CHAPTER 4

Docker Images and Containers

The building blocks for working with Docker are *images* and *containers*. An image is a template for an application and the files required to run it. A container is created from an image and is used to execute the application in isolation so that one application doesn't interfere with another. In this chapter, I explain how to create and use images and containers and demonstrate how to use Docker to containerize an ASP.NET Core MVC application for use on Linux and Windows servers. Table 4-1 puts this chapter in context.

Table 4-1. *Putting Docker Images and Containers in Context*

Question	Answer
What are they?	Images are templates that contain the files that an application needs. Images can be built on top of one another, which makes the process of preparing an image for an application relatively simple. Containers are instances of an application created from an image. A single image can be used to create multiple containers, all of which are isolated from one another.
Why are they useful?	Images and containers are the key Docker building blocks. Images can be published to the Docker Hub so they can be used more widely, either within your organization or publicly.
How are they used?	Images are created using instructions contained in a Docker file using the `docker build` command. Containers are created from an image using `docker create` and are started and stopped using `docker start` and `docker stop`.
Are there any pitfalls or limitations?	For complex applications, the process of writing a Docker file, using it to create an image, and then testing a container generated from that image can be a time-consuming process.
Are there any alternatives?	No. Images and containers are core Docker features.

A. Freeman, *Essential Docker for ASP.NET Core MVC*, DOI 10.1007/978-1-4842-2778-7_4

Table 4-2 summarizes the chapter.

Table 4-2. *Chapter Summary*

Problem	Solution	Listing
List the images that are available on the local system	Use the `docker images` command	1
Download an image from a repository	Use the `docker pull` command	2, 3
Delete images	Use the `docker rmi` command	4, 5, 29
Create a custom image	Create a Docker file and use it with the `docker build` command	6, 8, 34
Prepare an ASP.NET Core MVC application for containerization	Use the `dotnet publish` command	7, 35
Create a container	Use the `docker create` command	9, 10
List the containers on the local system	Use the `docker ps` command	11
Start a container	Use the `docker start` command	12, 13, 20
Stop a container	Use the `docker stop` command	14, 15, 19
See the output from a container	Use the `docker logs` command	16, 17
Create and start a container in a single step	Use the `docker run` command	18, 36
Copy a file into a container	Use the `docker cp` command	21, 22
See the changes in a container's file system	Use the `docker diff` command	23
Run a command in a container	Use the `docker exec` command	24–26, 38–40
Create an image from a modified container	Use the `docker commit` command	27
Assign a tag to an image	Use the `docker tag` command	28, 32
Manage authentication with a repository	Use the `docker login` and `docker logout` commands	30, 33
Publish an image to a repository	Use the `docker push` command	31, 32
View the configuration of a container	Use the `docker inspect` command	37

Preparing for This Chapter

This chapter depends on the ExampleApp MVC project and the tools and packages from Chapter 3. If you don't want to work through the process of creating the example application, you can get the project as part of the free source code download that accompanies this book. See the `apress.com` page for this book.

Working with Images

Images are templates that are used to create containers and that contain a file system with all the files the application in the container requires. When you tested Docker in Chapter 3, the command you used instructed Docker to use an image called `hello-world`, which has been published to the public repository of images, known as the Docker Hub.

The `hello-world` image contains all the files required by an application that prints out a simple greeting, providing a self-contained way to distribute the files so they can be used to run the application. Run the command shown in Listing 4-1 to list the images installed on your system.

Listing 4-1. Listing the Available Images

```
docker images
```

The response to the command is a list of the images that are available locally, like this:

```
REPOSITORY     TAG       IMAGE ID        CREATED         SIZE
hello-world    latest    c54a2cc56cbb    4 months ago    1.848 kB
```

There is only one image shown in this output, which is the `hello-world` image. When Docker downloads images to create containers, it stores them locally to speed up future tasks.

Downloading Images

The `docker pull` command is used to download an image from the repository so that it is available locally. This isn't something you usually need to do explicitly because other Docker commands that manage images and containers will automatically pull the images they need. Run the command shown in Listing 4-2 to pull an image from the Docker Hub.

Listing 4-2. Pulling an Image from the Docker Hub

```
docker pull alpine
```

It can take a while for Docker to download the image, which contains an embedded version of Linux called Alpine. This image doesn't have any bearing on .NET development, but it is relatively small, which means that it can be downloaded quickly.

When the new image has been downloaded, run the `docker images` command and you will see that the list has been updated.

```
REPOSITORY     TAG       IMAGE ID        CREATED         SIZE
alpine         latest    baa5d63471ea    8 weeks ago     4.803 MB
hello-world    latest    c54a2cc56cbb    5 months ago    1.848 kB
```

Repository images can be tagged, allowing different versions of an image to coexist and ensuring that you get the right version of an image when you pull it. A tag is specified by appending a colon (the `:` character) to the image name, followed by the tag. Run the command in Listing 4-3 to pull a variation of the `alpine` image.

Listing 4-3. Pulling an Image Variation

```
docker pull alpine:3.4
```

This command pulls the version of the `alpine` image that has been tagged as `3.4`, indicating that the image contains version 3.4. When the image has been downloaded, the `docker images` command will show the variations in the list.

```
REPOSITORY      TAG         IMAGE ID        CREATED          SIZE
alpine          3.4         baa5d63471ea    8 weeks ago      4.803 MB
alpine          latest      baa5d63471ea    8 weeks ago      4.803 MB
hello-world     latest      c54a2cc56cbb    5 months ago     1.848 kB
```

The images have the same `IMAGE ID` because they contain identical content, meaning that the `alpine` maintainers have tagged the same image with two different tags. Docker is smart enough to know that it has the content that it requires already and won't download a duplicate image. (You may not see the same IDs if a more recent version of Alpine has been released by the time you read this chapter.)

Omitting the image tag is equivalent to requesting the variation tagged `latest`, so `docker pull alpine` and `docker pull alpine:latest` are equivalent commands. Since I didn't specify a tag when I pulled the earlier `alpine` image or the `hello-world` image used in Chapter 3, the `latest` versions were retrieved.

■ **Tip** You can see which tags are available for an image by going to the Docker Hub (`https://hub.docker.com`) and searching for an image.

Deleting Images

The `docker rmi` command is used to remove one or more images from your machine. Images are deleted by specifying their unique ID.

Run the command shown in Listing 4-4 to delete the `alpine:3.4` image, using the ID shown by `docker images`. (You might not have the same ID, which may have changed since I wrote this chapter. Check the output from the `docker images` command to see which ID you need to use for this command.)

Listing 4-4. Deleting an Image by ID

```
docker rmi -f baa5d63471ea
```

Run the `docker images` command again and you will see that both `alpine` images have been removed, since both images had the specified ID. The `-f` argument is used to remove images even when they are being used by containers (which I describe later in the chapter).

Copying and pasting individual image IDs is a tedious and error-prone process. If you want to delete all the images you have installed, then you can use the command in Listing 4-5.

Listing 4-5. Deleting All Images

```
docker rmi -f $(docker images -q)
```

The `-q` argument specifies the oddly named quiet mode, which returns only the `IMAGE ID` values from the `docker images` command, which are processed by the `docker rmi` command, removing all the images in the list.

Creating a Docker File for a Custom Image

The Docker Hub contains a wide range of images for prepackaged applications, but for ASP.NET Core development, the real power of Docker comes from being able to create custom images for MVC applications.

Custom images are described in a Docker file, conventionally named `Dockerfile`, which contains a series of instructions that Docker follows like a recipe.

To demonstrate how custom images work, I am going to create one for the example application from Chapter 3. Open the `Dockerfile` that was added to the project and replace the contents with those shown in Listing 4-6.

Listing 4-6. The Contents of the Dockerfile File in the ExampleApp Folder

```
FROM microsoft/aspnetcore:1.1.1

COPY dist /app

WORKDIR /app

EXPOSE 80/tcp

ENTRYPOINT ["dotnet", "ExampleApp.dll"]
```

These five commands are all that's required to create a Docker image for the example application. Each of the commands is described in detail in the sections that follow.

Setting the Base Image

One of the most powerful features of Docker images is that they can be based on existing images, meaning that they include all the files that the base image contains. The `FROM` command is the first command in a Docker file, and it specifies the base image that will be used.

In this case, the base image is called `microsoft/aspnetcore`, and I have specified that the version tagged `1.1.1` should be used, which contains .NET Core and ASP.NET Core version 1.1.1.

```
...
FROM microsoft/aspnetcore:1.1.1
...
```

This image is produced by Microsoft, and it contains the .NET Core runtime and the ASP.NET Core packages, compiled into native code to improve application startup. This image doesn't contain the .NET SDK, which means the MVC application must be prepared before it is used in the image, as demonstrated in the "*Preparing the Application for the Image*" section.

■ **Tip** I show you how to create a containerized development environment in Chapter 8, which uses a base image that does include the .NET SDK.

Copying the Application Files

When you containerize an ASP.NET Core application, all the compiled class files, NuGet packages, configuration files, and the Razor views are added to the image. The COPY command copies files or folders into the container.

```
...
COPY dist /app
...
```

This command copies the files from a folder called dist into a folder called /app in the container. The dist folder doesn't exist at the moment, but I'll create it when I prepare the MVC project for use with the container.

Setting the Working Directory

The WORKDIR command sets the working directory for the container, which is useful if you need to run commands or use files without having to specify the full path each time. The command in the Docker file sets the path to the /app folder that the COPY command created and that contains the application files.

Exposing the HTTP Port

Processes inside a container can open network ports without any special measures, but Docker won't allow the outside world to access them unless the Docker file contains an EXPOSE command that specifies the port number, like this:

```
...
EXPOSE 80/tcp
...
```

This command tells Docker that it can make port 80 available for TCP traffic from outside the container. For the example application, this is required so that the ASP.NET Core Kestrel server can receive HTTP requests.

■ **Tip** Working with ports in containers is a two-step process. See the "*Working with Containers*" section for details of how to complete the configuration so that the server can receive requests.

Running the Application

The final step in the Docker file is the ENTRYPOINT command, which tells Docker what to do when the container starts.

```
...
ENTRYPOINT ["dotnet", "ExampleApp.dll"]
...
```

This command tells Docker to run the `dotnet` command-line tool to execute the `ExampleApp.dll` file, which I will create in the next section. The path to the `ExampleApp.dll` file doesn't have to be specified because it is assumed to be within the directory specified by the `WORKDIR` command, which will contain all the application's files.

Preparing the Application for the Image

There are some entries in the Docker file that may not make immediate sense, especially if you are used to working with ASP.NET Core MVC projects through Visual Studio or Visual Studio Code.

The base image specified by the Docker file in Listing 4-6 doesn't include the .NET Core SDK, which means that the compiler isn't available and the MVC project files cannot be compiled automatically when the application is started, which is what usually happens in development.

Instead, the application has to be compiled before it is incorporated into the image. Run the commands in Listing 4-7 from the `ExampleApp` folder to prepare the example application.

Listing 4-7. Preparing the Example Application in the ExampleApp Folder

```
dotnet restore
dotnet publish --framework netcoreapp1.1 --configuration Release --output dist
```

The `dotnet restore` command is a precautionary step to make sure that the project has all the NuGet packages it needs.

The important command is `dotnet publish`, which compiles the application and then transforms it into a stand-alone set of files that includes everything the application requires. The `--output` argument specifies that the compiled project should be written to a folder called `dist`, which corresponds to the `COPY` command from the Docker file. The `--framework` argument specifies that .NET Core version 1.1.1 should be used, and the `--configuration` argument specifies that the `Release` mode should be used.

Look at the contents of the `dist` folder when the `dotnet publish` command has completed and you will see that one of the files is called `ExampleApp.dll`. This file contains the custom code from the example project and provides the entry point for running the application, corresponding to the `ENTRYPOINT` command in the Docker file.

Creating a Custom Image

To process the Docker file and generate the image for the example application, run the command shown in Listing 4-8 in the `ExampleApp` folder.

Listing 4-8. Creating a Custom Image in the ExampleApp Folder

```
docker build . -t apress/exampleapp -f Dockerfile
```

The `docker build` command creates a new image. The period that follows the `build` keyword provides the *context*, which is the location that is used for commands such as `COPY` in the Docker file. The `-t` argument tags the new image as `apress/exampleapp`, and the `-f` argument specifies the Docker file that contains the instructions for creating the image. (The convention for naming images is to use your name or your organization's name, followed by the application name.)

Docker will download the base images it needs and then follow the instructions in the Docker file to generate the new image. When the build process has completed, you can see the new image by running the `docker images` command, which will produce output like this:

```
REPOSITORY              TAG       IMAGE ID        CREATED          SIZE
apress/exampleapp       latest    e2e0945a741d    4 seconds ago    280 MB
microsoft/aspnetcore    1.1.1     da08e329253c    23 hours ago     268 MB
```

The `apress/exampleapp` image is the custom image containing the MVC application. The `microsoft/aspnetcore` image is shown because Docker had to pull that image from the Docker Hub when it followed the `FROM` command in the Docker file.

Working with Containers

Containers bring images to life. Each container is an instance of an application created from an image, and a host system can run multiple containers, each of which is isolated from the others. In the sections that follow, I explain how to create, use, and manage containers.

Creating Containers

Containers can be created from any image, including custom images you have created.

Run the command shown in Listing 4-9 to create a new container using the custom image from the previous section as the template.

Listing 4-9. Creating a Container

```
docker create -p 3000:80 --name exampleApp3000 apress/exampleapp
```

The `docker create` command is used to create a new image.

The `-p` argument to the `docker create` command tells Docker how to map port 80 inside the container to the host operating system. In this case, I have specified that port 80 inside the container should be mapped to port 3000 in the host operating system. This corresponds to the `EXPOSE` command in the Docker file in Listing 4-6.

The `--name` argument assigns a name to the container, which makes it easier to work with once it has been created. The name in this case is `exampleApp3000`, indicating that this container will respond to requests sent to port 3000 in the host operating system.

The final argument tells Docker which image to use as the template for the new container. This command specifies the `apress/exampleapp` image, which is the name used with the `docker build` command in Listing 4-8.

Creating Additional Containers from an Image

You can create more than one container from an image, but you must ensure that there are no conflicts for configuration options such as names and port mappings. Run the command shown in Listing 4-10 to create a second container using the custom image with a different name and port mapping.

Listing 4-10. Creating Another Container

```
docker create -p 4000:80 --name exampleApp4000 apress/exampleapp
```

This command creates a container called `exampleApp4000` using the `apress/exampleapp` image but maps port 80 to port 4000 in the host. This container will be able to coexist with the `exampleApp3000` container because they use different network ports and names, even though they contain the same application.

Listing Containers

The `docker ps` command is used to list the containers that exist on a system. By default, the `docker ps` command omits containers that are not running, so the `-a` argument must be used if you want to see all the containers that are available, as shown in Listing 4-11.

Listing 4-11. Listing All Containers

```
docker ps -a
```

This command produces the following output, showing the two containers created in the previous section (I have shown only the most important columns to fit the output on the page):

```
CONTAINER ID    IMAGE                STATUS     PORTS      NAMES
765b418bc16f    apress/exampleapp    Created               exampleApp4000
136b2a3e2246    apress/exampleapp    Created               exampleApp3000
```

Each container is assigned a unique ID, which is shown in the `CONTAINER ID` column and can be used to refer to the container in Docker commands. A more natural way to refer to containers is using their name, which is shown in the `NAMES` column. The `IMAGE` column shows the image used to create the container.

The `STATUS` column shows `Created` for both containers, indicating that the containers have been successfully created and are ready to be started. The `PORTS` column is empty because neither of the containers has any active network ports at the moment, but that will change when the containers are active.

Starting Containers

The previous section used the `docker create` command to create two containers from the same image. These containers are identical on the inside and contain identical files. Only the configuration outside the containers is different, allowing the containers to coexist by using different names and mapping different network ports to port 80 inside the container.

At the moment, however, the containers are not doing anything. The applications they contain are not running, and the network ports they have been configured to use are not active.

The `docker start` command is used to start one or more containers, which are referred to by their unique ID or by their name. Run the command shown in Listing 4-12 to start the container that is called `exampleApp3000`.

Listing 4-12. Starting a Container

```
docker start exampleApp3000
```

Docker will use the ENTRYPOINT command from the Docker file to start the application in the container. In this case, that means the .NET Core runtime is started, followed by the ASP.NET Core Kestrel server, which will listen for incoming HTTP requests on port 80 inside the container.

As the container is started, Docker will also set up the port mapping so that network traffic received on port 3000 on the host operating system will be directed to port 80 inside the container, allowing Kestrel to receive HTTP requests from outside the container.

To test the container, open a new browser window and request the URL `http://localhost:3000`, which will send an HTTP request to port 3000 on the host operating system. Docker will direct the request to port 80 inside the container, which allows it to be received by Kestrel, which will start ASP.NET Core MVC and run the example application.

After a couple of seconds, you will see the response from the example MVC application, as shown in Figure 4-1. Reload the browser window to send another request to the same URL and you will see that it is much faster now that the application is up and running.

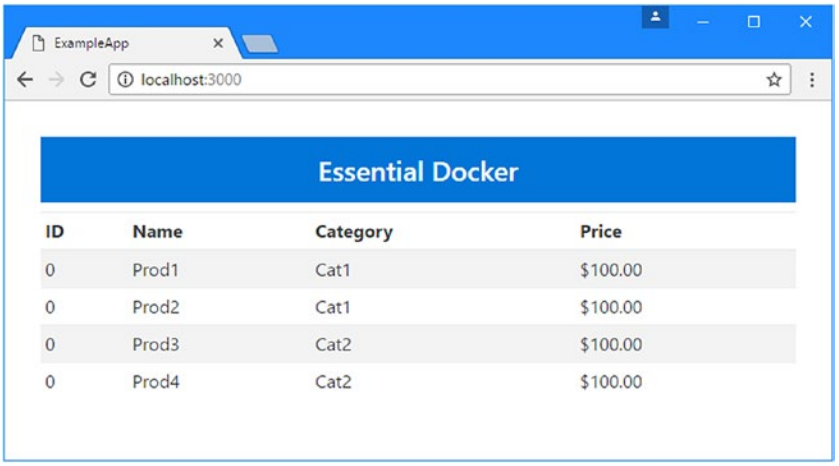

Figure 4-1. *Running the example application*

Run the command shown in Listing 4-13 to start all the containers on a system.

Listing 4-13. Starting All Containers

```
docker start $(docker ps -aq)
```

The command combines `docker start` with the output of the `docker ps` command. The `-a` argument includes containers that are not running, and the `-q` argument returns just the container IDs.

Since one of the containers is already running, the command has the effect of starting the container that is mapped to port 4000, which you can test by requesting the URL `http://localhost:4000` in the browser window, which will show the same content as Figure 4-1 because both containers are running the same application.

You can see the container's change in status by running the `docker ps -a` command, which will produce output like this:

```
CONTAINER ID  IMAGE              STATUS       PORTS                  NAMES
765b418bc16f  apress/exampleapp  Up 4 seconds 0.0.0.0:4000->80/tcp   exampleApp4000
136b2a3e2246  apress/exampleapp  Up 9 minutes 0.0.0.0:3000->80/tcp   exampleApp3000
```

The STATUS column reports that both containers are Up and how long they have been running for. The PORTS column shows the ports that each container has mapped from the host operating system. In this case, you can see that one container maps port 3000 to port 80 and the other maps port 4000 also to port 80.

These containers can coexist because the applications within the containers are isolated from each other and have no knowledge of the port mapping system. The Kestrel server that is handling HTTP requests inside the container starts listening to port 80, unware that it is running in a container and unware that the requests are coming through a port mapping on the host operating system. The ability to create multiple containers from the same image and run them side by side by varying their configuration is a key feature of Docker and is illustrated by Figure 4-2. I return to this topic in Chapter 6, when I demonstrate how to scale up an application, and in Chapter 7, when I show you how to deploy and application into a server cluster.

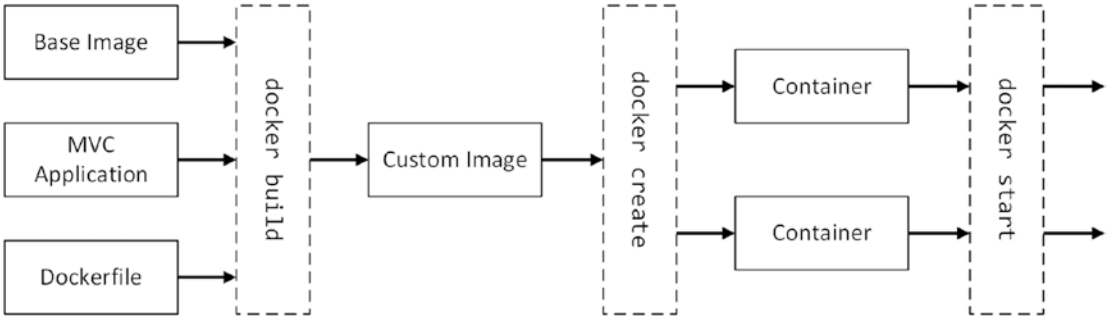

Figure 4-2. *The path from image to container to service*

Stopping Containers

Containers are stopped using the `docker stop` command, which can stop one or more containers by name or by ID. Run the command in Listing 4-14 to stop the container that is handling requests on port 3000.

Listing 4-14. Stopping a Container Using Its Name

```
docker stop exampleApp3000
```

Run the command shown in Listing 4-15 to stop all the running containers, using the list of containers generated by the `docker ps` command.

Listing 4-15. Stopping All Containers

```
docker stop $(docker ps -q)
```

The only argument required for the `docker ps` command is `-q`. The `-a` argument is not used because only the IDs of running containers are needed for the `stop` command, and this is what the `ps` command returns by default.

■ **Tip** There is also a `docker kill` command, which sends a SIGKILL signal to the container. I tend not to use this command since `docker stop` will automatically send this signal if the container hasn't stopped after ten seconds, a period that can be changed using the `-t` argument.

Getting Container Output

By default, Docker doesn't display the output from the application when you start a container using the `docker start` command. But it does keep a record that can be inspected using the `docker logs` command, as shown in Listing 4-16.

Listing 4-16. Getting Container Logs

```
docker logs exampleApp3000
```

The ASP.NET Core runtime writes out a message each time it receives an HTTP request, and the `docker logs` command displays those messages, which look like this:

```
...
Hosting environment: Production
Content root path: /app
Now listening on: http://+:80
Application started. Press Ctrl+C to shut down.
info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]
      Request starting HTTP/1.1 GET http://localhost:3000/
info: Microsoft.AspNetCore.Mvc.Internal.ControllerActionInvoker[1]
      Executing action method ExampleApp.Controllers.HomeController.Index
        (ExampleApp) with arguments ((null)) - ModelState is Valid
...
```

The `docker logs` command shows you the most recent output from the container, even after the container has been stopped. For running containers, you can use the `-f` argument to monitor the output so that you will see any new messages that are produced. Run the commands in Listing 4-17 to start a container and monitor its output.

Listing 4-17. Following a Container's Logs

```
docker start exampleApp3000
docker logs -f exampleApp3000
```

Request `http://localhost:3000` in the browser to generate some output messages. When you are done, type `Control+C` to stop the displaying the output. The container is unaffected by the `docker logs` command and continues running in the background.

Creating and Starting Containers with a Single Command

The `docker run` command is used to create a container from an image and start it in a single step, combining the effects of the `docker create` and `docker start` commands. Run the command in Listing 4-18 to create and start a container from the custom image, with a port mapping that forwards network traffic from port 5000 in the host operating system to port 80 inside the container.

Listing 4-18. Creating and Running a Container with a Single Command

```
docker run -p 5000:80 --name exampleApp5000 apress/exampleapp
```

This command takes the same arguments as the `docker create` command from Listing 4-10: it tells Docker to create the container from the `apress/exampleapp` image, sets up the port mapping, and assigns the container the name `exampleApp5000`.

The difference is that the container is started once it has been created. The `docker run` command keeps the command prompt attached to the container output so that the messages generated by the Kestrel server are displayed in the command prompt.

To test the new container, open a browser tab and request the URL `http://localhost:5000`. The HTTP request that it sends to port 5000 will be received by Docker and forwarded to port 80 inside the container, producing the same response from the MVC application you saw in earlier examples.

If you are using Linux or macOS, you can stop the container by typing `Control+C`. If you are using Windows, `Control+C` detaches the command prompt from the container but leaves it running in the background, and you will have to run the command in Listing 4-19 to stop the container.

Listing 4-19. Stopping a Container

```
docker stop exampleApp5000
```

REMOVING CONTAINERS AUTOMATICALLY

The `docker run` command can be used with the `--rm` argument, which tells Docker to remove the container when it stops. Run this command to create a container that maps port 6500 in the host container to port 80 in the new container:

```
docker run -p 6500:80 --rm --name exampleApp6500 apress/exampleapp
```

You can test the container by requesting `http://localhost:6500` in the browser and by running `docker ps`. Once you have checked that the container is working, stop the container using `Control+C` (for Linux or macOS) or using this command (Windows).

```
docker stop exampleApp6500
```

Docker will remove the container as soon as it stops, which you can confirm by running `docker ps -a` to see all the containers that exist on the system.

Modifying Containers

Images are immutable, but containers are not. Each container has its own writable file system. If you create two containers from the same image, they will be identical at first and contain the same files. But, as the applications in the containers run, the data and log files they create can cause the containers to become different, with content that reflects the user requests they process.

You can also modify a container deliberately, using the Docker tools, and then use those changes to create a new image that, in turn, can be used to create containers. This can be useful if you need to perform some kind of manual configuration to allow an application to work properly in a container. In the sections that follow, I show you different ways to make changes and then use those changes to generate a new image.

Changing a Container

To understand how containers can be changed, run the command in Listing 4-20 to ensure that the MVC containers with mappings to ports 3000 and 4000 are running.

Listing 4-20. Starting the MVC Application Containers

```
docker start exampleApp3000 exampleApp4000
```

These containers were created from the same image and contain an identical Razor view, which is used to generate a response for the MVC application's default URL. Confirm that the applications in both containers generate the same response by opening browser tabs and requesting the URLs `http://localhost:3000` and `http://localhost:4000`, as shown in Figure 4-3.

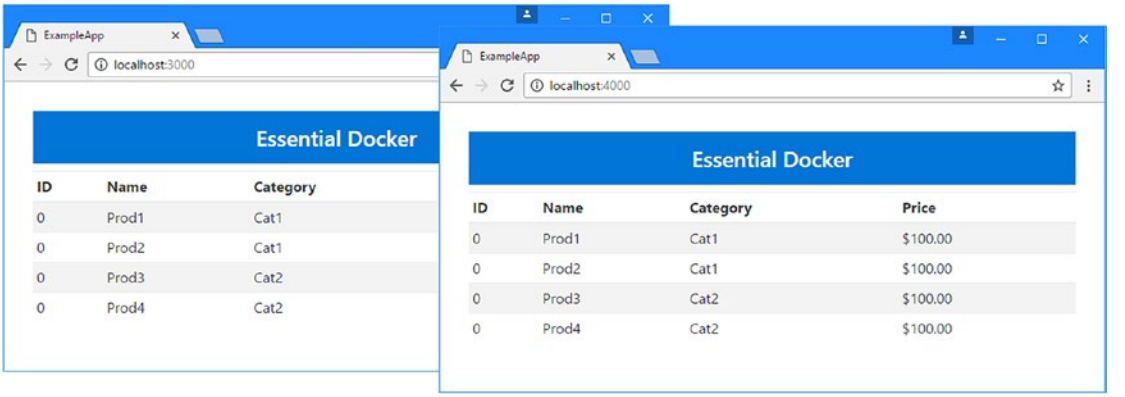

Figure 4-3. *Responses from the containerized MVC applications*

Each container has its own writeable file system, which can be modified independently of the other containers created from the same image. To create a change that has a visible effect, use your IDE to change the message displayed in the banner in the Razor view in the `ExampleApp/Views/Home` folder, as shown in Listing 4-21.

Listing 4-21. Modifying the View in 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-success text-xs-center p-1 text-white">This is new content</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 changes apply a different Bootstrap background class and change the content in the `h4` element. At the moment, the changed Razor view exists in the `ExampleApp` folder of the host operating system and has no effect on the containers.

Run the command shown in Listing 4-22 from the `ExampleApp` folder to copy the view into one of the containers.

Listing 4-22. Modifying a Container

```
docker cp ./Views/Home/Index.cshtml exampleApp3000:/app/Views/Home/
```

The `docker cp` command is used to copy files in and out of containers. This command copies the modified `Index.cshtml` file from the project folder in the host operating system into the `/app/Views` folder in the `exampleApp3000` folder, which is the folder from which the MVC application inside the container gets its views.

To see the effect of the change, use your browser to request the URL `http://localhost:3000`. The MVC application in the container that receives the HTTP request will detect the changed file, compile it into a C# class, and use it to generate a response, as shown in Figure 4-4.

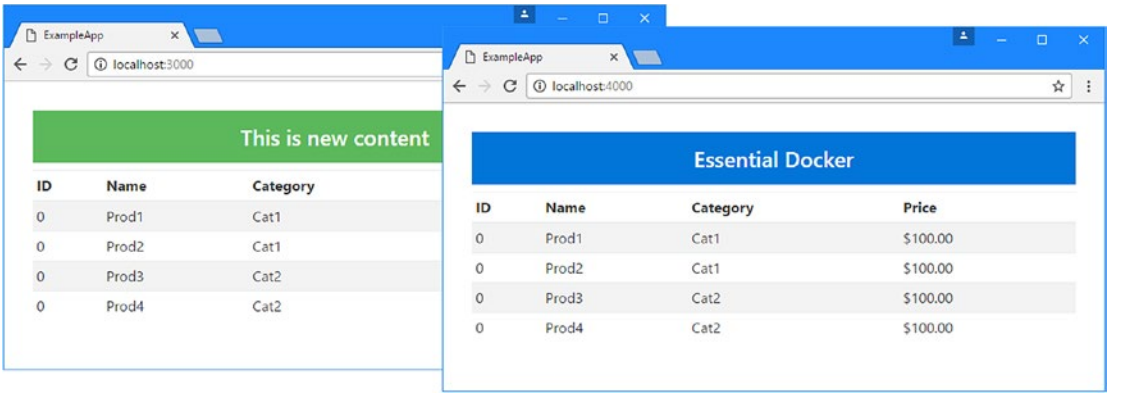

Figure 4-4. *Modifying a file in a container*

The figure also shows the response you will see when you request the URL `http://localhost:4000`. This illustrates that each container has its own storage and that changes to one container do not affect the other. The changes to a container's file system are persistent, which means you can stop and start the modified container and the change made by the command in Listing 4-22 will still be used.

■ **Caution** Modifying files in containers should be done with caution and never done to a container in production. See Chapter 8 if you want to change the files in an application because you are actively developing the project, where I explain how to create a containerized development environment.

Examining Changes to a Container

Run the command shown in Listing 4-23 to see the changes to the container's file system.

Listing 4-23. Examining Container Changes

```
docker diff exampleApp3000
```

The `docker diff` command shows the differences between the files in the container and the image that was used to create it and produces output like this (you may see slightly different results):

```
C /app
C /app/Views
C /app/Views/Home
C /app/Views/Home/Index.cshtml
C /root
A /root/.aspnet
A /root/.aspnet/DataProtection-Keys
```

```
C /tmp
A /tmp/clr-debug-pipe-1-1154405-in
A /tmp/clr-debug-pipe-1-1154405-out
```

Each entry in the results is annotated with a letter that indicates the type of change, as described in Table 4-3.

Table 4-3. *The Change Annotations from the docker diff Command*

Annotation	Description
A	This annotation indicates that a file or folder has been added to the container
C	This annotation indicates that a file or folder has been modified. For folders, a change indicates that a file has been added or removed inside that folder.
D	This annotation indicates that a file or folder has been removed from the container.

In addition to the change I made to the `Index.cshtml` file, you can see that some files have been created by the ASP.NET Core runtime. Some of files are related to debugging, which I show you how to perform in a container in Chapter 8.

Executing Commands in Containers

An alternative to copying files in and out of containers is to interact with the container directly, executing commands inside the container. This is a technique that should be used with caution, but it can be useful to run configuration tasks or to diagnose problems with a container once it is running.

Run the command in Listing 4-24 to list the content of the Razor view file in one of the containers.

Listing 4-24. Executing a Command

```
docker exec exampleApp3000 cat /app/Views/Home/Index.cshtml
```

The `docker exec` command is used to execute commands inside the container. The name of the container is followed by the command and any arguments that it requires. You can use only the commands that are available within the container. The command in Listing 4-24 tells the container to use the Linux `cat` command to list the contents of the Razor view file, which will produce the following response:

```
@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-success text-xs-center p-1 text-white">This is new content</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>
```

■ **Tip** Use the `docker start` command to make sure that a container is running before you use `docker exec`.

An extension of the ability to execute commands is to run an interactive shell, which can be more convenient if the work you need to do on a container involves chaining together several steps or would be easier with features such as file completion for navigating around the file system.

The base Linux image that is used for ASP.NET Core containers includes the Bash shell, which you can start by running the command shown in Listing 4-25.

Listing 4-25. Starting a Shell in a Container

```
docker exec -it exampleApp3000 /bin/bash
```

The `-it` argument to the `docker exec` command tells Docker that this is an interactive command that requires terminal support. Once the interactive shell has started, run the commands in Listing 4-26 to modify the contents of the Razor view in the MVC application.

Listing 4-26. Modifying the Razor View in the Bash Shell

```
cd /app/Views/Home
sed -i "s/new content/interactive content/g" Index.cshtml
exit
```

The first command changes the working directory to the folder that contains the view. The second command uses `sed` to replace the phrase `new content` with `interactive content` inside the view file. The final command, `exit`, quits the shell and leaves the container to run in the background.

You can see the effect of the change by using the browser to navigate to `http://localhost:3000`, which will produce the result shown in Figure 4-5.

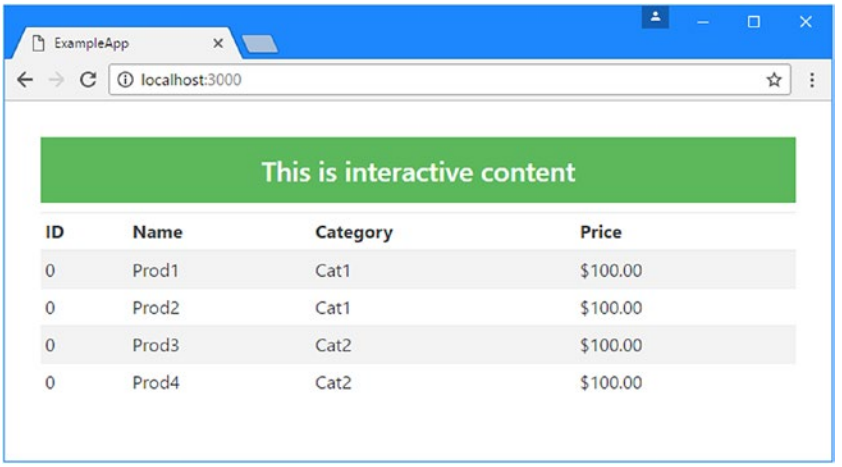

Figure 4-5. *Using the interactive shell in a Linux container*

INSTALLING AN EDITOR IN A LINUX CONTAINER

The most common task that requires interacting with a container is to use an editor to modify configuration files that are not created until the application has been started at least once. The base image for Linux containers for ASP.NET Core applications doesn't contain an editor, but you can add one using the following commands:

```
apt-get update
apt-get install vim
```

These commands download and install the venerable `vi` editor. If you have not used `vi`, then you can read about its commands at `www.vim.org`. That said, if you find yourself using a text editor, then ask yourself if you can solve the problem by changing the Docker file that creates the image from which the container has been created.

Creating Images from Modified Containers

Once you have modified a container, you can use the `docker commit` command to create a new image that incorporates the changes. Run the command in Listing 4-27 to create a new image that contains the changes made to the Razor view in Listing 4-26.

Listing 4-27. Committing Container Changes

```
docker commit exampleApp3000 apress/exampleapp:changed
```

This command creates a new variation of the apress/exampleapp image tagged as change. If you run the docker images command, you will see the new image has been added to the list.

```
REPOSITORY              TAG         IMAGE ID
apress/exampleapp       changed     00b418fa6548
apress/exampleapp       latest      827c2d48beca
microsoft/aspnetcore    1.1.1       da08e329253c
```

This image can be used to create new containers that contain the modified Razor view (and any other changes that were made to the original container).

Publishing Images

Once you have created and tested the images for an application, you can publish them so that they can be pulled to servers and used to create containers. Docker runs a public repository for images called the Docker Hub, which has been the source of the base images for the examples in this chapter. At the time of writing, you can create a free account that allows you to publish unlimited public repositories and one private repository. There are also paid-for accounts that allow more private repositories to be created. (There is also the Docker Store for paid-for software, but that isn't open to the public for publishing.)

■ **Note** For this section, you will need to visit http://hub.docker.com and create an account (the free account is sufficient).

Tagging the Images for Publication

The tags I have been using for the example images in this chapter start with apress, such as apress/exampleApp3000. When you publish an image to the Docker Hub, the first part of the tag must be the user name you used to create your account.

■ **Note** See https://docs.docker.com/registry/deploying for instructions for creating a private repository, which you can use to locally distribute images within your organization. This can be useful if you have images that contain sensitive information or if you have to conform to a policy that prohibits using third-party services.

For this section, I created a Docker Hub account with the user name of adamfreeman so that I can re-create the process that you will need to follow to set up the tags required for your own account. To publish your own images, replace adamfreeman in the commands that follow with your user name. Run the commands shown in Listing 4-28 to apply new tags to custom images so they have the right account information.

Listing 4-28. Adding Image Tags in Preparation for Publishing

```
docker tag apress/exampleapp:changed adamfreeman/exampleapp:changed
docker tag apress/exampleapp:latest adamfreeman/exampleapp:unchanged
```

The new tags create two variations of the image, changed and unchanged. These tags don't replace the original ones, which you can see using the `docker images` command, which will produce a result like this one:

```
REPOSITORY                TAG          IMAGE ID            CREATED
apress/exampleapp         changed      b1af7e78f418        10 minutes ago
adamfreeman/exampleapp    changed      b1af7e78f418        10 minutes ago
apress/exampleapp         latest       452007c3b3dd        31 minutes ago
adamfreeman/exampleapp    unchanged    452007c3b3dd        31 minutes ago
microsoft/aspnetcore      1.1.1        da08e329253c        5 days ago
```

I like to remove the original tags to keep the list of images clean. Run the command shown in Listing 4-29 to remove the original tags, leaving just the new tags in place.

Listing 4-29. Removing Old Image Tags

```
docker rmi apress/exampleapp:changed apress/exampleapp:latest
```

Run the `docker images` command again and you will see that only the images with my account name are left.

```
REPOSITORY                TAG          IMAGE ID            CREATED
adamfreeman/exampleapp    changed      b1af7e78f418        14 minutes ago
adamfreeman/exampleapp    unchanged    452007c3b3dd        35 minutes ago
microsoft/aspnetcore      1.1.1        da08e329253c        5 days ago
```

Authenticating with the Hub

The Docker Hub requires authentication before you can publish images to your account. Run the command shown in Listing 4-30 to authenticate yourself with the Docker Hub using the user name and password you used to create the account.

Listing 4-30. Authenticating with the Docker Hub

```
docker login -u <yourUsername> -p <yourPassword>
```

The `docker login` command uses the user name and password provided by the `-u` and `-p` arguments to perform authentication with the Hub. Once you have used the `docker login` command, subsequent commands sent to the Hub will include your authentication credentials.

Publishing the Images

Run the commands shown in Listing 4-31 to push your images to the Docker Hub.

Listing 4-31. Publishing Images to the Docker Hub

```
docker push adamfreeman/exampleapp:changed
docker push adamfreeman/exampleapp:unchanged
```

The first image can take a while to upload as the files it contains are transferred to the repository. The second image should be much quicker, since only the changes between the containers are needed.

If you omit the variation in the tag, then the images will be published as `:latest`. The Docker Hub does not enforce any kind of version control and doesn't automatically assign the most recent push request as the `:latest` image. For this reason, it is sensible to make an explicit push request that sets the image you want used by default. Run the commands shown in Listing 4-32 to tag and push a new image.

Listing 4-32. Tagging and Pushing the Default Variation for an Image

```
docker tag adamfreeman/exampleapp:unchanged adamfreeman/exampleapp:latest
docker push adamfreeman/exampleapp:latest
```

Finally, prevent anyone else using your account from publishing images by running the command shown in Listing 4-33 to log out of the Docker Hub. Further push requests will not work until the `docker login` command is used again.

Listing 4-33. Logging Out of the Docker Hub

```
docker logout
```

The images are now available in the Docker Hub. Log into `hub.docker.com`, locate your repository, and click the `Tags` section to see the images that have been published, as shown in Figure 4-6. You can use the Docker Hub web site to control access to your images and to provide additional information about them.

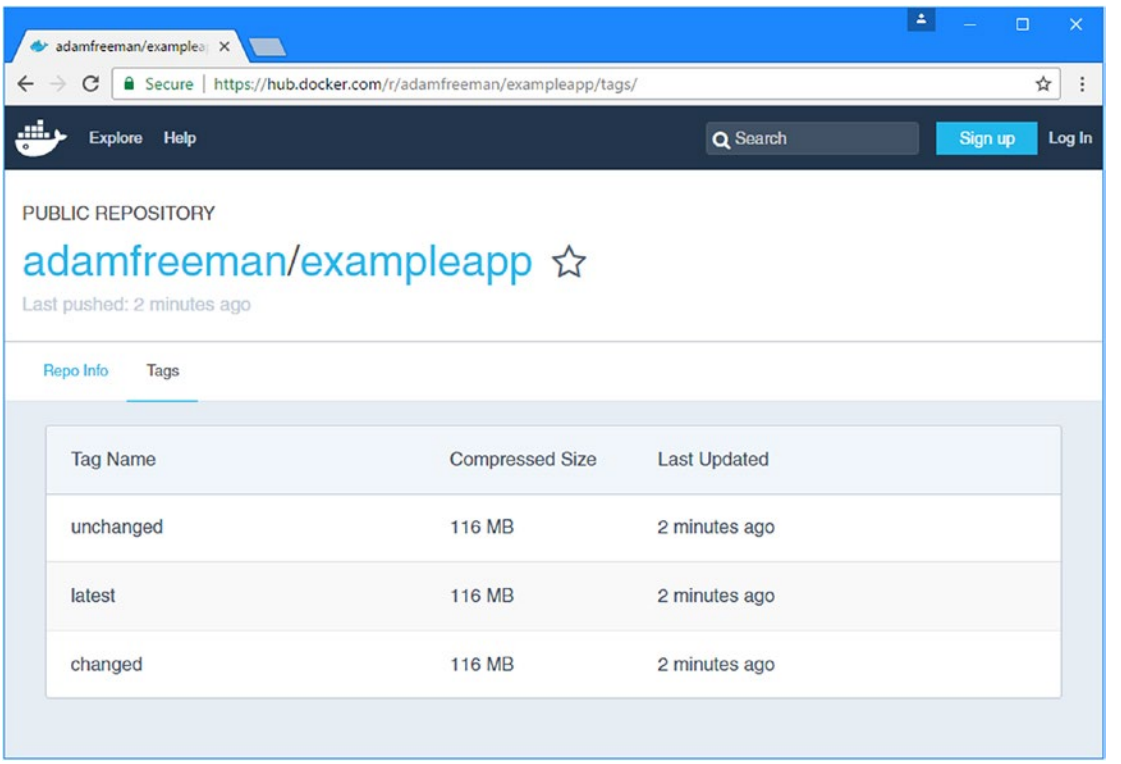

Figure 4-6. *Publishing images to the Docker Hub*

Creating Windows Containers

The images and containers created in the previous sections all rely on Linux as the execution platform. If you are using Linux as your development machine, the containers in the example have been running directly on your operating system using the Linux containers feature.

For Windows or macOS, Docker installs a Linux virtual machine that is used to execute containers. This blurs the lines between containers and virtual machines, but it does mean that a wider range of operating systems can be used to develop and test containers and that Windows and macOS users have access to the large library of images for containerized Linux applications that are available through Docker Hub.

But Linux isn't the only operating system that supports containers. Recent versions of Windows 10 and Windows Server 2016 also include container support that can be used to isolate and run Windows applications running on the Windows operating system.

From the perspective of ASP.NET Core MVC, support for Windows containers can be useful if you want to deploy your applications using Internet Information Services (IIS) or if your application depends on components that cannot run on Linux.

In the sections that follow, I explain how to create and test a Windows container. Creating a Windows container for an ASP.NET Core MVC application requires a similar process to the Linux equivalent but requires some important configuration changes.

■ **Note** Windows containers can be created only using the Windows operating system and can be deployed only to Windows Server 2016. Linux containers are more widely supported, and there are more base images to work with. You should use Linux containers unless you have a specific need to containerize a Windows-only application.

Switching to Windows Containers

You must tell Docker that you want to switch from using Linux containers (the default) to using Windows containers. Right-click the Docker icon in the Windows task bar and select Switch To Windows Containers, as shown in Figure 4-7. (You don't have to perform this step when using Windows Server 2016, which supports only Windows containers.)

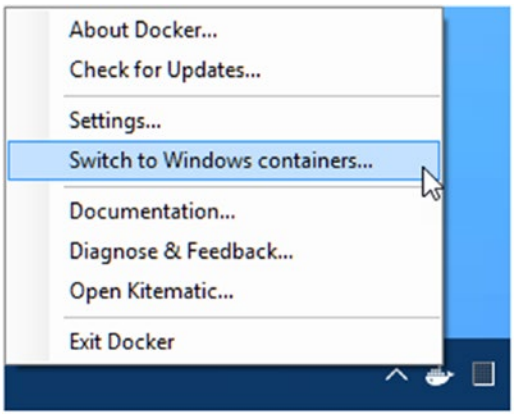

Figure 4-7. *Switching to Windows containers*

As part of the switch, you may be prompted to enable the Windows Containers feature, which will require a reboot.

If you have problems using Docker after you have switched between container types, then a reboot will usually solve the problem.

CREATING A WINDOWS SERVER VIRTUAL MACHINE

As I write this, the support for Windows containers on Windows 10 is at an early stage and has some problems, especially if you are using Hyper-V for conventional virtual machines alongside Docker. I found I got the best results by creating a regular Hyper-V virtual machine running Windows Server 2016 and using it to create and run my Docker Windows containers.

This technique requires switching on nested virtualization on the Windows 2016 virtual machine. Run the following command using PowerShell in the host operating system:

```
Set-VMProcessor -VMName <VMName> -ExposeVirtualizationExtensions $true
```

Replace `<VMName>` with the name assigned to the Windows Server 2016 virtual machine in Hyper-V. This command allows Docker to run in the Windows Server virtual machine and doesn't require switching between container types. See Chapter 3 for details of installing Docker in Windows Server 2016.

Creating a .NET Core Windows Image

Creating the image for a Windows container requires a different Docker file configuration. Create a file called `Dockerfile.windows` in the `ExampleApp` folder and add the commands shown in Listing 4-34.

Listing 4-34. The Contents of the Dockerfile.windows File in the ExampleApp Folder

```
FROM microsoft/dotnet:1.1.1-runtime-nanoserver

COPY dist /app

WORKDIR /app

EXPOSE 80/tcp

ENV ASPNETCORE_URLS http://+:80

ENTRYPOINT ["dotnet", "ExampleApp.dll"]
```

There are two differences between this Docker file and the one used for Linux containers. The first is that the `FROM` command specifies a different base image.

```
...
FROM microsoft/dotnet:1.1.1-runtime-nanoserver
...
```

The microsoft/dotnet image is the official image for .NET Core and is available in variations that provide different versions of .NET, a choice between the runtime or the SDK, and both Linux and Windows options. The 1.1.1-runtime-nanoserver variation specified in the listing contains the version 1.1.1 of the .NET Core runtime and is based on Windows. (Windows Nano Server is a minimal installation of Windows that is suitable for use in containers.)

■ **Tip** You can see the full set of variations and versions of the microsoft/dotnet image at the Docker Hub.

The second change is the addition of this ENV command:

```
...
ENV ASPNETCORE_URLS http://+:80
...
```

The ENV command sets an environment variable in the container. In this case, the command sets the value of the ASPNETCORE_URLS environment variable, which sets the port that the Kestrel server listens on to 80. The base image I used for the Linux containers includes this environment variable. Setting the port isn't a requirement but ensures that I can create and use Windows containers using the same Docker commands I used for the Linux containers earlier in the chapter.

Creating the Image and the Container

The process for creating an image and container for Windows is the same as for Linux. Run the commands in Listing 4-35 in the ExampleApp folder to publish the application and use the new Docker file to create an image.

Listing 4-35. Creating an Image for a Windows Container

```
dotnet restore
dotnet publish --framework netcoreapp1.1 --configuration Release --output dist
docker build . -t apress/exampleapp:windows -f Dockerfile.windows
```

To differentiate between the Linux and Windows containers for the example applications, I used the -t argument to the docker build command to specify a tag for the image that contains a variation so that the name of the image is apress/exampleapp:windows.

■ **Tip** If you see an unknown blob error when you run the command in Listing 4-35, then use the task bar icon to check that Docker has switched to Windows containers. If you had to reboot to enable containers, then Docker may have reset to Linux containers when it started.

Run the command shown in Listing 4-36 to create and start a new container using the Windows image.

Listing 4-36. Creating and Starting the Windows Container

```
docker run -p 7000:80 --name exampleAppWin apress/exampleapp:windows
```

Type Control+C to detach the command prompt and leave the container running. (The messages written out indicate that Control+C will terminate the application, but this is a message from Kestrel, the ASP.NET Core server, which doesn't receive the keystrokes. When working with Docker, it can be easy to confuse the container with the application that is running inside of it.)

Testing the Windows Container

At the time of writing, there is a problem with Windows containers that means they cannot be tested by requesting the mapped port through localhost, as I did for the Linux containers. This is a result of the way that the networking for containers is set up by Docker.

If you have another machine (or virtual machine available), then you can test the Windows container using the host operating system's IP address and the mapped port, which is 7000 in this example.

If you have only your development machine, then you can test using the IP address assigned to the container. Run the command in Listing 4-37 to get detailed information about the Windows container.

Listing 4-37. Inspecting a Container

```
docker inspect exampleAppWin
```

The output will include a Networks/nat section that contains an IP address, like this:

```
...
"Networks": {
     "nat": {
         "IPAMConfig": null,
         "Links": null,
         "Aliases": null,
         "NetworkID": "d41dba49b91fcd7fdfc5f7f520976db353",
         "EndpointID": "ca60ef2b19f591a0cf03b27407142cea7",
         "Gateway": "",
         "IPAddress": " 172.29.172.154",
         "IPPrefixLen": 16,
         "IPv6Gateway": "",
         "GlobalIPv6Address": "",
         "GlobalIPv6PrefixLen": 0,
         "MacAddress": "00:15:5d:68:1f:22"
     }
 }
...
```

Use this address to send an HTTP request to the container, using the port used by the application (and not the port mapped to the host operating system outside the container).

You may see a different IP address in the output from the docker inspect command, but for me, the URL that will test the container is http://172.29.172.154:80, producing the result shown in Figure 4-8.

■ **Note** The microsoft/dotnet image doesn't contain the natively compiled packages that are included in the microsoft/aspnetcore image that I used for the Linux image earlier in the chapter. This doesn't stop the container from running, but it does mean it takes slightly longer to process the first request.

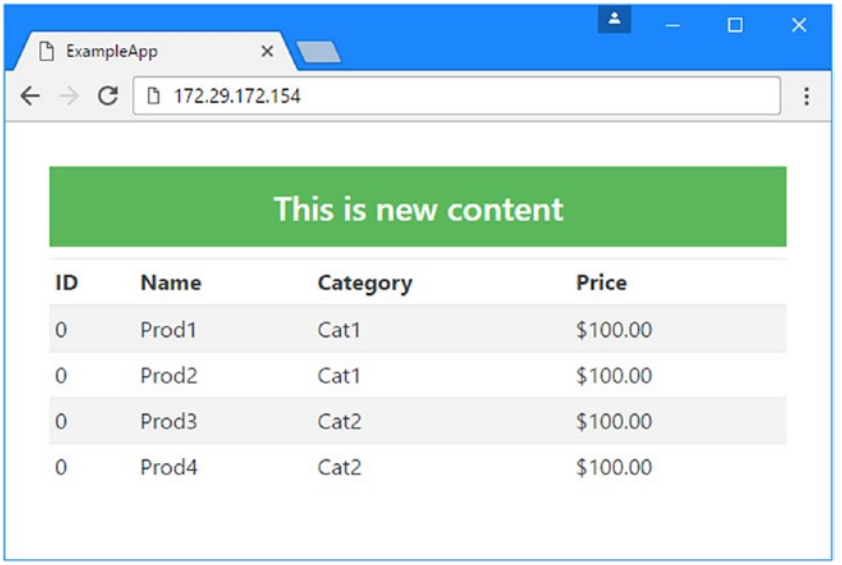

Figure 4-8. *Testing the Windows container*

Executing Commands in Windows Containers

If you are using a Windows container, then you can invoke commands inside the container with PowerShell. Open a second PowerShell and run the command shown in Listing 4-38 to execute a command inside the Windows container.

Listing 4-38. Executing a Command in a Windows Container

```
docker exec exampleAppWin PowerShell cat /app/Views/Home/Index.cshtml
```

This command returns the contents of the Razor view, as follows:

```
@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-success text-xs-center p-1 text-white">This is new content</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>
```

Interacting with Windows Containers

You can also interact with a container using PowerShell directly. Run the command shown in Listing 4-39 to interactively execute PowerShell inside the Windows container.

Listing 4-39. Starting a Shell in a Windows Container

```
docker exec -it exampleAppWin PowerShell
```

Once the shell has started, use the commands in Listing 4-40 to modify the contents of the Razor view.

Listing 4-40. Modifying the Razor View in PowerShell

```
cd C:\app\Views\Home
(Get-Content .\Index.cshtml).replace("This is new content", "Essential (Windows) Docker") |
Set-Content Index.cshtml
exit
```

The first command changes the working directory, and the second command replaces the phrase `This is new content` with `Essential (Windows) Docker`. The final command exits the shell. If you use the browser to send a request to the browser (as explained in the *"Creating Windows Containers"* section), you will see the effect of the change, as illustrated by Figure 4-9.

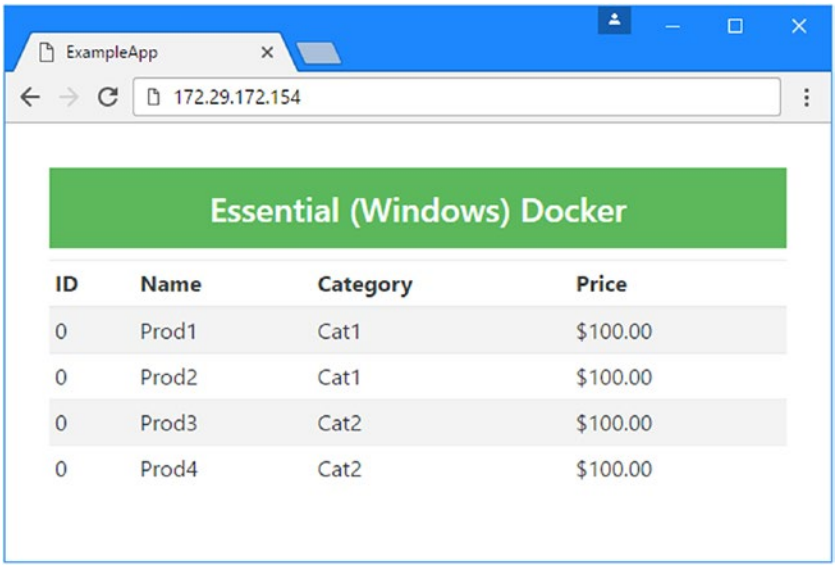

Figure 4-9. *Using the interactive shell in a Windows container*

Summary

In this chapter, I demonstrated how to create and manage Docker images and containers. I showed you how to create Windows and Linux containers, how to modify containers and use the changes to create new images, and how to publish images to the Docker Hub. In the next chapter, I explain how Docker deals with application data and how multiple containers can be connected together using software-defined networks.