https --trust
```

You may not need to, but if you already have a cert in there from other development you can clean it out first:

```
$ dotnet dev-certs https --clean
```

Now we'll just look at the `docker-compose.debug.yml` file. Note that it is run from the *solution root folder* so that the docker has access to the other projects as well.

```
$ docker-compose -f "src\docker-compose.debug.yml" up -d --build
```

And here is the compose debug configuration. (*NOTE* this is for development only, production settings will likely be different, like a more complex password for starters)

```
# Please refer https://aka.ms/HTTPSinContainer on how to setup an h
version: '3.4'

services:
  api:
    image: api
    build:
      context: .
      dockerfile: API/Dockerfile # the Dockerfile is stored in the
    ports:
      - 5000:80 # port mapping
      - 5001:443
    depends_on:
      - db
    environment:
      - ASPNETCORE_ENVIRONMENT=Development # debug runs in developm
      - ASPNETCORE_URLS=https://+:443;http://+:80
      - ASPNETCORE_Kestrel__Certificates__Default__Password={passwo
      - ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspn
    volumes:
      - ~/.vsdbg:/remote_debugger:rw
      - ~/.aspnet/https:/https:ro

  db: # this is used as the host in your connection string
    image: postgres
    container_name: 'postgres_container'
    environment:
      - POSTGRES_USER=app_user
      - POSTGRES_PASSWORD=app_user
      - POSTGRES_DB=dbname
    volumes:
      - pgdata:/var/lib/postgresql/data # this is where your data p
    ports:
      - 5432:5432 # use this port in your connection string

volumes:
  pgdata:
```

3...2...1... 🚀

Everything should be in place, now time to fire it up!

Run the `docker-compose` command from above shown below again. Remember to be up one directory from the `docker-compose.debug.yml` file.

```
$ docker-compose -f "src\docker-compose.debug.yml" up -d --build
```

Your terminal will loop through each step of the `Dockerfile` - remember your compose file referenced the one in the API project. I recommend you carefully read through the output to see all that docker is doing when building the containers.

If you have docker desktop, you should see your containers running!

Or you can show docker processes with `docker ps`.

```
$ docker ps
CONTAINER ID        IMAGE                   COMMAND                    CR
bde76d1c4138        api                     "dotnet API.dll"           Ab
b93643bdf5f0        postgres                "docker-entrypoint.s…"     Ab
```

Notice the ports! The API can be accessed through HTTPS at https://localhost:5001

Inside the container, it's on port 443.

Postgres is on 5432 inside and out of the container.

Happy coding 🍦

#asp.net core #docker

ABOUT THE AUTHOR

This article was written by Josiah Mortenson.
Feel free to reach out at josiah.mortenson@gmail.com