CHAPTER 8

Containerized Development

Many of the problems that Docker solves in production also occur during development, especially for large teams working on projects that have multiple components. It can be hard to make sure that every developer has the right version of the tool chain, the runtime libraries, third-party packages, database servers, and schemas and that they are all set up consistently. Add in differences in operating systems and patch levels and it becomes an impossible task.

Using containers for the development environment brings consistency and uniformity to the development process, while still allowing individual developers the freedom to customize their development tools to suit their personal styles. In fact, one of the attractions of a containerized environment is that it standardizes only what is important, without trying to force every developer to work in the same way.

It may seem odd to look at containerized development at the end of a book, but it is only when you understand how the different features fit together that the use of Docker during development can be explained. In this chapter, I show you how to set up a containerized development environment and how to debug an application that is running in a container. Table 8-1 puts containerized development in context.

Table 8-1. *Putting Containerized Development in Context*

Question	Answer
What is it?	Containerized development moves part of the development environment into a Docker container.
Why is it useful?	Containerized development compiles and executes a project using the .NET Core tools running inside a container, which means they will be consistent with the production environment and consistent between developers.
How is it used?	A base image that contains the .NET Core Software Development Kit is used to create an image. A container is created from that image, using a Docker volume to provide access to the project files, which are stored on the host operating system so that they can be put under regular revision control.
Are there any pitfalls or limitations?	Some additional configuration is required to prepare a project for containerization.
Are there any alternatives?	You do not have to containerize development for a project to use Docker to deploy it.

A. Freeman, *Essential Docker for ASP.NET Core MVC*, DOI 10.1007/978-1-4842-2778-7_8

Table 8-2 summarizes the chapter.

Table 8-2. *Chapter Summary*

Problem	Solution	Listing
Build a project in a container	Configure the project to build and run automatically, using a volume to access the project files	1–11
Prepare a project for debugging in a container	Create and run a container that includes the remote debugger	12–17
Debug in a container using Visual Studio 2017	Create an XML configuration file and use it to start the debugger from the command window	18–20
Debug in a container using Visual Studio Code	Configure the debugger using the `launch.json` file	21, 22

Preparing for This Chapter

This chapter depends on the ExampleApp MVC project created in Chapter 3 and modified in the chapters since. If you don't want to work through the process of creating the example, you can get the project as part of the free source code download that accompanies this book. See `apress.com` for details.

To ensure that there is no conflict with examples from previous chapters, run the commands shown in Listing 8-1 to remove the Docker containers, networks, and volumes. Ignore any errors or warnings these commands produce.

Listing 8-1. Removing the Docker Components

```
docker rm -f $(docker ps -aq)
docker network rm $(docker network ls -q)
docker volume rm $(docker volume ls -q)
```

Run the commands shown in Listing 8-2 if you are a Linux user and you used your development machine as the manager of a swarm in Chapter 7.

Listing 8-2. Removing the Services and Leaving the Swarm

```
docker stack rm exampleapp
docker swarm leave --force
```

Using Development Mode to Prepare the Database

Previous chapters used a range of techniques to prepare the database for the application, dealing with the need to apply Entity Framework Core migrations and generate seed data.

The situation is simpler in development because there is no production data that can be lost. To that end, modify the `Startup` class, as shown in Listing 8-3, so that the database is prepared automatically when the application starts in the `Development` environment.

Listing 8-3. Enabling Database Updates in the Startup.cs File in the ExampleApp Folder

```
...
public void Configure(IApplicationBuilder app,
        IHostingEnvironment env, ILoggerFactory loggerFactory) {

    loggerFactory.AddConsole();
    app.UseDeveloperExceptionPage();
    app.UseStatusCodePages();
    app.UseStaticFiles();
    app.UseMvcWithDefaultRoute();

    if (env.IsDevelopment()) {
        SeedData.EnsurePopulated(app);
    }
}
...
```

Understanding Containerized ASP.NET Core Development

The roles of application and data are reversed when compared to the deployment containers created in earlier chapters. It is the tool chain—the compiler, the runtime, the debugger—that becomes the application and lives inside the container. The data files are the ASP.NET Core MVC project files, which exist outside of the container in a Docker volume. The Docker volume used for development containers uses a folder from the host operating system to provide the contents of a directory inside the container. This is similar to the way that the load balancer configuration file was set up in earlier chapters but uses an entire folder rather than a single file. Figure 8-1 shows how the pieces fit together in a containerized development environment.

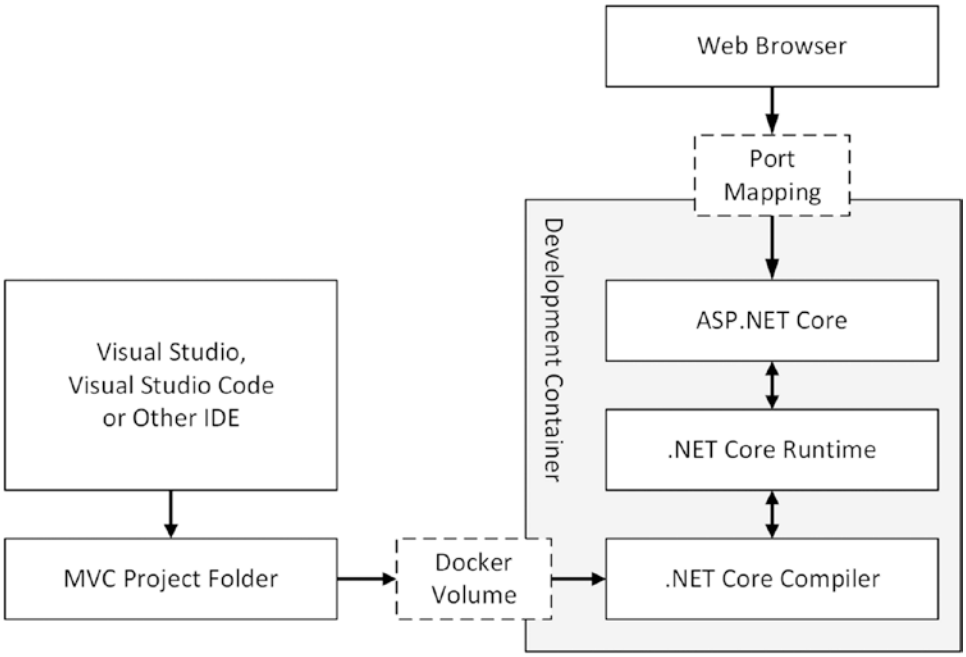

Figure 8-1. *Containerized development*

Using a volume to mount a host folder means that ASP.NET Core MVC project files can be edited using Visual Studio or Visual Studio Code as normal and then compiled inside the development container using the .NET Core compiler. The development container also includes the .NET Core runtime, which can execute the application and expose it for testing through a port mapping to the host operating system. The project files can be managed outside of the container, using regular source code control tools like Git or Subversion, while the tools and execution environment are standardized through the container, whose Docker file can also be placed under version control.

Setting Up Containerized Development

Containerized development is still a relatively new idea and can be a little awkward, especially when it comes to debugging. The quality of tooling for containerized development will improve as containers become more widely used, but for the moment some careful configuration is required to create the development environment and get it working. In the sections that follow, I start by setting up a container for developing and testing an ASP.NET MVC Core application, and then, with a little extra work, I demonstrate how use the Visual Studio and Visual Studio Code debuggers.

Adding the DotNet Watcher Package

The first challenge in setting up a development container is making sure that it reflects the changes that you make to the code files. ASP.NET Core MVC responds immediately when one of its Razor views changes but doesn't detect changes in the C# classes.

The `Microsoft.DotNet.Watcher.Tools` package is used to monitor a project from inside the container and restart the .NET Core runtime when a file change is detected, ensuring that changes made by the developer are reflected inside the container.

Add the package shown in Listing 8-4 to the `ExampleApp.csproj` file.

Listing 8-4. Adding a Package in the ExampleApp.csproj File in the ExampleApp Folder

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

  <PropertyGroup>
    <TargetFramework>netcoreapp1.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.AspNetCore.Mvc" Version="1.1.2" />
    <PackageReference Include="Microsoft.AspNetCore.StaticFiles" Version="1.1.1" />
    <PackageReference Include="Microsoft.Extensions.Logging.Debug" Version="1.1.1" />
    <PackageReference Include="Microsoft.VisualStudio.Web.BrowserLink"
        Version="1.1.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="1.1.1" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools"
        Version="1.1.0" />
```

```
    <PackageReference Include="Pomelo.EntityFrameworkCore.MySql" Version="1.1.0" />
    <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet"
        Version="1.0.0" />
    <PackageReference Include="Microsoft.Extensions.Configuration.CommandLine"
        Version="1.1.1" />
    <DotNetCliToolReference Include="Microsoft.DotNet.Watcher.Tools"
         Version="1.0.0" />
  </ItemGroup>
</Project>
```

The change adds the version of the DotNet Watcher package that works with version 1.1.1 of .NET Core and ASP.NET Core and that is required for the examples in this book.

CONTAINERIZING A NEW PROJECT

The examples in this chapter show you how to create a containerized development environment for a project that is already in progress, but you can also create a container right at the start of the development process.

To do this, you need to create a project on the host operating system so that there is a `csproj` file that lists the NuGet packages you need. Once you have the `csproj` file, you can create and start the container and start adding the C# and Razor files that your project requires.

If you are using Visual Studio, you can use the New Project Wizard to create the project. If you are using Visual Studio Code (or you just prefer working with the command line), then use the following command to create a new project, where the name of the folder in which the command is run is used as the name for the project.

```
dotnet new mvc --language C# --auth None --framework netcoreapp1.1
```

Once the project has been created, open a new command prompt and navigate to the project. Add the DotNet Watcher package as shown in Listing 8-4 and any other packages that you require. You can then follow the process from Listing 8-5 onward.

Once you have a development container up and running, use you IDE to create the data model, controllers, and views that your application requires.

Creating the Development Image and Compose File

It is possible to create a development container using only the Docker command-line tools, but doing so requires careful attention to ensure that the configuration settings required by the development tools are correct. A more reliable approach, and one that helps ensure that all developers are working with the same environment, is to create a Docker file that is used in conjunction with a compose file that includes the volumes, software-defined networks, and any other containers that the application depends on.

Create a file called Dockerfile.dev in the ExampleApp folder and add the command settings shown in Listing 8-5. This Docker file will be used to create the image for development containers.

Listing 8-5. The Contents of the Dockerfile.dev File in the ExampleApp Folder

```
FROM microsoft/aspnetcore-build:1.1.1

COPY node_modules/wait-for-it.sh/bin/wait-for-it /tools/wait-for-it.sh

RUN chmod +x /tools/wait-for-it.sh

ENV DBHOST=dev_mysql WAITHOST=dev_mysql WAITPORT=3306

ENV DOTNET_USE_POLLING_FILE_WATCHER=true

EXPOSE 80/tcp

VOLUME /app

WORKDIR /app

ENTRYPOINT dotnet restore \
  && /tools/wait-for-it.sh $WAITHOST:$WAITPORT --timeout=0 \
  && dotnet watch run --environment=Development
```

The base image for this Docker file is microsoft/aspnetcore-build:1.1.1, which is provided by Microsoft and contains the .NET Core SDK rather than just the runtime contained in the images used in previous chapters. This means the compiler is available, which allows the code to be compiled inside the container.

The contents of the /app folder will be provided using a Docker volume when the container is created and will act as a bridge between the code editing done using the IDE and the .NET Core compiler and runtime inside the container. This is an important difference from the Docker files in earlier chapters, which copied the project files into the /app directory in the container's file system with a COPY command.

■ **Tip** The DotNet Watcher package requires that the DOTNET_USE_POLLING_FILE_WATCHER environment to be set to true when it is used in a container; otherwise, it won't be able to detect file changes.

Table 8-3 describes all the commands used in the development Docker file.

Table 8-3. *The Commands in the Development Docker File*

Name	Description
FROM	This command specifies the base image, which includes the .NET Core SDK so that the code in the project can be compiled inside the container.
COPY	This command copies the wait-for-it script into the container's file system. Previous examples have copied the file into the /app directory, but /tools is used in this example because any file copied into /app will not be available once the volume that contains the project files is mounted when the container is started.
RUN	This command executes the Linux chmod command so that the wait-for-it script can be executed when the container is started.
ENV	This command sets environment variables in the container. The DBHOST variable sets the name of the database for the Entity Framework Core connection string. The WAITHOST and WAITPORT variables are used by the wait-for-it package to make sure that the database is ready before the .NET Core runtime is started. The DOTNET_USE_POLLING_FILE_WATCHER must be set to true when the Dotnet Watcher package is used in a container.
EXPOSE	This command exposes a port so the application inside the container can receive network requests. In this example, port 80 is exposed so that HTTP requests can be received by the ASP.NET Core Kestrel server.
VOLUME	This command is used to specify that a volume will be used to provide the contents of a specified directory.
WORKDIR	This command sets the working directory for the container. In this example, the working directory is set to /app, which will contain the ASP.NET Core MVC project files.
ENTRYPOINT	This command tells Docker what to execute when the container is started. In this example, the wait-for-it script is used to wait for the MySQL database to be ready to receive connections, and then the DotNet Watcher package is used to start the .NET project in the Development environment.

The next step is to create a Docker Compose file that describes the overall environment required by the application. Create a file called docker-compose-dev.yml in the ExampleApp folder and add the content shown in Listing 8-6.

Listing 8-6. The Contents of the docker-compose-dev.yml File in the ExampleApp Folder

```
version: "3"

volumes:
  productdata:

networks:
  backend:

services:

  mysql:
    image: "mysql:8.0.0"
    volumes:
      - productdata:/var/lib/mysql
```

```
    networks:
      - backend
    environment:
      - MYSQL_ROOT_PASSWORD=mysecret
      - bind-address=0.0.0.0

  mvc:
    build:
      context: .
      dockerfile: Dockerfile.dev
    volumes:
      - .:/app
      - /app/obj
      - /app/bin
      - ~/.nuget:/root/.nuget
      - /root/.nuget/packages/.tools
    ports:
      - 3000:80
    networks:
      - backend
    environment:
      - DBHOST=mysql
      - WAITHOST=mysql
    depends_on:
      - mysql
```

The compose file is similar to the one used for production, with some changes to make development easier. HTTP requests will be received directly by the MVC application through a port mapped to the host operating system, without the use of a load balancer. You can add another software-defined network and a container for the load balancer if you want to re-create the environment from earlier chapters, but for most projects, working directly with the MVC container will be sufficient during development.

The most significant changes in Listing 8-6 are in the `volumes` section of the `mvc` service. In earlier chapters, the application was prepared for deployment using the `dotnet publish` command and then copied into the container's file system using a `COPY` command in the Docker file. This process isn't suited to development, where you want to see the effect of a code change without having to regenerate the Docker image and create a new container. The first `volume` entry in the Docker file solves this problem by sharing the local directory with the container and making its contents available for use in the `/app` folder, where they will be compiled and executed using the development tools in the .NET Core SDK.

```
...
volumes:
  - .:/app
  - /app/obj
  - /app/bin
  - ~/.nuget:/root/.nuget
  - /root/.nuget/packages/.tools
...
```

This means the container must be started in the `ExampleApp` folder so that the first volume provides the development container with the files it requires.

However, there are platform-specific files added to the `bin` and `obj` folders for the NuGet tooling packages, so the second and third volume entries tell Docker to create new volumes for those directories. Without these entries, the application would work either on the host or in the container but not both.

The second volume setting shares the `.nuget/packages` folder that is found in the current user's home directory.

```
...
volumes:
  - .:/app
  - /app/obj
  - /app/bin
  - ~/.nuget:/root/.nuget
  - /root/.nuget/packages/.tools
...
```

This is the location that is used to store NuGet packages when they are installed using the `dotnet restore` command, and sharing this folder with the container means that the container can use the packages you have installed on the host operating system so that you don't have to use `dotnet restore` each time you create a new development container. The final volumes entry creates a new volume for one of the folders into which NuGet will put the platform-specific files for the NuGet tools packages.

For quick reference, Table 8-4 describes the development-specific settings in the compose file.

Table 8-4. *The Development-Specific Configuration Settings in the Compose File*

Name	Description
ports	This setting maps port 3000 on the host operating system to port 80 within the MVC container. This will allow HTTP requests to be sent directly to the MVC container without needing to use a load balancer.
volumes	This setting is used to provide the development container with access to the project files and allow it to make use of the NuGet packages that have been installed on the host operating system.

Preparing for a Development Session

Ensure that all the NuGet packages required by the application are available on the host operating system by running the command shown in Listing 8-7 in the `ExampleApp` folder.

Listing 8-7. Updating the NuGet Packages

```
dotnet restore
```

Using a Docker volume to share the host operating system's NuGet packages with the development container is a useful technique, but you must ensure that they are up-to-date before you start a development container. The advantage of this approach, however, is that you don't have to restore the NuGet packages inside the container.

Once the package update has completed, run the command shown in Listing 8-8 in the `ExampleApp` folder to process the development compose file and build the image that will be used to create the container that will run the .NET Core development tools and the MVC application.

Listing 8-8. Processing the Development Compose File

```
docker-compose -f docker-compose-dev.yml -p dev build
```

■ **Note** If you are a Windows or macOS user, you must enable drive sharing, as described in Chapter 5, so that Docker can share the project directory with the container.

The `-f` argument is used to specify the compose file, without which the production file would be used.

The `-p` argument is used to override the prefix used for the names of the images, containers, and networks that will be created using the compose file. I have specified a prefix of `dev`, which ensures that I can run the development services without interfering with the production services I created in earlier chapters (which are prefixed with `exampleapp_`).

■ **Caution** The commands in the rest of the chapter rely on the `dev` prefix. If you use a different prefix when you run the `docker-compose build` command, then you will need to follow through that change everywhere you see a Docker component referred to by a name that starts `dev_`.

Starting a Development Session

To begin development, run the command shown in Listing 8-9 in the `ExampleApp` folder to scale up the `mvc` service.

Listing 8-9. Creating the Development Service

```
docker-compose -f docker-compose-dev.yml -p dev up mvc
```

Docker will process the compose file and follow the `depends_on` setting to determine that a container for MySQL is required, along with a software-defined network to connect them and a volume to store the database files. All four components will be created, and you will see the following output as Docker does its work and MySQL goes through its first-use initialization process:

```
...
Creating network "dev_backend" with the default driver
Creating volume "dev_productdata" with default driver
Creating dev_mysql_1
Creating dev_mvc_1
Attaching to dev_mvc_1
mvc_1    | wait-for-it.sh: waiting for mysql:3306 without a timeout
mvc_1    | wait-for-it.sh: mysql:3306 is available after 0 seconds
mvc_1    | watch : Started
mvc_1    | Starting ASP.NET...
mvc_1    | Hosting environment: Development
mvc_1    | Content root path: /app
mvc_1    | Now listening on: http://+:80
mvc_1    | Application started. Press Ctrl+C to shut down.
mvc_1    | Application is shutting down...
```

```
mvc_1    | watch : Exited
mvc_1    | watch : File changed: /app/Controllers/HomeController.cs
mvc_1    | watch : Started
mvc_1    | Starting ASP.NET...
mvc_1    | Hosting environment: Development
mvc_1    | Content root path: /app
mvc_1    | Now listening on: http://+:80
mvc_1    | Application started. Press Ctrl+C to shut down.
...
```

The advantage of specifying the name of a service when using the `docker-compose up` command is that the command prompt will only attach to the containers for the specified service. In this case, that means you will see the messages produced by the MVC container and not the verbose startup from the `MySQL` container.

Test that the application is running correctly by opening a new browser window and requesting `http://localhost:3000`, which corresponds to the port mapping configured for the MVC container in the compose file.

Once the application is running, use your IDE to open the example project on the host operating system, just as you have been doing in earlier chapters, and make the change shown in Listing 8-10 to alter the format of the message passed by the `Index` action method in the `Home` controller to its view.

Listing 8-10. Changing the Message in the HomeController.cs File in the ExampleApp/Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using ExampleApp.Models;
using Microsoft.Extensions.Configuration;

namespace ExampleApp.Controllers {
    public class HomeController : Controller {
        private IRepository repository;
        private string message;

        public HomeController(IRepository repo, IConfiguration config) {
            repository = repo;
            message =  $"Host ({config["HOSTNAME"]})";
        }

        public IActionResult Index() {
            ViewBag.Message = message;
            return View(repository.Products);
        }
    }
}
```

The changed file is contained within the host operating system folder that has been shared with the MVC container and that is used to provide the contents of the `/app` directory. The DotNet Watcher package detects the file change and restarts the .NET Core runtime. When the .NET Core runtime starts, it will also detect the changed file and recompile the project, after which the `Program` class is executed and ASP.NET Core is started. Reload the browser and you will see the changed response shown in Figure 8-2. (The message will be prefixed with `Swarm` if you are a Linux user and you followed the examples in Chapter 7.)

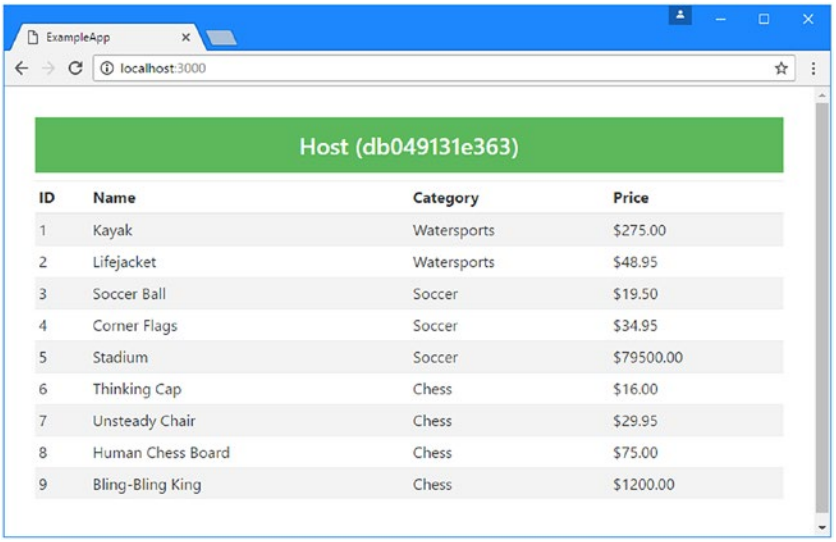

Figure 8-2. *Making a change to the example application*

■ **Tip** If you make a change that generates a compiler error, then the details of the problem will be displayed in the output from the development container. The DotNet Watcher package is smart enough to wait until the project compiles without error before starting the ASP.NET Core MVC application again.

When your development session ends, you can type `Control+C` at the command prompt to stop the MVC container, leaving the MySQL container running in the background. If you want to stop all the containers, run the command shown in Listing 8-11 in the `ExampleApp` folder. The containers are left in place and can be restarted using the command from Listing 8-9.

Listing 8-11. Stopping All Containers

```
docker-compose -f docker-compose-dev.yml -p dev stop
```

Setting Up Container Debugging

Visual Studio and Visual Studio Code can be configured to debug an application running in the container, although the process is a little awkward. The awkwardness arises because the debugger has to run inside the container but must be controlled from outside through the IDE, as shown in Figure 8-3.

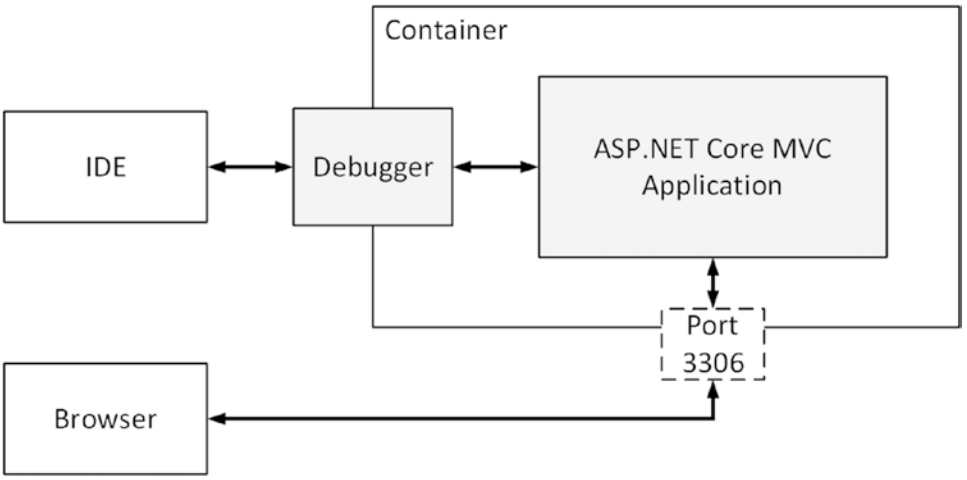

Figure 8-3. *Debugging in containerized development*

In the sections that follow, I describe the process of setting up the debugger in a container and configuring both IDEs to use it.

Creating the Debugging Docker File

The debugger isn't included in the base image used for the development container and must be added in the Docker file. Create a file called `Dockerfile.debug` in the `ExampleApp` folder and add the configuration settings shown in Listing 8-12.

Listing 8-12. The Contents of the Dockerfile.debug File in the ExampleApp Folder

```
FROM microsoft/aspnetcore-build:1.1.1

RUN apt-get update && apt-get install -y unzip

WORKDIR /clrdbg

RUN curl -SL \
        https://raw.githubusercontent.com/Microsoft/MIEngine/getclrdbg-release/scripts/GetClrDbg.sh \
         --output GetClrDbg.sh \
    && chmod 700 GetClrDbg.sh \
    && ./GetClrDbg.sh -v latest -l . \
    && rm GetClrDbg.sh
```

```
EXPOSE 80/tcp

VOLUME /app

WORKDIR /app

ENTRYPOINT echo "Restoring packages..." && dotnet restore \
       && echo "Building project..." && dotnet build \
        && echo "Ready for debugging." && sleep infinity
```

The process for installing the debugger involves three steps. First, the `apt-get` tool is used to install the unzip tool, which will be used to decompress the debugger installer. Next, the `curl` command is used to download an installation script. Finally, the install script is run and downloads and sets up the debugger in the `/clrdbg` directory.

■ **Caution** It is important that you create the Docker file exactly as it is shown in the listing; otherwise, the debugger won't work. If you encounter problems, then use the Docker file included in the source code download for this chapter, which is linked from the `apress.com` page for this book.

Notice that the `ENTRYPOINT` command doesn't start the .NET Core runtime or the debugger. The process for debugging in a container is to use the `docker exec` command to connect to the container once it is running, start the debugger, and then use it to start the application. This means the `ENTRYPOINT` only needs to prevent the container from stopping, which is done using the `sleep infinity` command.

Creating the Debugging Service

The next step is to update the development compose file so that it includes a description of the service that will be used for debugging, as shown in Listing 8-13.

Listing 8-13. Describing the Debugging Service in the docker-compose-dev.yml File

```
version: "3"

volumes:
  productdata:

networks:
  backend:

services:

  mysql:
    image: "mysql:8.0.0"
    volumes:
      - productdata:/var/lib/mysql
    networks:
      - backend
    environment:
      - MYSQL_ROOT_PASSWORD=mysecret
      - bind-address=0.0.0.0
```

```
  mvc:
    build:
      context: .
      dockerfile: Dockerfile.dev
    volumes:
      - .:/app
      - /app/obj
      - /app/bin
      - ~/.nuget:/root/.nuget
      - /root/.nuget/packages/.tools
    ports:
      - 3000:80
    networks:
      - backend
    environment:
      - DBHOST=mysql
      - WAITHOST=mysql
    depends_on:
      - mysql

  debug:
    build:
      context: .
      dockerfile: Dockerfile.debug
    volumes:
      - .:/app
      - /app/obj
      - /app/bin
      - ~/.nuget:/root/.nuget
      - /root/.nuget/packages/.tools
    ports:
      - 3000:80
    networks:
      - backend
    environment:
      - DBHOST=mysql
    depends_on:
      - mysql
```

The debug service is similar to the mvc service but uses the debug Docker file and omits the WAITHOST environment variable used by the wait-for-it package. Save the changes to the compose file and run the command shown in Listing 8-14 in the ExampleApp folder to create the images that will be used for the development services. The RUN commands in the debug Docker file will take a few moments to execute as the installation script and the debugger are downloaded and processed.

Listing 8-14. Building the Development Services

```
docker-compose -f docker-compose-dev.yml -p dev build
```

Starting the Debugging Service

The development and debug containers require access to the same files and use the same port mapping, which means they cannot run at the same time. Run the command shown in Listing 8-15 in the ExampleApp folder to ensure that the regular development container is not running.

Listing 8-15. Stopping the Development Container

```
docker-compose -f docker-compose-dev.yml -p dev stop mvc
```

Once you are sure that the development container has stopped, run the command shown in Listing 8-16 in the ExampleApp folder to start the debugging container.

Listing 8-16. Starting the Debugging Service

```
docker-compose -f docker-compose-dev.yml -p dev up debug
```

The container will start and write out the following message, indicating that it is ready for the debugger to be started:

```
...
debug_1  | Ready for debugging.
...
```

To test the configuration of the container, open another command prompt and run the command shown in Listing 8-17, which tells Docker to start the debugger in the container.

Listing 8-17. Testing the Debugger

```
docker exec -i dev_debug_1 /clrdbg/clrdbg --interpreter=mi
```

The debugger process will start and display the following startup message before waiting for a command:

```
...
=message,text="-----------------------------------------------------------------\n
You may only use the Microsoft .NET Core Debugger (clrdbg) with Visual Studio\nCode, Visual
Studio or Visual Studio for Mac software to help you develop and\ntest your applications.
\n-----------------------------------------------------------------------------\n",
send-to="output-window"
(gdb)
...
```

The configuration of the IDEs in the following sections allows them to execute the docker exec command to start the debugger and start issuing commands to it, beginning with executing the example ASP.NET Core MVC application. Type Control+C at the command prompt to stop the debugger, leaving the container running and ready for use in the sections that follow.

Debugging with Visual Studio 2017

Visual Studio 2017 includes integrated support for the debugger that is used in containers, but the support is rudimentary and requires rigid adherence to specific Docker files and compose file names and contents, which makes it difficult to use. The approach I use is to take advantage of the debugger support without having to give up control of the Docker configuration.

Create a file called `debug_config.xml` in the `ExampleApp` folder, ensuring the contents match those in Listing 8-18.

Listing 8-18. The Contents of the debug_config.xml File in the ExampleApp Folder

```
<?xml version="1.0" encoding="utf-8" ?>
<PipeLaunchOptions
  PipePath="docker"
  PipeArguments="exec -i dev_debug_1 /clrdbg/clrdbg --interpreter=mi"
  TargetArchitecture="x64" MIMode="clrdbg"
  ExePath="dotnet" WorkingDirectory="/app"
  ExeArguments="/app/bin/Debug/netcoreapp1.1/ExampleApp.dll">
</PipeLaunchOptions>
```

This XML file provides the configuration information that will allow Visual Studio to run the command that starts the debugger in the container and send commands to it. The configuration is expressed using the attributes of the `PipeLaunchOptions` element, which are described in Table 8-5.

Table 8-5. *The Configuration for the Debugger*

Name	Description
PipePath	This is the name of the program that will be run to provide the IDE with a connection to the debugger. For containerized development, this is Docker.
`PipeArguments`	These are the arguments that are passed to Docker to create the connection to the debugger.
`TargetArchitecture`	This is the architecture for the platform, which must be set to `x64`.
`MIMode`	This setting specifies the machine interface mode, which must be set to `clrdbg` for containerized development.
`ExePath`	This is the name of the command that will be run to start the application that will be debugged. For containerized projects, this is `dotnet`, referring to the .NET Core runtime.
`ExeArguments`	This is the argument that will be passed to `dotnet` to start the project and that should be set to the DLL that is produced by the compilation process.
`WorkingDirectory`	This specifies the working directory for running the application under the debugger.

Open the Visual Studio command window by selecting Other Windows ➤ Command Window from the View menu. The command window allows you to run commands directly in Visual Studio and is required to start the debugger.

Make sure the debugging container is running using the command in Listing 8-16, then enter the command shown in Listing 8-19 into the Visual Studio command window, and finally press Return to start the debugger.

Listing 8-19. Starting the Debugger

```
Debug.MIDebugLaunch /Executable:dotnet /OptionsFile:"C:\ExampleApp\debug_config.xml"
```

When entering the command, set the `/OptionsFile` argument to the full path to the `debug_config.xml` file. For me, this file is in the `C:\ExampleApp` folder, but you must set this value to reflect the location of the file on your system.

Testing the Debugger

The debugger will start and load the files that it needs to execute the ASP.NET Core MVC application. Once the startup process is complete, the standard Visual Studio debugger controls and features are enabled.

To test the debugger, open the `HomeController.cs` file, right-click the first statement in the `Index` action method, and select Breakpoint ➤ Insert Breakpoint from the pop-up menu. Open a browser tab and request `http://localhost:3000` to send an HTTP request to the application through the container's port mapping. When the handling of the request reaches the breakpoint, the debugger will halt execution of the application and pass control to Visual Studio, as shown in Figure 8-4. You can use the menu items in the Debug menu to control the execution of the application and to inspect the current state.

```
4
5   namespace ExampleApp.Controllers {
6       public class HomeController : Controller {
7           private IRepository repository;
8           private string message;
9
10          public HomeController(IRepository repo, IConfiguration config) {
11              repository = repo;
12              message = $"Host ({config["HOSTNAME"]})";
13          }
14
15          public IActionResult Index() {
16              ViewBag.Message = message;
17              return View(repository.Products);
18          }
19      }
20  }
21
```

Figure 8-4. *Debugging an application in a container using Visual Studio*

Select Debug ➤ Continue to resume execution of the application or Debug ➤ Stop Debugging to bring the debugging session to an end. When you have finished debugging, you can stop the container by typing `Control+C` in the command prompt you used to start it or by opening another command prompt, navigating to the `ExampleApp` folder, and running the command shown in Listing 8-20.

Listing 8-20. Stopping the Debugging Container

```
docker-compose -f docker-compose-dev.yml -p dev stop debug
```

Debugging with Visual Studio Code

Open the `ExampleApp` folder using Visual Studio Code and select Debug from the View menu to open the debugging controls. If you are prompted to add assets to build and debug the project, then click the Yes button.

Click the settings icon, as shown in Figure 8-5, and select .NET Core from the list of options.

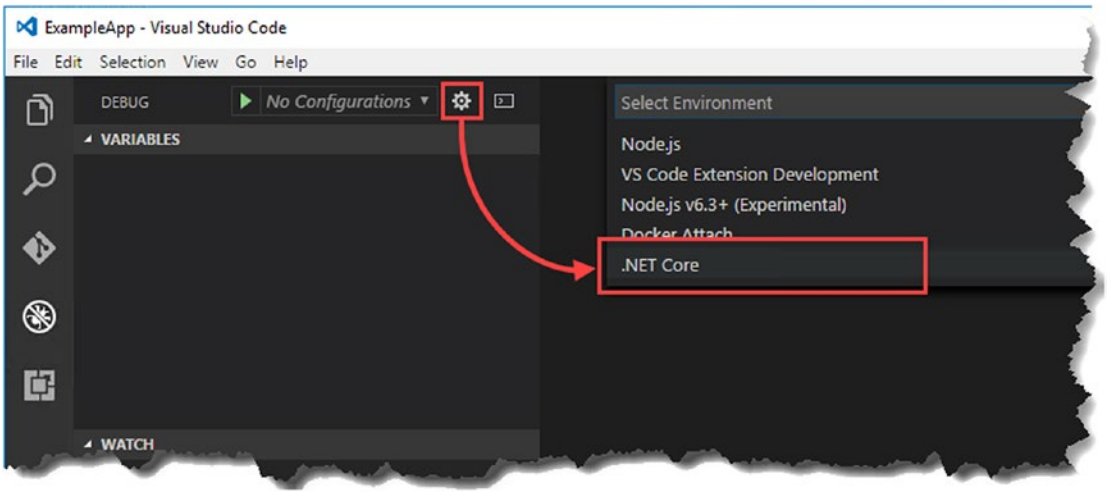

Figure 8-5. *Beginning the debugging configuration*

■ **Note** The example in this section requires the C# extension for Visual Studio Code described in Chapter 3.

Visual Studio Code will create a file called `launch.json` in a new folder called `.vscode`, and the file will be opened for editing. Edit the file to make the changes and additions shown in Listing 8-21, which configure Visual Studio Code so that it can run the debugger in the container.

Listing 8-21. Configuring Debugging in the launch.json File in the ExampleApp/.vscode Folder

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": ".NET Core Launch (web)",
            "type": "coreclr",
            "request": "launch",
            "preLaunchTask": "build",
```

```
            "program": "/app/bin/Debug/netcoreapp1.1/ExampleApp.dll",
            "args": [],
            "cwd": "/app",
            "stopAtEntry": false,
            "internalConsoleOptions": "openOnSessionStart",
            "launchBrowser": {
                "enabled": false,
                "args": "${auto-detect-url}",
                "windows": {
                    "command": "cmd.exe",
                    "args": "/C start ${auto-detect-url}"
                },
                "osx": {
                    "command": "open"
                },
                "linux": {
                    "command": "xdg-open"
                }
            },
            "env": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            },
            "sourceFileMap": {
                "/app": "${workspaceRoot}",
                "/Views": "${workspaceRoot}/Views"
            },
            "pipeTransport": {
                "pipeProgram": "/bin/bash",
                "pipeCwd": "${workspaceRoot}",
                "pipeArgs": ["-c",
                    "docker exec -i dev_debug_1 /clrdbg/clrdbg --interpreter=mi"],
                "windows": {
                    "pipeProgram":
                  "${env.windir}\\System32\\WindowsPowerShell\\v1.0\\powershell.exe",
                    "pipeCwd": "${workspaceRoot}",
                    "pipeArgs":
                     [ "docker exec -i dev_debug_1 /clrdbg/clrdbg --interpreter=mi" ]
                }
            }
        },
        {
            "name": ".NET Core Attach",
            "type": "coreclr",
            "request": "attach",
            "processId": "${command.pickProcess}"
        }
    ]
}
```

■ **Caution** It is important that you edit the `launch.json` file so that it precisely matches the listing. If you have problems running the debugger, then try using the project for this chapter that is included with the source code download, which is linked from the `apress.com` page for this book.

Table 8-6 lists the configuration settings that have been changed or added in the `launch.json` file and explains what they do.

Table 8-6. *The Debugging Configuration Settings for Visual Studio Code*

Name	Description
`program`	This setting specifies the path to the DLL that the debugger will use to start the project. The value shown is the default.
`cwd`	This setting specifies the directory that contains the project files in the container.
`launchBrowser:enabled`	This setting disables the feature that opens a new browser window automatically when debugging is started.
`pipeTransport`	This setting denotes the category for configuring the debugger so that it can work with the container.
`pipeProgram`	This setting specifies the shell that will be used to run Docker.
`pipeArgs`	This setting specifies the Docker command that will connect to the container and start the debugger.
`windows`	This section contains the configuration settings for Windows. The individual settings have the same meaning as in the main section, but the values will be used when the debugger is started on Windows machines.

Save the changes to the `launch.json` file and make sure that the debugging container is running using the command in Listing 8-16. Ensure that .NET Core Launch (web) is selected in the Visual Studio Code Debug drop-down list, and click the green start arrow at the top of the Debug window, as shown in Figure 8-6. If you are prompted to select a task runner, then choose .NET Core from the list and then click the green start arrow to start debugging.

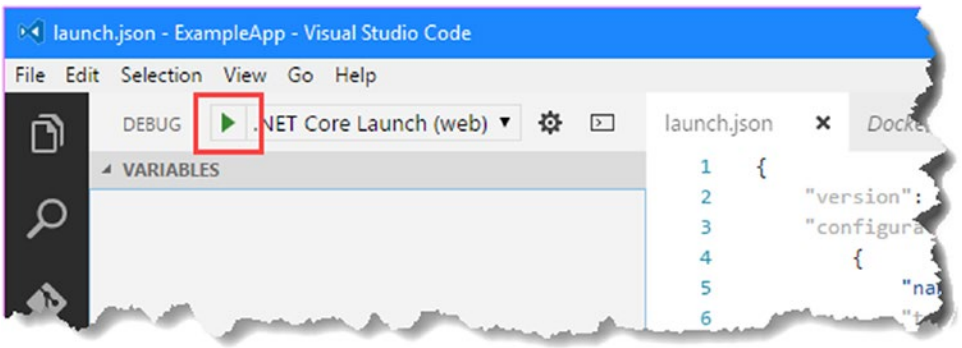

Figure 8-6. *Starting the Visual Studio Code debugger*

Testing the Debugger

The debugger will start and load the files that it needs to execute the ASP.NET Core MVC application. Once the startup process is complete, the standard Visual Studio Code debugger controls and features are enabled, just as though you were running the normal built-in debugger.

Select Explorer from the View menu to return to the list of files in the project, open the `HomeController.cs` file in the `Controllers` folder, and click at the left edge of the window next to either of the lines of code in the `Index` method to create a breakpoint, as illustrated by Figure 8-7.

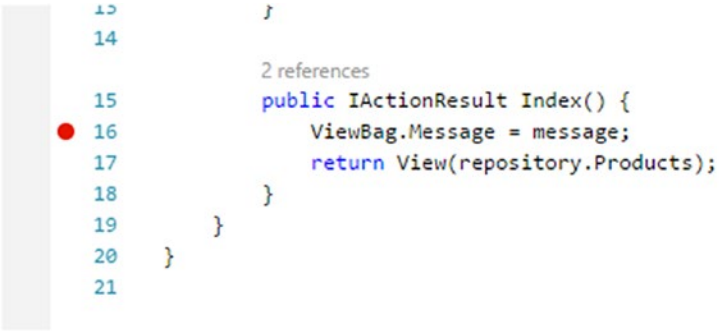

Figure 8-7. *Creating a breakpoint*

Open a new browser window and request the `http://localhost:3000` URL, which will target the MVC application running in the debugging container. When the request handling reaches the breakpoint, the debugger will halt execution of the application and pass control to Visual Studio Code. At this point, you can use the normal Visual Studio Code debugging controls to step through the application or inspect the current state.

Click the green button in the debugging control palette, as shown in Figure 8-8, to resume execution of the application.

Figure 8-8. *Resuming application execution*

When you have finished debugging, you can stop the container by typing `Control+C` in the command prompt you used to start it or by opening another command prompt, navigating to the `ExampleApp` folder, and running the command shown in Listing 8-22.

Listing 8-22. Stopping the Debugging Container

```
docker-compose -f docker-compose-dev.yml -p dev stop debug
```

Summary

In this chapter, I explained how the Docker features described in earlier chapters can be used to create a containerized development environment, which ensures that all the developers on a project are able to work consistently while still able to use their preferred IDE configuration. I showed you how to set up the development container so that it automatically compiles and runs the application when there is a code change and how to debug a containerized application.

And that is all I have to teach you about the essentials of using Docker for ASP.NET Core MVC applications. I started by creating a simple Docker image and then took you on a tour of the different Docker features including containers, volumes, software defined-networks, composition, swarms, and, finally, containerized development.

I wish you every success in your Docker/ASP.NET Core MVC projects, and I can only hope that you have enjoyed reading this book as much as I enjoyed writing it.