

CHAPTER 3



Getting Ready

In this chapter, I explain how to set up the packages required by ASP.NET Core MVC and Docker and create the example MVC project that is used in the rest of the book. There are instructions for Windows, Linux, and macOS, which are the three operating systems that are supported by both .NET Core and Docker.

Installing the Required Software Packages

The following sections go through the process of installing the packages that are required for ASP.NET Core MVC development and working with Docker. For quick reference, Table 3-1 lists the packages and explains their purpose.

Table 3-1. *The Software Packages Used in This Book*

Name	Description
.NET SDK	The .NET Core Software Development Kit includes the .NET runtime for executing .NET applications and the development tools required to prepare an application for containerization.
Node.js	Node.js is used in this book to run the tools that create the ASP.NET Core MVC project and to download and run Node Package Manager (NPM) packages.
NPM Package	The example application relies on an NPM package to manage its client-side libraries.
Git	Git is a revision control system. It is used indirectly in this book by bower, which is the NPM package that is used to manage client-side packages.
Docker	The Docker package includes the tools and runtime required to create and manage containers. The Windows and macOS versions of Docker include the Docker Compose tool, but it must be installed separately on Linux, as described in Chapter 6.
Visual Studio	Visual Studio is the Windows-only IDE that provides the full-featured development experience for .NET.
Visual Studio Code	Visual Studio Code is a lightweight IDE that can be used on Windows, macOS, and Linux. It doesn't provide the full range of features of the Windows-only Visual Studio product but is well-suited to ASP.NET Core MVC development.

Installing the .NET Core Software Development Kit

The .NET Core Software Development Kit (SDK) includes the runtime and development tools needed to start the development project and to prepare a .NET Core application for use in a container.

Installing the .NET Core SDK on Windows

To install the .NET Core SDK on Windows, download the installer from <https://go.microsoft.com/fwlink/?linkid=843448>. This URL is for the 64-bit .NET Core SDK version 1.1.1, which is the version that I use throughout this book and that you should install to ensure you get the expected results from the examples. (Microsoft also publishes a runtime-only installer, but this does not contain the tools that are required for this book.)

Run the installer, and once the install process is complete, open a new PowerShell command prompt and run the command shown in Listing 3-1 to check that .NET Core is working.

Listing 3-1. Testing .NET Core

```
dotnet --version
```

The output from this command will display the version of the .NET Core runtime that is installed. If you have installed only the version specified earlier, this will be 1.0.1. (Don't worry that the version number reported by this command doesn't correspond to the version you download; this is expected.)

USING LONG- AND SHORT-FORM COMMAND ARGUMENTS

Most of the examples in this book use the command line, both for .NET and for Docker. There are two types of arguments for commands: long form and short form. The default form, long form, uses two hyphens, like this:

```
dotnet --help
```

This is the long form of the `help` argument. Some commands also have a short-form argument, which uses a single hyphen, like this:

```
dotnet -h
```

Short-form and long-form arguments have the same effect. In this case, they print out a help message. Not all long-form arguments have a short-form equivalent, but you can switch between them freely when they are available.

Installing .NET Core SDK on Linux

The .NET Core SDK can be installed on popular Linux distributions. The easiest way to install .NET Core is to visit <https://www.microsoft.com/net/core>, select your distribution from the list, and copy and paste the commands into a command prompt to ensure you don't mistype any configuration arguments. For completeness, this section shows the installation process for Ubuntu 16.04, which I use throughout this book and which is the current Long Term Support (LTS) release at the time of writing.

To install .NET Core SDK on Ubuntu 16.04, open a command prompt and enter the commands in Listing 3-2 to configure package management so that Microsoft packages can be installed.

Listing 3-2. Preparing Package Management for .NET Core

```
sudo sh -c 'echo "deb [arch=amd64] https://apt-mo.trafficmanager.net/repos/dotnet-release/ xenial main" > /etc/apt/sources.list.d/dotnetdev.list'
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 417A0893
sudo apt-get update
```

Run the command shown in Listing 3-3 to download and install the .NET Core SDK package. It is important that you use the version number shown in the listing so that you get the expected results from the examples in this book.

Listing 3-3. Installing the .NET Core Package

```
sudo apt-get install dotnet-dev-1.0.1
```

Once the package has been downloaded and installed, run the command shown in Listing 3-4 to check that .NET Core is installed and working.

Listing 3-4. Testing the .NET Core Package

```
dotnet --version
```

The output from this command will display the version of the .NET Core runtime that is installed. If you have installed only the version specified earlier, this will be 1.0.1. (Don't worry that the version number reported by this command doesn't correspond to the version you download; this is expected.)

Installing .NET Core on macOS

Before installing .NET Core SDK, open a new command prompt and run the command in Listing 3-5 to install the HomeBrew package manager.

Listing 3-5. Installing the Package Manager

```
/usr/bin/ruby -e \
"$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Once installation is complete, run the commands shown in Listing 3-6 to install the OpenSSL library, which is a prerequisite for some .NET Core features.

Listing 3-6. Installing the OpenSSL Package

```
brew install openssl
mkdir -p /usr/local/lib
ln -s /usr/local/opt/openssl/lib/libcrypto.1.0.0.dylib /usr/local/lib/
ln -s /usr/local/opt/openssl/lib/libssl.1.0.0.dylib /usr/local/lib/
```

To install .NET Core on macOS, download the SDK installer from <https://go.microsoft.com/fwlink/?linkid=843444>. This URL is for the .NET Core SDK version 1.1.1, which is the version that I use throughout this book and that you should install to ensure you get the expected results from the examples.

Run the installer, and once the process is complete, open a Terminal window and run the command shown in Listing 3-7 at the prompt to check that .NET Core is working.

Listing 3-7. Testing .NET Core

```
dotnet --version
```

The output from this command will display the version of the .NET Core runtime that is installed. If you have installed only the version specified earlier, this will be 1.0.1. (Don't worry that the version number reported by this command doesn't correspond to the version you download; this is expected.)

Installing Node.js

The tools that I use to create the example ASP.NET Core MVC project in this book rely on Node.js (also known as Node), which is a runtime for server-side JavaScript applications and which has become a popular platform for development tools. It is important that you download the same version of Node.js that I use throughout this book. Although Node.js is relatively stable, there are still breaking API changes from time to time that may stop the examples from working.

The version I have used is the 6.9.2 release. You may prefer more recent releases for your own projects, but you should stick with the 6.9.2 release for the rest of this book. A complete set of 6.9.2 releases, with installers for Windows and macOS, is available at <https://nodejs.org/dist/v6.9.2>. Table 3-2 shows the installer files required for Windows and macOS (Linux installations are handled differently).

Table 3-2. *The Node.js Distribution for Windows and macOS*

Operating System	Node.js Distribution File
Windows 10	https://nodejs.org/dist/v6.9.2/node-v6.9.2-x64.msi
macOS	https://nodejs.org/dist/v6.9.2/node-v6.9.2.pkg

Installing Node.js on Windows

To install Node.js on Windows, download and run the installer listed in Table 3-2. During the installation process, ensure that the npm package manager and Add to PATH options are selected, as shown in Figure 3-1.

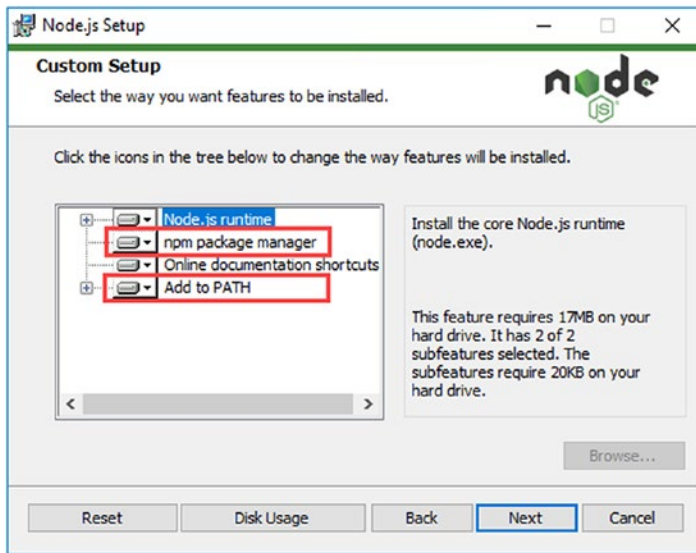


Figure 3-1. Installing Node.js on Windows

The NPM package manager is used to download and install Node packages. Adding Node.js to the PATH ensures that you can use the Node.js runtime at the command prompt just by typing `node`. Once you have completed the installation, open a new command prompt and run the command shown in Listing 3-8.

Listing 3-8. Checking That Node.js Is Installed Correctly

```
node -v
```

You should see the following version number displayed: `v6.9.2`. If the installation has been successful, then proceed to the “Installing the NPM Packages” section.

Installing Node.js on Linux

For Linux, the easiest way to install Node.js is through a package manager, using the procedures described at <https://nodejs.org/en/download/package-manager>. For Ubuntu, I ran the commands shown in Listing 3-9 to download and install Node.js.

Listing 3-9. Installing Node.js on Ubuntu

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
sudo apt-get install nodejs
```

Once you have installed Node.js, run the command shown in Listing 3-10 to check that the installation has been successful and that you have the right version.

Listing 3-10. Checking That Node.js Is Installed Correctly

```
node -v
```

You should see that version 6.x.x is installed. Version 6.9.2 is the latest at the time of writing, but there may be updates pushed into the package manager feed for version 6.x by the time you read this.

Installing Node.js on macOS

To install Node.js on macOS, download and run the installer listed in Table 3-2. Open a new Terminal and run the command shown in Listing 3-11 at the command prompt once the installer has completed.

Listing 3-11. Checking That Node.js Is Installed Correctly

```
node -v
```

You will see the following version number displayed: v6.9.2. If the installation has been successful, then proceed to the “Installing the NPM Packages” section.

Installing the NPM Package

The example application used throughout this book relies on the Bower package, which is re-installed using the Node Package Manager (NPM). NPM is included in the Node installation. Run the command shown in Listing 3-12 to install the package that will be used to manage client-side packages in the example application. For macOS and Linux, you need to run this command using `sudo` or as an administrator.

Listing 3-12. Installing the NPM Packages

```
npm install -g bower@1.8.0
```

There may be a later version of this package available by the time you read this book, but it is important that you use the version specified to ensure that you get the expected results from the examples. Table 3-3 describes the package installed by the command in Listing 3-12.

Table 3-3. The NPM Package

Name	Description
bower	Bower is a package manager that handles client-side packages, such as the Bootstrap CSS framework used by the example application in this book.

Installing Git

The Git revision control tool is required to download the client-side packages used by the example ASP.NET Core MVC application created later in this chapter. Visual Studio Code includes integrated Git support, but a separate installation is still required.

Installing Git on Windows or macOS

Download and run the installer from <https://git-scm.com/downloads>. (On macOS, you may have to change your security settings to open the installer, which has not been signed by the developers.) When the installation is complete, open a new command prompt and run the command in Listing 3-13 to check that Git is installed and working properly.

Listing 3-13. Checking the Git Install

```
git --version
```

This command prints out the version of the Git package that has been installed. At the time of writing, the latest version of Git for Windows is 2.12.0, and the latest version of Git for macOS is 2.10.1.

Installing Git on Linux

Git is already installed on most Linux distributions. If you want to install the latest version, then consult the installation instructions for your distribution at <https://git-scm.com/download/linux>. For Ubuntu, I used the following command:

```
sudo apt-get install git
```

Once you have completed the installation, open a new command prompt and run the command in Listing 3-14 to check that Git is installed and available.

Listing 3-14. Checking the Git Install

```
git --version
```

This command prints out the version of the Git package that has been installed. At the time of writing, the latest version of Git for Linux is 2.7.4.

Installing Docker

Docker supports Windows, macOS, and a range of Linux distributions. The installation process for all platforms is reasonably straightforward, as described in the following sections. Docker is available in Community and Enterprise editions, the difference being the support and certifications offered for the Enterprise edition. Both editions provide the same set of core features, and I use the free Docker Community edition throughout this book.

Installing Docker for Windows

Docker can be used on Windows, taking advantage of the integrated Hyper-V support so that Linux containers can be used.

■ **Note** At the time of writing, only 64-bit versions of Windows 10 Pro, Enterprise, and Education are supported, with the latest updates installed.

Go to <https://store.docker.com/editions/community/docker-ce-desktop-windows>, click the Get Docker CE for Windows (stable) link, and run the installer that is downloaded. Docker will start automatically when the installation is complete. You may be prompted to enable Hyper-V, as shown in Figure 3-2. Hyper-V allows Linux containers to be used on Windows and must be enabled.

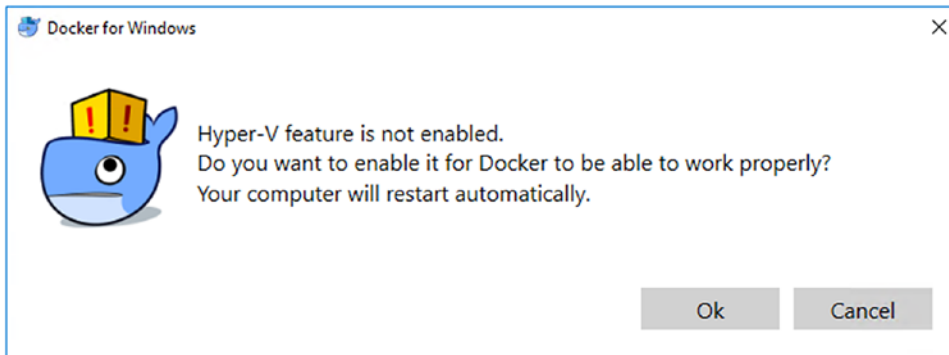


Figure 3-2. Enabling Hyper-V

Once the installation has completed, open a new PowerShell command prompt and run the command shown in Listing 3-15 to check that the installation has been successful.

Listing 3-15. Checking That Docker Is Working

```
docker run --rm hello-world
```

Docker will download the files it needs to run a simple Hello World application. Docker will write out messages like these, indicating that everything is working as expected (the command produces more output than is shown here, but this is the important part):

```
...
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world

c04b14da8d14: Pull complete
Digest: sha256:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a1feb60fd9
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```


INSTALLING DOCKER ON WINDOWS SERVER 2016

If you are creating containers for Windows-only applications, which I demonstrate in Chapter 4, you can run them on Windows Server 2016. This will be of limited appeal for most developers because Linux containers are more widely supported and easier to use, even for ASP.NET Core applications. To install Docker on Windows Server 2016, run the following commands in an Administrative PowerShell:

```
Install-Module -Name DockerMsftProvider -Force
```

```
Install-Package -Name docker -ProviderName DockerMsftProvider -Force
```

These commands install the latest version of Docker. Once the installation has completed, reboot the server and then run the following command to ensure that Docker is installed correctly and working:

```
docker run --rm hello-world:nanoserver
```

Docker will download the files it needs to run a simple Hello World application, which it will run automatically. Docker will write out messages like these, indicating that everything is working as expected (the command produces more output than is shown here, but this is the important part):

```
...
Unable to find image 'hello-world:nanoserver' locally
nanoserver: Pulling from library/hello-world

5496abde368a: Pull complete
482ab31872a2: Pull complete
4256836bc8f8: Pull complete
5bc5abeff404: Pull complete
Digest: sha256:3f5a4d0983b0cf36db8b767a25b0db6e4ae3e5abec8831dc03fe773c58ee404a
Status: Downloaded newer image for hello-world:nanoserver
```

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

Bear in mind that Windows Server 2016 cannot run Linux containers and cannot be used to follow the majority of the examples in this book (or run most of the packages that have been published via Docker).

Installing Docker for Linux

To install Docker on Linux, visit <https://www.docker.com/community-edition>, select the distribution that you are using from the list, and follow the installation instructions, copying and pasting the commands to avoid typos.

This section shows the installation process for Ubuntu 16.04, which is the distribution I have used throughout this book. Open a new command prompt and enter the commands in Listing 3-16 to configure the package manager and install the prerequisite packages that Docker relies on.

Listing 3-16. Preparing the Package Manager and Installing Prerequisite Packages

```
sudo apt-get -y install apt-transport-https ca-certificates curl
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
        $(lsb_release -cs) stable"
sudo apt-get update
```

To install Docker, run the command shown in Listing 3-17.

Listing 3-17. Installing Docker

```
sudo apt-get -y install docker-ce
```

Once Docker is installed, run the commands shown in Listing 3-18 so that you can use Docker without sudo.

Listing 3-18. Configuring Docker So That Root Access Is Not Required

```
sudo groupadd docker
sudo usermod -aG docker $USER
```

Log out of your current session and log back in again for the commands in Listing 3-18 to take effect. Once you have logged back in, run the command shown in Listing 3-19 to check that the installation has been successful.

Listing 3-19. Checking That Docker Is Working

```
docker run --rm hello-world
```

Docker will download the files it needs to run a simple Hello World application. Docker will write out messages like these, indicating that everything is working as expected (the command produces more output than is shown here, but this is the important part):

```
...
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world

c04b14da8d14: Pull complete
Digest: sha256:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a1feb60fd9
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

Installing Docker on macOS

Go to <https://store.docker.com/editions/community/docker-ce-desktop-mac>, click the Get Docker for CE Mac (stable) link, and run the installer that is downloaded. Drag the whale to the Applications folder, as shown in Figure 3-3.

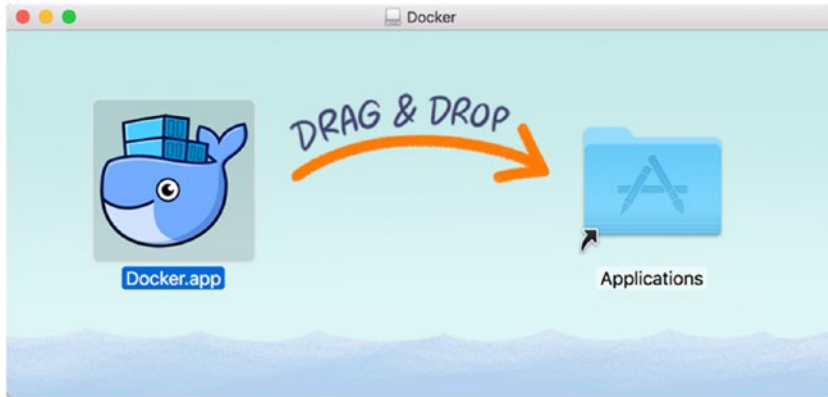


Figure 3-3. *Installing Docker*

Open Launchpad and click the Docker icon to perform the setup process. At the end of the process, open a new Terminal and run the command shown in Listing 3-20 to check that the installation has been successful.

Listing 3-20. Checking That Docker Is Working

```
docker run --rm hello-world
```

Docker will download the files it needs to run a simple Hello World application. Docker will write out messages like these, indicating that everything is working as expected (the command produces more output than is shown here, but this is the important part):

```
...
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world

c04b14da8d14: Pull complete
Digest: sha256:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdcf9431a1feb60fd9
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

Installing an IDE

Although any IDE can be used to develop ASP.NET Core projects, the most common choices are Visual Studio for Windows and Visual Studio Code for macOS and Linux (although you can also use Visual Studio Code on Windows if you wish). Make the choice that suits you best and follow the setup instructions in the sections that follow.

Installing Visual Studio 2017

Download the installer from <https://www.visualstudio.com/vs>. There are different editions of Visual Studio 2017 available, but the free Community edition is sufficient for the examples in this book. Run the installer and ensure that the .NET Core Cross-Platform Development workload is selected, as shown in Figure 3-4.

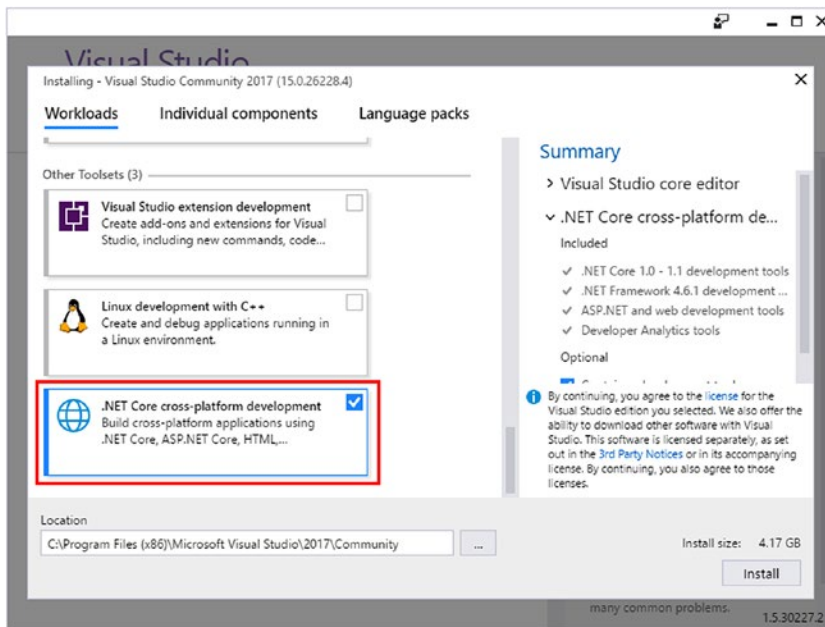


Figure 3-4. Selecting the Visual Studio packages

Click the Install button to begin the process of downloading and installing the Visual Studio features.

Adding a YAML Extension to Visual Studio

Some important Docker features are configured using files written in the YAML format. I explain what you need to know to work with these files in Chapter 6, but one aspect of the YAML format that can be frustrating is that tab characters are not allowed. Working with YAML files is made much easier by installing an extension for Visual Studio.

Start Visual Studio and select Extensions and Updates from the Tools menu. Navigate to the Online section, enter **yaml** into the search bar, and click the Download button for the Syntax Highlighting Pack extension, as shown in Figure 3-5. Close Visual Studio and the extension will be installed.

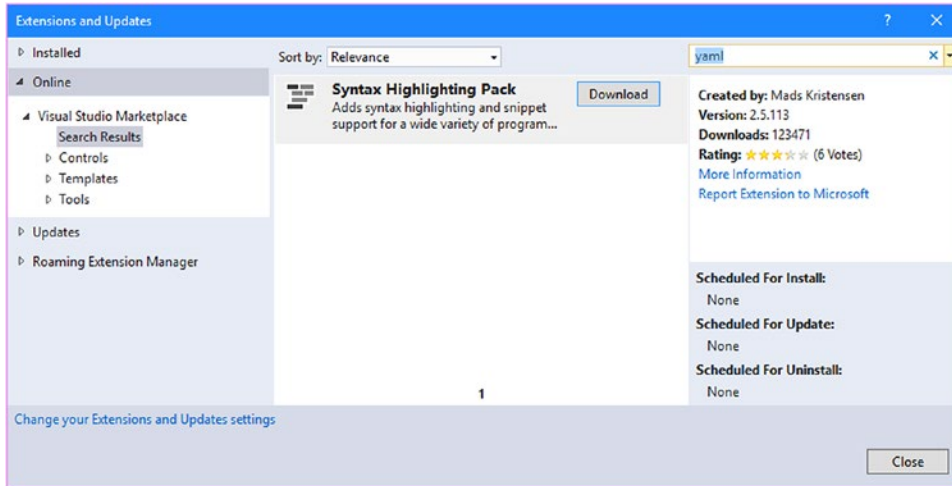


Figure 3-5. Installing the YAML editing extension

Installing Visual Studio Code

Visual Studio Code is a lightweight editor that doesn't have all the features of the full Visual Studio product but works across platforms and is perfectly capable of handling ASP.NET Core MVC projects.

Installing Visual Studio Code for Windows and Linux

To install Visual Studio code, visit <http://code.visualstudio.com> and click the download link for your platform. Run the installer and then start Visual Studio Code, which will present a standard integrated development environment.

Installing Visual Studio Code for macOS

To install Visual Studio code, visit <http://code.visualstudio.com> and click the Download for Mac link, which will download a zip file. Double-click the downloaded file to decompress it, drag the Visual Studio Code file to the Applications folder, and then use Launch Pad to start Visual Studio Code.

Adding Extensions to Visual Studio Code

There are two packages to install to follow the examples in this book using Visual Studio Code. To install these packages, click the Extensions button in the sidebar and locate and install the packages described in Table 3-4. The Docker Support package is optional, but it can help avoid typos in your Docker configuration files and makes it easier to work with YAML files, which are described in Chapter 6. Restart Visual Studio Code once the extensions are installed.

Table 3-4. Required Visual Studio Code Packages

Package Name	Description
C#	This extension provides support for editing and compiling C# files.
Docker	This extension from Microsoft provides support for working with Docker configuration files (and for running Docker commands within Visual Studio Code, although I don't use those features in this book).

Testing the Development Environment

This section contains a simple test to determine whether the development environment is capable of creating a .NET Core 1.1.1 project, which is the version of .NET Core used throughout this book.

Select a convenient location and create a folder called EnvTest. Open a new command prompt, navigate to the EnvTest folder, and run the command shown in Listing 3-21 to start a new .NET Core project.

Listing 3-21. Creating a New .NET Core Project

```
dotnet new console
```

Two files will be created in the EnvTest folder: EnvTest.csproj and Program.cs. If you are using Visual Studio, select Open ► Project/Solution from the File menu, navigate to the EnvTest folder, select the EnvTest.csproj file, and click the Open button.

If you are using Visual Studio Code, select Open Folder from the File menu (or Open if you are using macOS), select the EnvTest folder, and click the OK or Select Folder button.

Check the contents of the EnvTest.csproj file to make sure they match Listing 3-22. (If you are using Visual Studio, right-click the EnvTest project item in the Solution Explorer and select Edit EnvTest.csproj from the pop-up menu.)

Listing 3-22. The Contents of the EnvTest.csproj File in the EnvTest Folder

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

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

</Project>
```

Next, open the `Program.cs` file and make the change shown in Listing 3-23.

Listing 3-23. The Contents of the `Program.cs` File in the `EnvTest` Folder

```
using System;

namespace ConsoleApplication {
    public class Program {
        public static void Main(string[] args) {
            Console.WriteLine("Essential Docker");
        }
    }
}
```

The code in this file writes out a simple message to the console. Save the changes to the files and run the following command in the `EnvTest` folder to install the NuGet packages for the project:

```
dotnet restore
```

Next, run the following command in the `EnvTest` folder to build the code and run the test application:

```
dotnet run
```

This command will compile and run the project. If everything goes well, then you should see output similar to the following, as the code is compiled and the application is run:

```
Essential Docker
```

■ **Caution** Do not proceed until you can build and run the test application. The first thing to try is to install the SDK for .NET Core version 1.1.1 using the instructions in the “*Installing .NET Core Software Development Kit*” section of this chapter. If that doesn’t work, then try removing all other versions of .NET Core so that only version 1.1.1 is installed. If all else fails and you can’t determine the cause of the problem, you can email me at adam@adam-freeman.com and I will try to help you get back on track.

Creating the Example MVC Application

The best way to understand how Docker containers work is to get hands-on experience. This means I need a simple ASP.NET Core MVC project that can be used to demonstrate how you would use Docker containers for your own applications.

■ **Tip** You can download the example project from the source repository for this book. See the apress.com page for this book for the URL.

You may be used to relying on the built-in support provided by Visual Studio or Visual Studio Code for creating and managing ASP.NET Core projects, but I rely directly on the command-line tools in this book. As you will learn, working with Docker means understanding how projects work outside of the IDE, which requires familiarity with the .NET command line.

Creating the Project

Select a convenient location and create a folder called `ExampleApp`. Open a new command prompt, navigate to the `ExampleApp` folder, and run the command in Listing 3-24 to create a new project with basic ASP.NET Core MVC content.

Listing 3-24. Creating the ASP.NET Core Project

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

Once the project has been created, run the command shown in Listing 3-25 in the `ExampleApp` folder to install the NuGet packages it requires.

Listing 3-25. Installing NuGet Packages

```
dotnet restore
```

Opening the Project Using Visual Studio

Select **File** ► **Open** ► **Project/Solution** from the Visual Studio File menu and navigate to the `ExampleApp` folder created in the previous section. Select the `ExampleApp.csproj` file, click the **Open** button, and Visual Studio will open the project for editing.

Select **File** ► **Save All** and Visual Studio will prompt you to save a solution file, with the `.sln` extension. Save this file in the `ExampleApp` folder and you can use it to open the project in future development sessions.

Opening the Project Using Visual Studio Code

Select **Open Folder** from the File menu (or **Open** if you are using macOS), select the `ExampleApp` folder, and click the **OK** or **Select Folder** button. Visual Studio Code will open the folder and display all of the files it contains in the Explorer window. No project or solution files are required when working with Visual Studio Code.

Configuring the Client-Side Packages

Client-side packages in an ASP.NET Core MVC project are conventionally managed using the `bower` tool, which you installed earlier in this chapter. To specify the client-side packages that will be used, edit the `bower.json` file in the `ExampleApp` folder and replace the contents with those shown in Listing 3-26.

Listing 3-26. The Contents of the `bower.json` File in the `ExampleApp` Folder

```
{
  "name": "exampleapp",
  "private": true,
  "dependencies": {
    "bootstrap": "4.0.0-alpha.5"
  }
}
```


The only client-side package that I use in the example application is the Bootstrap CSS framework. To download the package, run the following command in the `ExampleApp` folder:

```
bower install
```

Bower will download the Bootstrap package and store it in the `www/lib/bootstrap` folder, along with the packages that Bootstrap depends on.

USING THE BOOTSTRAP PRE-RELEASE

Throughout this book, I use a prerelease version of the Bootstrap CSS framework. As I write this, the Bootstrap team is in the process of developing Bootstrap version 4 and has made several early releases. These releases have been labeled as “alpha,” but the quality is high, and they are stable enough for use in the examples in this book.

Given the choice of writing this book using the soon-to-be-obsolete Bootstrap 3 and a prerelease version of Bootstrap 4, I decided to use the new version even though some of the class names that are used to style HTML elements are likely to change before the final release. This means you must use the same version of Bootstrap shown in the `bower.json` file in Listing 3-30 to get the expected results from the examples.

Creating the Data Model and Repository

I am going to create a simple data model and a repository with some test data. The test data will be a placeholder until Chapter 5, when I introduce a real database that is accessed through Entity Framework Core, the ASP.NET object/relational mapping framework.

Create a folder called `ExampleApp/Models` and add to it a file called `Product.cs`, with the content shown in Listing 3-27.

Listing 3-27. The Contents of the `Product.cs` File in the `ExampleApp/Models` Folder

```
namespace ExampleApp.Models {

    public class Product {

        public Product() {}

        public Product(string name = null,
                       string category = null,
                       decimal price = 0) {
            Name = name;
            Category = category;
            Price = price;
        }
        public int ProductID { get; set; }
        public string Name { get; set; }
        public string Category { get; set; }
        public decimal Price { get; set; }
    }
}
```

This is the Product model class that I use in the SportsStore application in my *Pro ASP.NET MVC Core* book. It is easy to work with and won't take us too far away from the world of containers.

I like to follow the repository pattern in my MVC applications. I am going to create a dummy repository with static test data to get started and replace it with an implementation that accesses a real database in Chapter 5. To make the transition from static to real data as seamless as possible, I am going to expose the data model through a repository interface, whose implementation will be provided using the ASP.NET Core Dependency Injection feature at runtime.

To define the interface, create a file called `IRepository.cs` in the `ExampleApp/Models` folder and add the code shown in Listing 3-28.

Listing 3-28. The Contents of the `IRepository.cs` File in the `ExampleApp/Models` Folder

```
using System.Linq;

namespace ExampleApp.Models {

    public interface IRepository {

        IQueryable<Product> Products { get; }

    }

}
```

This interface provides access to a collection of `Product` objects through a property called `Products`. A real project would require a repository with support to create and modify objects, but read-only access will be enough functionality for the examples in this book, where the focus is on Docker.

To provide the placeholder test data, add a file called `DummyRepository.cs` to the `ExampleApp/Models` folder and add the code shown in Listing 3-29.

Listing 3-29. The Contents of the `DummyRepository.cs` File in the `ExampleApp.Models` Folder

```
using System.Linq;

namespace ExampleApp.Models {

    public class DummyRepository : IRepository {
        private static Product[] DummyData = new Product[] {
            new Product { Name = "Prod1", Category = "Cat1", Price = 100 },
            new Product { Name = "Prod2", Category = "Cat1", Price = 100 },
            new Product { Name = "Prod3", Category = "Cat2", Price = 100 },
            new Product { Name = "Prod4", Category = "Cat2", Price = 100 },
        };

        public IQueryable<Product> Products => DummyData.AsQueryable();

    }

}
```

The `DummyRepository` class implements the `IRepository` class, and its `Products` property returns a collection of `Product` objects created with static placeholder data. This isn't data you would display to a user, but it is enough to get started with until a real database is added in Chapter 5.

Preparing the Controller and View

An MVC application needs at least one controller and a view to display. Edit the `HomeController.cs` file in the `ExampleApp/Controllers` folder and replace the contents with those shown in Listing 3-30.

Listing 3-30. The Contents of 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 = config["MESSAGE"] ?? "Essential Docker";
        }

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

This controller has a constructor that declares a dependency on the `IRepository` interface, which will be resolved using the ASP.NET dependency injection feature at runtime. There is one action method, called `Index`, that will be targeted by the default MVC route and that provides its view with the `Product` objects retrieved from the repository.

The constructor declares a dependency on the `IConfiguration` interface, which will provide access to the application's configuration. This allows for a configuration setting called `MESSAGE` to be read, which is passed to the `Index` action method's view through the view bag and which I use in later chapters to differentiate the results returned by different instances of the MVC application when I show you how to scale up a containerized application.

To provide the view, edit the `Index.cshtml` file in the `ExampleApp/Views/Home` folder and replace its contents with those shown in Listing 3-31.

Listing 3-31. The Contents of the `Index.cshtml` File in the `ExampleApp/Views/Home` Folder

```
@model IEnumerable<ExampleApp.Models.Product>
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ExampleApp</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
</head>
```

```

<body>
  <div class="m-1 p-1">
    <h4 class="bg-primary text-xs-center p-1 text-white">@ViewBag.Message</h4>
    <table class="table table-sm table-striped">
      <thead>
        <tr><th>ID</th><th>Name</th><th>Category</th><th>Price</th></tr>
      </thead>
      <tbody>
        @foreach (var p in Model) {
          <tr>
            <td>@p.ProductID</td>
            <td>@p.Name</td>
            <td>@p.Category</td>
            <td>@p.Price.ToString("F2")</td>
          </tr>
        }
      </tbody>
    </table>
  </div>
</body>
</html>

```

The view displays a banner with the message received via the view bag and a table containing the details of the Product objects provided as the view model by the action method.

Configuring ASP.NET and Creating the Entry Point

To configure the application, open the `Startup.cs` file and replace the contents with the code shown in Listing 3-32, which configures the basic functionality required for an MVC application.

Listing 3-32. The Contents of the `Startup.cs` File in the `ExampleApp` Folder

```

using ExampleApp.Models;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Configuration;

namespace ExampleApp {
  public class Startup {
    private IConfiguration Configuration;

    public Startup(IHostingEnvironment env) {
      Configuration = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddEnvironmentVariables()
        .Build();
    }
  }
}

```

```

public void ConfigureServices(IServiceCollection services) {
    services.AddSingleton<IConfiguration>(Configuration);
    services.AddTransient<IRepository, DummyRepository>();
    services.AddMvc();
}

public void Configure(IApplicationBuilder app,
    IHostingEnvironment env, ILoggerFactory loggerFactory) {
    loggerFactory.AddConsole();
    app.UseDeveloperExceptionPage();
    app.UseStatusCodePages();
    app.UseStaticFiles();
    app.UseMvcWithDefaultRoute();
}
}
}

```

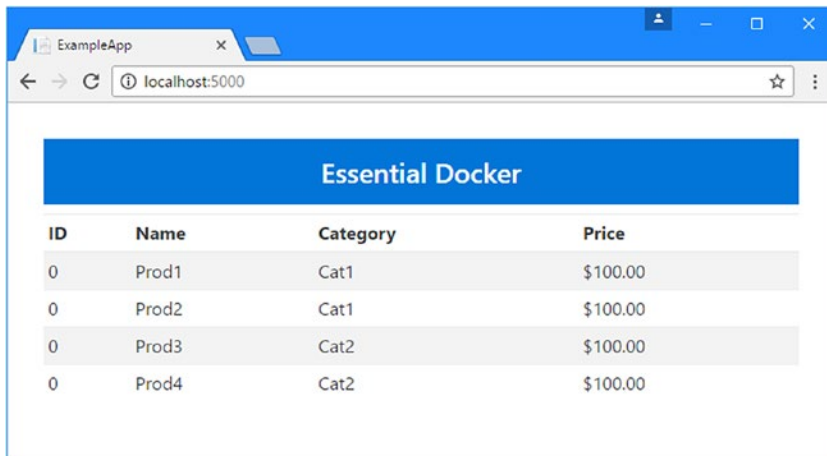
Running the MVC Application

If you are using Visual Studio, you can run the project by selecting **Start Without Debugging** from the **Debug** menu. A new browser tab or window will open and show the application.

Alternatively, run the following command in the `ExampleApp` folder to build and run the project from the command line:

```
dotnet run
```

Once the project has started, open a new browser window or tab and navigate to `http://localhost:5000`, which is the default port used by the built-in server. The browser will show the content illustrated in Figure 3-6, displaying the data from the dummy repository.



Essential Docker			
ID	Name	Category	Price
0	Prod1	Cat1	\$100.00
0	Prod2	Cat1	\$100.00
0	Prod3	Cat2	\$100.00
0	Prod4	Cat2	\$100.00

Figure 3-6. Running the example application

Once you have tested the application, type `Control+C` to exit the .NET Core runtime and return control to the command prompt.

Summary

In this chapter, I explained how to install the tools and packages that are required to work on ASP.NET Core MVC packages with Docker and created the example MVC application that will be used in the examples. In the next chapter, I describe the fundamental Docker building blocks: images and containers.