.NET Core Dev in Docker with Live Compilation and Debugging

Back

Published on March 12, 2021

.NET Core C# Docker

// Introduction

I have typically always used Visual Studio for .NET development, mainly out of habit, and it pretty much being a necessity for building apps with the .NET Framework. But also because it is a very powerful IDE for development, debugging and running tests, with minimal configuration required.

In contrast, I do all of my JavaScript development using Visual Studio Code, which is likely no surprise to most people. My apps are typically all setup to run inside Docker with live reload enabled, meaning any of my changes are reflected almost instantly, without the need for re-building any containers. **This is a really efficient way of developing.**

I was keen to try and adopt a similarly efficient setup for my .NET development. I had tried VS Code for .NET Core in the past, but it never quite worked how I wanted, and I always felt a little less productive then I would be using Visual Studio.

This post aims to explain how to setup an efficient local .NET Core development environment in Docker with debugging and Live Compilation (see changes immediately without needing to re-build the Docker container) using VS Code.

// Setting Up the Docker Image

The first step is to add our initial Dockerfile for running the .NET Core WebAPI inside a Container. The Dockerfile, as shown below, is nice and simple, and as a result, is really only suitable for building development Containers; a Dockerfile for a production application would look more complex than this, including separate steps to restore, build and publish the application, but that's beyond the scope of this blog post.

```
# 1
FROM mcr.microsoft.com/dotnet/core/sdk:2.1 AS build

# 2
WORKDIR /app
COPY . .

# 3
ENV ASPNETCORE_URLS=http://*:5000
ENV ASPNETCORE_ENVIRONMENT=Development

# 4
WORKDIR /app/WebAPI

# 5
ENTRYPOINT ["dotnet", "run"]
```

The main steps in the Dockerfile are:

1. Base the image on the .NET Core 2.1 SDK. The SDK base image has all of the tools we need to compile and run the application.
2. Copy all of the project files into a new `/app` directory within the image.
3. Set a couple of environment variables needed for running the application in development mode.
4. Set the working directory to the root of the web application where the project files are.
5. Set the command to run when the Container is started. In this case, `dotnet run`, which performs a restore and build of the application before starting it.

Whenever I'm using Docker, I always use Docker Compose for running Containers. Below is an example Docker Compose file for running the application:

```
version: "3"
services:
  webapi:
    container_name: webapi
    build:
```

```
      context: ./
      dockerfile: ./Dockerfile.dev
    ports:
      - 5000:5000
```

Again, this is nice and simple, specifying a name for the resulting container, the build context and Dockerfile to use, and finally the port bindings.

Testing the Changes

The app can be spun up by running the following command:

```
docker-compose up --build webapi
```

In under a minute, your app should be up and running in a Docker Container and ready to receive requests.

The next step is to add Live Compilation, so that any changes to the source code are automatically built and served up, without needing to stop and re-build the image again.

// Adding Live Compilation

As mentioned previously, Live Compilation allows for changes in your source code to be reflected inside your running Docker Container almost straight away, without needing to rebuild the image again and again. This makes developing a .NET Core app in Docker a much more efficient and pleasant experience.

Adding Live Compilation requires just 2 simple changes to the current setup.

Enabling the File Watcher

Firstly, change the Docker Container startup command to include the `watch` parameter, like so: `ENTRYPOINT ["dotnet", "watch", "run"]`. This uses the file watcher tool built into the .NET CLI, which will watch for file changes in your project, and run the command it is instructed to - in this case `run`, the same command we were using before to build and run the app.

Mounting the Source Files as a Volume in the Container

Secondly, given the application is running inside Docker, we need to mount the source files as a volume in the Container. This allows the Container to still be connected to the project files on the host machine, allowing the File Watcher to detect any changes made. Without the volume mounting, the File Watcher would never know of any changes, and so would not know when to

execute the `run` command again, nor would it have any of the updated source files in the image. To mount the source files as a volume, we need to add a volume mapping in the Docker Compose file from the `server` directory on the host machine which holds the WebAPI project, to the `app` directory inside the Container. The Docker Compose file should now look like this:

```
version: "3"
services:
  webapi:
    container_name: webapi
    build:
      context: ./
      dockerfile: ./Dockerfile.dev
    volumes:
      - ./:/app
    ports:
      - 5000:5000
```

Testing the Changes

With these changes in place, running the app again should result in a message in the logs saying that the File Watcher is enabled.

Try changing one of your source files — you should see the File Watcher detect which file has changed and will rebuild the app automatically with the latest change applied.

// Enabling Debugging in the Docker Container

The final step is to get debugging working within the Container such that we can set breakpoints and step through the code, as you would typically do when running in Visual Studio. This requires two changes to our setup.

Installing the Microsoft Debugger

The first change to make is to install the Microsoft debugger for .NET Core in the image which the remote debugger in VS Code can be attached to. Add the following `RUN` command right after the initial `FROM` command in the Dockerfile:

```
FROM mcr.microsoft.com/dotnet/core/sdk:2.1 AS build

RUN apt-get update \
    && apt-get install unzip \
    && curl -sSL https://aka.ms/getvsdbgsh | /bin/sh /dev/stdin
```

```
    # Remainder removed for brevity.
```

This command looks like it's doing quite a lot, but it's essentially updating the local package list, installing the `unzip` package and then finally downloading a shell script which is used to install the debugger by piping it to the shell executable, storing the output in `/vsdbg`.

Creating a VSCode Launch Profile

The second step is to create a VS Code launch profile which can be used to attach the remote debugger to the debugger we just installed in the running Container. Add the following profile to the `configurations` array inside `./.vscode/launch.json`.

```
{
    "name": "Debug .NET Core in Docker",
    "type": "coreclr",
    "request": "attach",
    "processId": "${command:pickRemoteProcess}",
    "sourceFileMap": {
        "/app": "${workspaceRoot}/"
    },
    "pipeTransport": {
        "pipeCwd": "${workspaceRoot}",
        "pipeProgram": "docker",
        "pipeArgs": ["exec", "-i", "webapi"],
        "quoteArgs": false,
        "debuggerPath": "/vsdbg/vsdbg"
    }
}
```

Unless you're really interested, it's not important to get bogged down with the details of what is being configured here, but the key thing to note is the value being supplied to `debuggerPath` — it is the path to the `vsdbg` debugger we installed earlier. Change `"/app": "${workspaceRoot}/"` and `"pipeArgs": ["exec", "-i", "webapi"]` as appropriate based on your own project setup.

If you are interested to learn more about VSCode Launch Profiles, the official docs has a detailed breakdown of each option and how they can be configured.

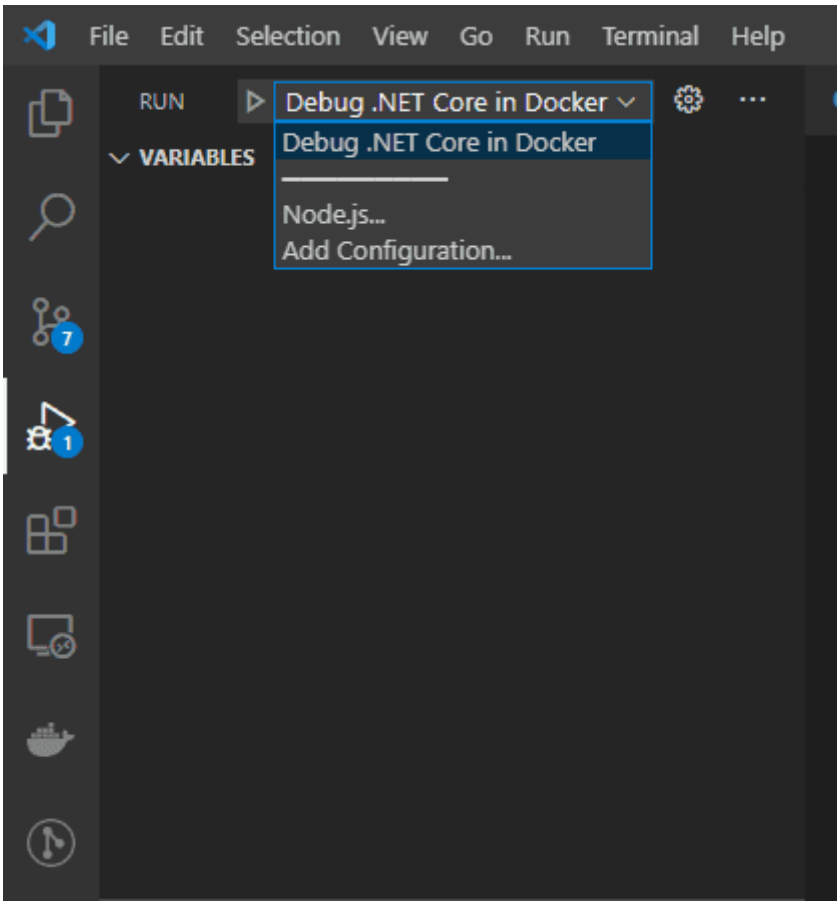
That's all we need to enable debugging inside the Docker Container.

Testing the Changes

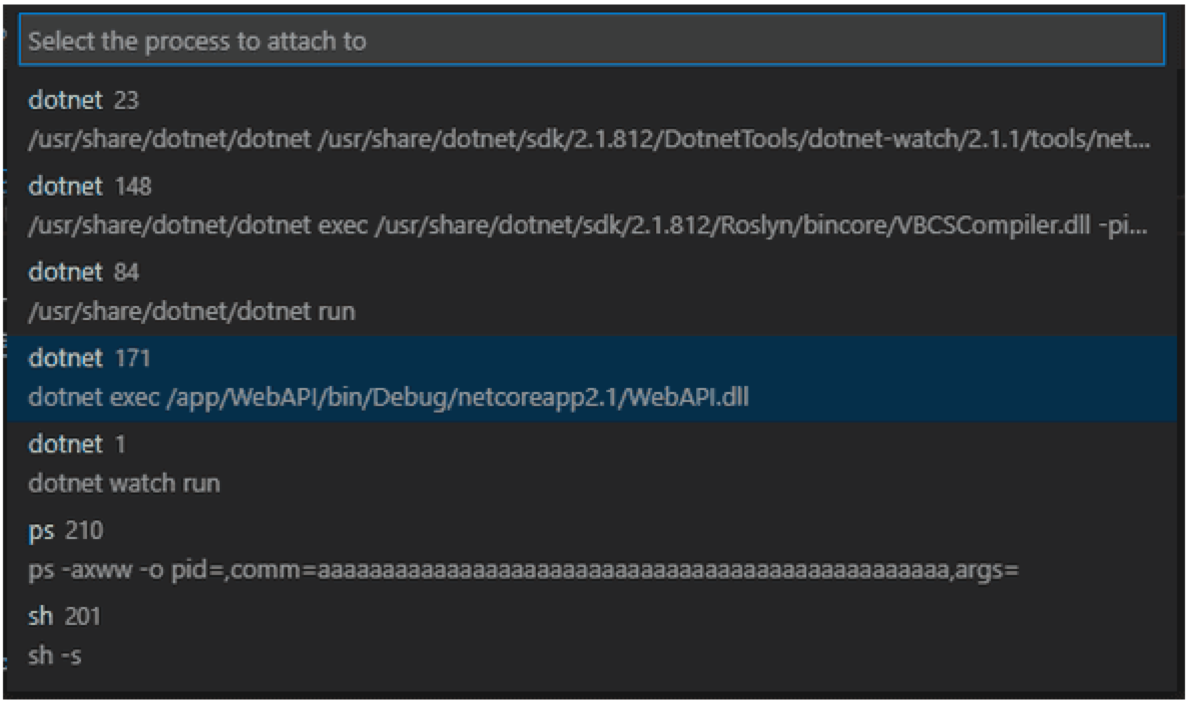
To test these changes, first start the app as before using Docker Compose:

```
docker-compose up --build webapi
```

Once the app has started successfully, head over to the Run tab in VSCode and choose the new launch profile we have just created and click the “Start Debugging” green button.



You will now be presented with the option to choose which process to attach the remote debugger to. Choose the process which is executing your DLL, which in my case is `WebAPI.dll`.



After a few moments, the remote debugger should be attached, allowing you to hit any breakpoints you have set.

// Wrapping Up

I have covered the steps required to have a .NET Core Web API application running inside Docker, with Live Compilation and debugging capabilities.

This has generally worked really well for me when developing locally as an alternative to using Visual Studio, with the Live Compilation occasionally not being able to recover if there is a build failure whilst the remote debugger is attached.

With this setup, you get the benefit of a Visual Studio-like development experience, but using more modern development practices, like running in Docker.

Improving Gatsby Performance: Replacing React with Preact

2020: In Review