Containers and Path to the Cloud Lab 1

Build a Containerized Web Application with Docker

(https://docs.microsoft.com/en-us/learn/modules/intro-to-containers/)

Retrieve an existing Docker image and deploy it locally

Docker is a technology that enables you to deploy applications and services quickly and easily. A Docker app runs using a Docker image. A Docker image is a prepackaged environment containing the application code and the environment in which the code executes.

In the corporate scenario described earlier, you want to investigate the feasibility of packaging and running an app with Docker. You decide to build and deploy a Docker image running a test web app.

In this unit, you'll learn about the key concepts and processes involved in running a containerized app stored in a Docker image.

Overview of Docker

Docker is a tool for running containerized apps. A containerized app includes the app and the filesystem that makes up the environment in which it runs. For example, a containerized app could consist of a database and other associated software and configuration information needed to run the app.

A containerized app typically has a much smaller footprint than a virtual machine configured to run the same app. This smaller footprint is because a virtual machine has to supply the entire operating system and associated supporting environment. A Docker container doesn't have this overhead because Docker uses the operating system kernel of the host computer to power the container. Downloading and starting a Docker image is typically much faster and more space-efficient than downloading and running a virtual machine that provides similar functionality.

You create a containerized app by building an **image** that contains a set of files and a section of configuration information used by Docker. You run the app by asking Docker to start a container based on the image. When the container starts, Docker uses the image configuration to determine what application to run inside the container. Docker

provides the operating system resources and the necessary security to ensure that containers are running concurrently and remain *relatively* isolated.

Important

Docker does not provide the level of isolation available with virtual machines. A virtual machine implements isolation at the hardware level. Docker containers share underlying operating system resources and libraries. However, Docker ensures that one container cannot access the resources of another unless the containers are configured to do so.

You can run Docker on your desktop or laptop if you're developing and testing locally. For production systems, Docker is available for server environments, including many variants of Linux and Microsoft Windows Server 2016. Many vendors also support Docker in the cloud. For example, you can store Docker images in Azure Container Registry and run containers with Azure Container Instances.

In this unit, you'll use Docker locally to build and run an image, then upload it to Azure Container Registry and run it in an Azure Container Instance. This version of Docker is suitable for local development and testing of Docker images.

Linux and Windows Docker images

Docker was initially developed for Linux and has expanded to support Windows. Individual Docker images are either Windows-based or Linux-based, but can't be both at the same time. The operating system of the image determines what kind of operating system environment is used inside the container.

Authors of Docker images who wish to offer similar functionality in both Linux-based and Windows-based images can build those images separately. For example, Microsoft offers Windows and Linux Docker images containing an ASP.NET Core environment that can be used as the basis for containerized ASP.NET Core applications.

Linux computers with Docker installed can only run Linux containers. Windows computers with Docker installed can run both kinds of containers. Windows accomplishes this by using a virtual machine to run a Linux system and uses the virtual Linux system to run Linux containers.

In this module, you will build and run a Linux-based image.

Docker registries and Docker Hub

Docker images are stored and made available in *registries*. A registry is a web service to which Docker can connect to upload and download container images. The most well-known registry is Docker Hub, which is a public registry. Many individuals and organizations publish images to Docker Hub, and you can download and run these images using Docker running on your desktop, on a server, or in the cloud. You can create a Docker Hub account and upload your images there for free.

A registry is organized as a series of *repositories*. Each repository contains multiple Docker images that share a common name and generally the same purpose and functionality. These images normally have different versions identified with a tag. This mechanism enables you to publish and retain multiple versions of images for compatibility reasons. When you download and run an image, you must specify the registry, repository, and version tag for the image. Tags are text labels, and you can use your version numbering system (v1.0, v1.1, v1.2, v2.0, and so on).

Suppose you want to use the ASP.NET Core Runtime Docker image. This image is available in two versions:

- `mcr.microsoft.com/dotnet/core/aspnet:2.2`
- `mcr.microsoft.com/dotnet/core/aspnet:2.1`

Now, let's suppose you want to use the .NET Core samples Docker images. Here we have four versions available to choose from:

- `mcr.microsoft.com/dotnet/core/samples:dotnetapp`
- `mcr.microsoft.com/dotnet/core/samples:aspnetapp`
- `mcr.microsoft.com/dotnet/core/samples:wcfservice`
- `mcr.microsoft.com/dotnet/core/samples:wcfclient`

Note

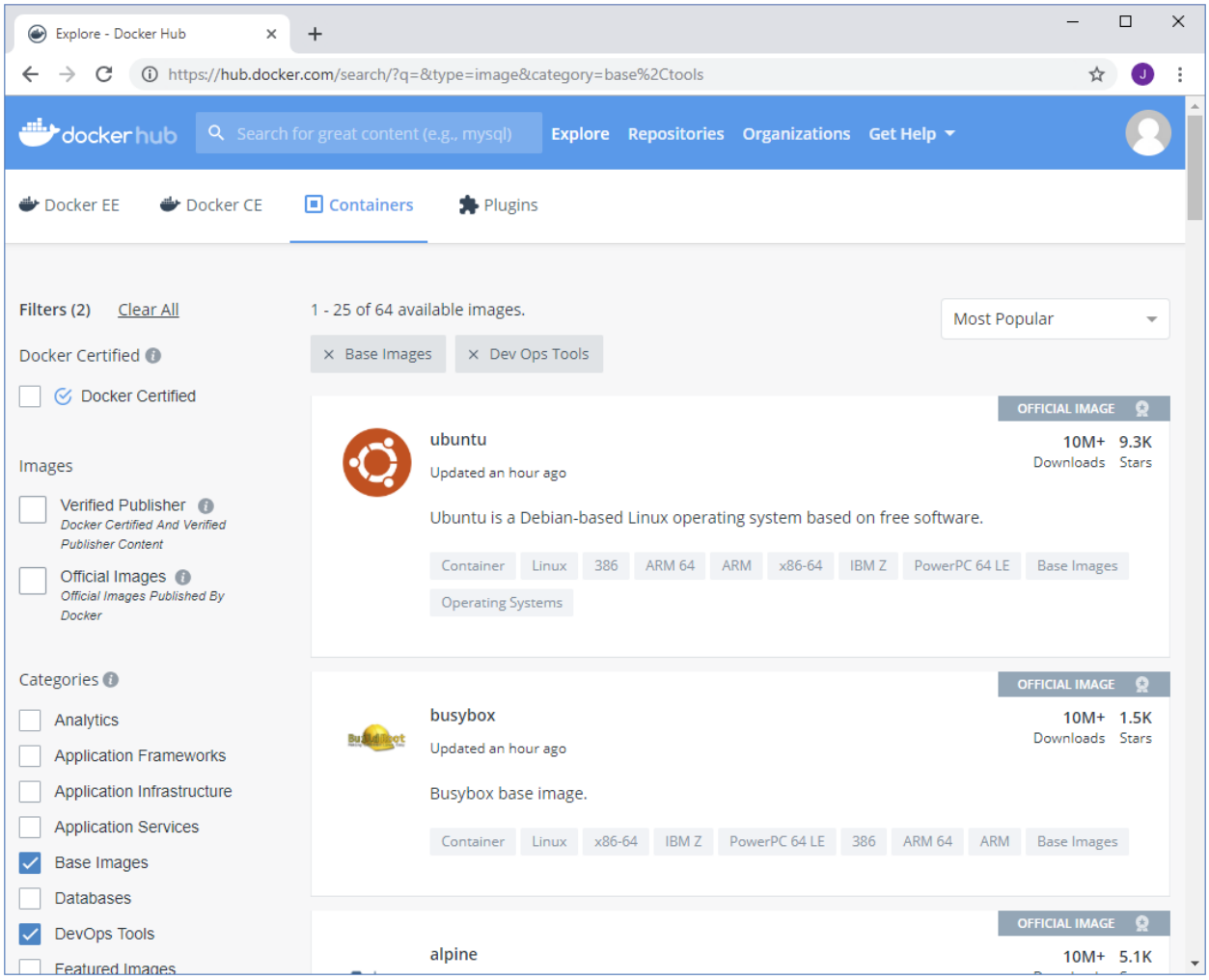
A single image can have multiple tags assigned to it. By convention, the most recent version of an image is assigned the tag *latest* in addition to a tag that describes its version number. When you release a new version of an image, you can reassign the *latest* tag to reference the new image.

A repository is also the unit of privacy for an image. If you don't wish to share an image, you can make the repository private. You can grant access to other users with whom you want to share the image.

Browse Docker Hub and pull an image

Often you'll find that there's an image in Docker Hub that closely matches the type of app you want to containerize. You can download such an image and extend it with your application code.

Docker Hub contains many thousands of images. You can search and browse a registry using Docker from the command line or the Docker Hub website. The website allows you to search, filter, and select images by type and publisher. The figure below shows an example of the search page.



You use the `docker pull` command with the image name to retrieve an image. By default, Docker will download the image tagged `latest` from that repository on Docker Hub if you specify only the repository name. Keep in mind that you can modify the command to pull different tags and from different repositories. This example fetches the image with the tag `aspnetapp` from

the *mcr.microsoft.com/dotnet/core/samples:aspnetapp* repository. This image contains a simple ASP.NET Core web app.

Note

The examples in this unit are intended to show the syntax of the various Docker commands. You don't need to run these commands while reading this unit. The exercises that follow this unit will guide you through working with Docker directly.

bash

```
docker pull mcr.microsoft.com/dotnet/core/samples:aspnetapp
```

When you fetch an image, Docker stores it locally and makes it available for running containers. You can view the images in your local registry with the *docker image list* command.

bash

```
docker image list
```

The output looks like the example below.

console

```
REPOSITORY TAG IMAGE ID CREATED SIZE
mcr.microsoft.com/dotnet/core/samples   aspnetapp           6e2737d83726        6
days ago          263MB
```

You use the image name ID to reference the image in many other Docker commands.

Run a Docker container

Use the `docker run` command to start a container. Specify the image to run with its name or ID. If you haven't `docker pull`ed the image already, Docker will do it for you.

bash

```
docker run mcr.microsoft.com/dotnet/core/samples:aspnetapp
```

In this example, the command will respond with the following message:

console

```
warn: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[35]
 No XML encryptor configured. Key {d8e1e1ea-126a-4383-add9-d9ab0b56520d} may be
persisted to storage in unencrypted form.
Hosting environment: Production
Content root path: /app
Now listening on: http://[::]:80
Application started. Press Ctrl+C to shut down.
```

This image contains a web app, so it's now listening for requests to arrive on HTTP port 80. However, if you open a web browser and navigate to `http://localhost:80`, you won't see the app.

By default, Docker doesn't allow inbound network requests to reach your container. You need to tell Docker to assign a specific port number from your computer to a specific port number in the container by adding the `-p` option to `docker run`. This instruction enables network requests to the container on the specified port.

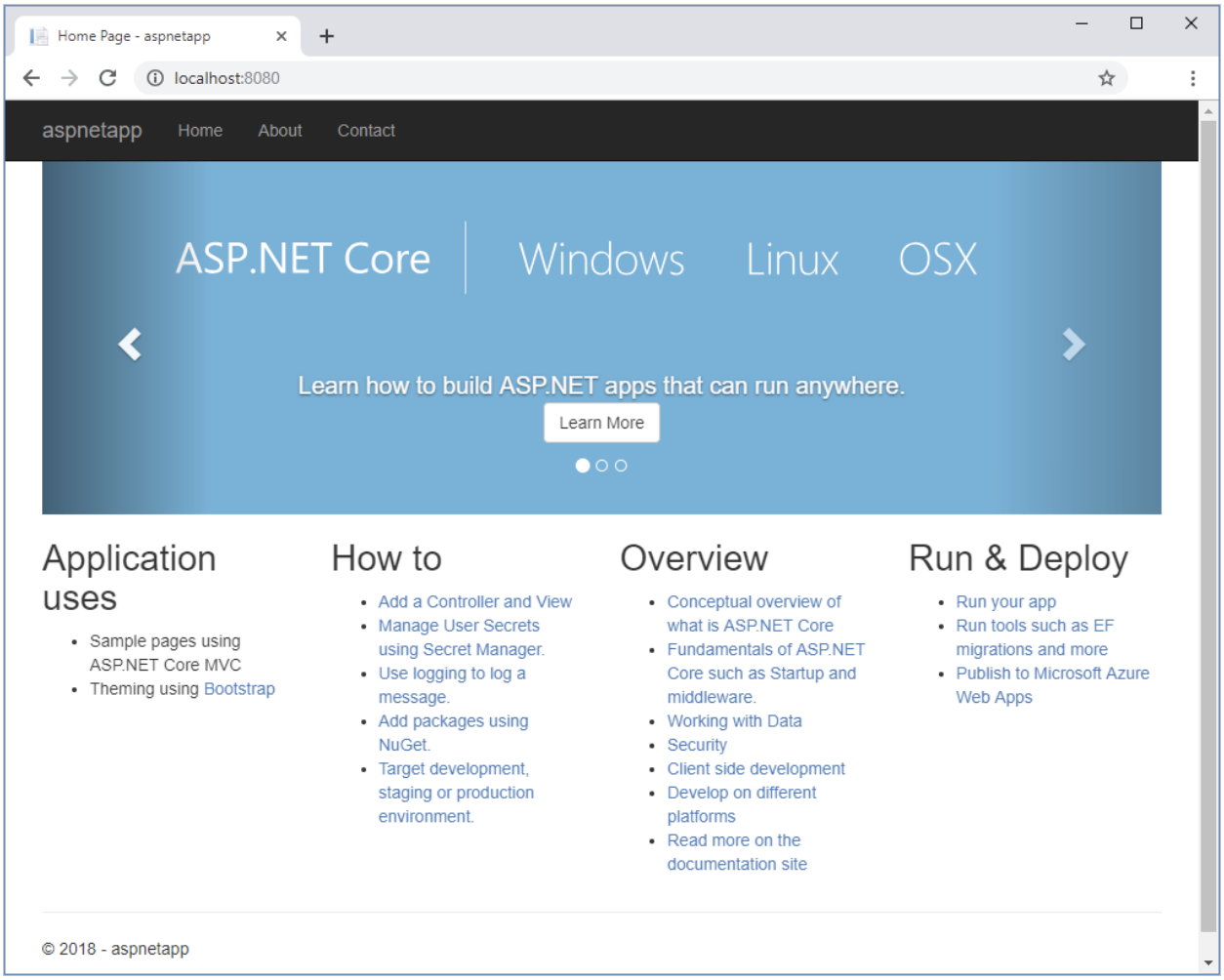
Additionally, the web app in this image isn't meant to be used interactively from the command line. When we start it, we want Docker to start it in the background and just let it run. Use the `-d` flag to instruct Docker to start the web app in the background.

You can press Ctrl-C to stop the image and then restart it as shown by the following example.

bash

```
docker run -p 8080:80 -d mcr.microsoft.com/dotnet/core/samples:aspnetapp
```

The command maps port 80 in the container to port 8080 on your computer. If you visit the page http://localhost:8080 in a browser, you'll see the sample web app.



Containers and files

If a running container makes changes to the files in its image, those changes only exist in the container where the changes are made. Unless you take specific steps to preserve the state of a container, these changes will be lost when the container is removed. Similarly, multiple containers based on the same image that run simultaneously do not share the files in the image - each container has its own independent copy. Any data written by one container to its filesystem are not visible to the other.

It's possible to add writable volumes to a container. A volume represents a filesystem that can be mounted by the container, and is made available to the application running in the container. The data in a volume does persist when the container stops, and multiple containers can share the same volume. The details for creating and using volumes are outside the scope of this module.

It is a best practice to avoid the need to make changes to the image filesystem for applications deployed with Docker. Only use it for temporary files that can afford to be lost.

Manage Docker containers

You can view active containers with the `docker ps` command.

bash

```
docker ps
```

The output includes the status of the container. *Up* if it is running, *Exited* if it has terminated, among other values such as the command line flags specified when the image was started, and additional information. Docker lets you run multiple containers from the same image simultaneously, so each container is assigned a unique ID as well as a unique human-readable name. Most Docker commands used to manage individual containers can use either the ID or the name to refer to a specific container.

In the output below, you can see two containers. The *PORTS* field shows that the container with ID `lucid-jang` is the image running with port 80 on the Docker host mapped to port 8080 on your computer. The `youthful_heisenberg` instance is the container for the previous run of the image. The *COMMAND* field shows the command that the container ran to start the application in the image. In this case, for both containers, it is *dotnet aspnetapp.dll*. Note that the image ID for the containers is also the same because both containers are executing the same image.

console

```
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
57b9587583e3        mcr.microsoft.com/dotnet/core/samples:aspnetapp   "dotnet
aspnetapp.dll"   42 seconds ago      Up 41 seconds       0.0.0.0:8080->80/tcp
elegant_ramanujan
```

Note

`docker ps` is a shortcut for `docker container ls`. The names of these commands are based on the Linux utilities `ps` and `ls`, which list running processes and files, respectively.

You can stop an active container with the `docker stop` command and specify the container ID.

bash

```
docker stop elegant_ramanujan
```

If you run `docker ps` again, you'll see that the *lucid_jang* container is no longer present in the output. The container still exists, but is no longer hosting a running process. You can include stopped containers in the output of `docker ps` by including the `-a` flag:

console

```
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
57b9587583e3        mcr.microsoft.com/dotnet/core/samples:aspnetapp   "dotnet
aspnetapp.dll"   2 minutes ago       Exited (0) 21 seconds ago
elegant_ramanujan
```

You can restart a stopped container with the `docker start` command. The main process of the container will be started anew.

bash

```
docker start elegant_ramanujan
```

Typically, once a container is stopped, it should also be removed. Removing a container cleans up any resources it leaves behind. Once you remove a container, any changes made within its image filesystem are permanently lost.

bash

```
docker rm elegant_ramanujan
```

You can't remove a container that is running, but you can force a container to be stopped and removed with the *-f* flag to the `docker rm` command. This is a quick way to stop and remove a container, but should only be used if the app inside the container does not need to perform a graceful shutdown.

bash

```
docker container rm -f elegant_ramanujan
```

Remove Docker images

You can remove an image from the local computer with the `docker image rm` command. Specify the image ID of the image to remove. This example removes the image for the sample web app:

bash

```
docker image rm mcr.microsoft.com/dotnet/core/samples:aspnetapp
```

Containers running the image must be terminated before the image can be removed. If the image is still in use by a container, you'll get an error message like the one shown below. In this example, the error occurs because the *youthful_hesienburg* container is still using the image.

console

```
Error response from daemon: conflict: unable to delete 575d85b4a69b (cannot be forced) - image is being used by running container c13165988cfe
```

Exercise - Retrieve an existing Docker image and deploy it locally

- 6 minutes

A good starting point for building and running your own Docker images is to take an existing image from Docker Hub and run it locally on your computer.

As a proof-of-concept for the company's applications, you decide to try running a sample image from Docker Hub. The image you have selected implements a basic .NET Core ASP.NET web app. Once you've established a process for deploying a Docker image, you'll be able to run one of your company's own web apps using Docker.

In this exercise, you'll pull an image from Docker Hub and run it. You'll examine the local state of Docker to help understand the elements that are deployed. Finally, you'll remove the container and image from your computer.

Important

This exercise takes place on your computer, not in Azure. You need a local installation of Docker to proceed with the exercise.

Pull and run a sample application from Docker Hub

1. Open a command prompt window on your local computer.
2. Pull the **ASP.NET Sample** app image from the Docker Hub registry. This image contains a sample web app developed by Microsoft. It's based on the default ASP.NET template available in Visual Studio.

bash

```
docker pull mcr.microsoft.com/dotnet/core/samples:aspnetapp
```

3. Verify that the image has been stored locally.

   bash

   ```
   docker image list
   ```

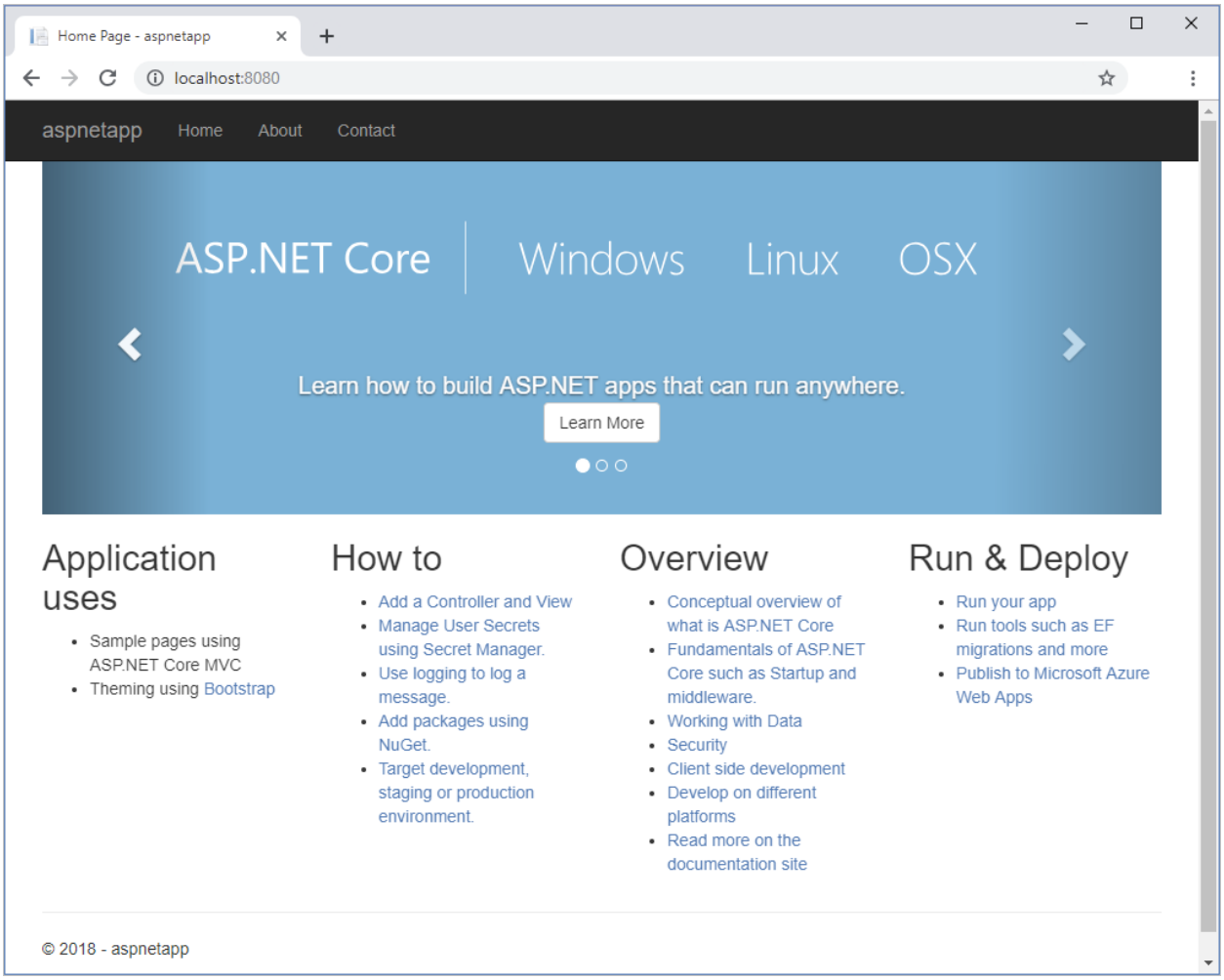
   You should see a repository named *mcr.microsoft.com/dotnet/core/samples* with a tag of *aspnetapp*.

4. Start the sample app. Specify the *-d* flag to run it as a background, non-interactive app. Use the *-p* flag to map port 80 in the container that is created to port 8080 locally, to avoid conflicts with any web apps already running on your computer. The command will respond with a lengthy hexadecimal identifier for the instance.

   bash

   ```
   docker run -d -p 8080:80 mcr.microsoft.com/dotnet/core/samples:aspnetapp
   ```

5. Open a web browser and go to the page for the sample web app at http://localhost:8080. The page looks like the following screenshot.



Examine the container in the local Docker registry

1. At the command prompt, view the running containers in the local registry.

   bash

   ```
   docker ps
   ```

   The output should look similar to this:

   console

   ```
   CONTAINER ID        IMAGE                                        COMMAND
   CREATED             STATUS              PORTS                    NAMES
   bffd59ae5c22        mcr.microsoft.com/dotnet/core/samples:aspnetapp   "dotnet
   aspnetapp.dll"   12 seconds ago      Up 11 seconds       0.0.0.0:8080->80/tcp
   competent_hoover
   ```

The **COMMAND** field shows the container started by running the command *dotnet aspnetapp.dll*. This command invokes the .NET Core runtime to start the code in the aspnetapp.dll (the code for the sample web app). The *PORTS* field indicates port 80 in the image was mapped to port 8080 on your computer.
The *STATUS* field shows the application is still running. Make a note of the container's *NAME*.

2. Stop the Docker container. Specify the container name for the web app in the following command, in place of `<NAME>`.

   bash

   ```
   docker container stop <NAME>
   ```

3. Verify that the container is no longer running. The following command shows the status of the container as *Exited*. The *-a* flag indicates that the command shows the status of all containers, not just those that are still running.

   bash

   ```
   docker ps -a
   ```

4. Return to the web browser and refresh the page for the sample web app. It should fail with a *Connection Refused* error.

Remove the container and image from the local registry

1. Although the container has stopped, it's still loaded and can be restarted. Remove it using the following command. As before, replace `<NAME>` with the name of your container.

   bash

   ```
   docker container rm <NAME>
   ```

2. Verify that the container has been removed with the following command. The command should no longer list the container.

   bash

   ```
   docker ps -a
   ```

3. List the images currently available on your computer.

bash

```
docker image list
```

4. Remove the image from the registry.

   bash

   ```
   docker image rm mcr.microsoft.com/dotnet/core/samples:aspnetapp
   ```

5. List the images again to verify that the image for the *microsoft/dotnet-samples* web app has disappeared.

   bash

   ```
   docker image list
   ```

Customize a Docker image to run your own web app

- 5 minutes

Docker Hub is an excellent source of images to get you started building your own containerized apps. You can download an image that provides the basic functionality you require, and *layer* your own application on top of it to create a new custom image. You can automate the steps for doing this process by writing a Dockerfile.

In the scenario of the online clothing store, the company has decided that Docker is the way forward. The next step is to determine the best way to containerize your web applications. The company plans to build many of the apps using ASP.NET Core, and you've noticed that Docker Hub contains a base image that includes this framework. As a proof of concept, you want to take this base image and add the code for one of the web apps to create a new custom image. You also want this process to be easily repeatable, so it can be automated whenever a new version of the web app is released.

In this unit, you'll learn how to create a custom Docker image, and how you can automate the process by writing a Dockerfile.

Create a custom image with a Dockerfile

To create a Docker image containing your application, you'll typically begin by identifying a *base image* to which you add additional files and configuration. The process of identifying a suitable base image usually starts with a search on Docker Hub

for a ready-made image that already contains an application framework and all the utilities and tools of a Linux distribution like Ubuntu or Alpine. For example, if you have an ASP.NET Core application that you want to package into a container, Microsoft publishes an image called `mcr.microsoft.com/dotnet/core/aspnet` that already contains the ASP.NET Core runtime.

An image can be customized by starting a container with the base image and making changes to it. Changes usually involve activities like copying files into the container from the local filesystem, and running various tools and utilities to compile code. When finished, you would use the `docker commit` command to save the changes to a new image.

Manually completing the above process is time-consuming and error-prone. It could be scripted with a script language like Bash, but Docker provides a more effective way of automating image creation via a *Dockerfile*.

A Dockerfile is a plain text file containing all the commands needed to build an image. Dockerfiles are written in a minimal scripting language designed for building and configuring images. documents the operations required to build an image starting with a base image.

The following example shows a Dockerfile that builds a .NET Core 2.2 application and packages it into a new image.

Dockerfile

```
FROM mcr.microsoft.com/dotnet/core/sdk:2.2
WORKDIR /app
COPY myapp_code .
RUN dotnet build -c Release -o /rel
EXPOSE 80
WORKDIR /rel
ENTRYPOINT ["dotnet", "myapp.dll"]
```

In this file:

- The **FROM** statement downloads the specified image and creates a new container based on this image.
- The **WORKDIR** command sets the current working directory in the container, used by the following commands.
- The **COPY** command copies files from the host computer to the container. The first argument (`myapp_code`) is a file or folder on the host computer. The second argument (`.`)

specifies the name of the file or folder to act as the destination in the container. In this case, the destination is the current working directory (`/app`).
- The **RUN** command executes a command in the container. Arguments to the RUN command are command-line commands.
- The **EXPOSE** command creates configuration in the new image that specifies which ports are intended to be opened when the container is run. If the container is running a web app, it's common to EXPOSE port 80.
- The **ENTRYPOINT** command specifies the operation the container should run when it starts. In this example, it runs the newly built app. You specify the command to be run and each of its arguments as a string array.

By convention, applications meant to be packaged as Docker images typically have a Dockerfile located in the root of their source code, and it's almost always named `Dockerfile`.

The `docker build` command creates a new image by running a Dockerfile. The *-f* flag indicates the name of the Dockerfile to use. The *-t* flag specifies the name of the image to be created, in this example, *myapp:v1*. The final parameter, ., provides the *build context* for the source files for the **COPY** command: the set of files on the host computer needed during the build process.

bash

```
docker build -t myapp:v1 .
```

Behind the scenes, the `docker build` command creates and runs a container, runs commands in it, then commits the changes to a new image.

Exercise - Customize a Docker image to run your own web app

- 12 minutes

A Dockerfile contains the steps for building a custom Docker image.

You now decide to deploy one of your organization's web apps using Docker. You select a simple web app that implements a web API for a hotel reservations web site. The web API exposes HTTP POST and GET operations that create and retrieve customer's bookings.

Note

In this version of the web app, the bookings are not actually persisted, and queries return dummy data.

In this exercise, you'll create a Dockerfile for an app that doesn't have one. Then, you'll build the image and run it locally.

Create a Dockerfile for the web app

1. In a command prompt window on your local computer, run the following command to download the source code for the web app.

   bash

   ```
   git clone https://github.com/MicrosoftDocs/mslearn-hotel-reservation-system.git
   ```

2. Move to the `src` folder.

   bash

   ```
   cd mslearn-hotel-reservation-system/src
   ```

3. In this directory, create a new file named `Dockerfile` with no file extension and open it in a text editor. On Windows, you can run the following commands:

   bash

   ```
   copy NUL Dockerfile
   notepad Dockerfile
   ```

4. Add the following commands to the Dockerfile. These commands fetch an image containing the .NET Core Framework SDK. The project files for the web app (`HotelReservationSystem.csproj`) and the library project (`HotelReservationSystemTypes.csproj`) are copied to the /src folder in the container. The *dotnet restore* command downloads the dependencies required by these projects from NuGet.

   Dockerfile

   ```
   FROM mcr.microsoft.com/dotnet/core/sdk:2.2
   WORKDIR /src
   COPY ["HotelReservationSystem/HotelReservationSystem.csproj",
   "HotelReservationSystem/"]
   COPY ["HotelReservationSystemTypes/HotelReservationSystemTypes.csproj",
   "HotelReservationSystemTypes/"]
   RUN dotnet restore "HotelReservationSystem/HotelReservationSystem.csproj"
   ```

5. Append the following commands to the Dockerfile. These commands copy the source code for the web app to the container and then run the dotnet build command to build the app. The resulting DLLs are written to the /app folder in the container.

   Dockerfile

   ```
   COPY . .
   WORKDIR "/src/HotelReservationSystem"
   RUN dotnet build "HotelReservationSystem.csproj" -c Release -o /app
   ```

6. Add the following command to the Dockerfile. The `dotnet publish` command copies the executables for the web site to a new folder and removes any interim files. The files in this folder can then be deployed to a web site.

   Dockerfile

   ```
   RUN dotnet publish "HotelReservationSystem.csproj" -c Release -o /app
   ```

7. Add the following commands to the Dockerfile. The first command opens port 80 in the container. The second command moves to the `/app` folder containing the published version of the web app. The final command specifies that when the container runs it should execute the command `dotnet HotelReservationSystem.dll`. This library contains the compiled code for the web app

   Dockerfile

   ```
   EXPOSE 80
   WORKDIR /app
   ENTRYPOINT ["dotnet", "HotelReservationSystem.dll"]
   ```

8. Save the file and close your text editor.

Build and deploy the image using the Dockerfile

1. At the command prompt, run the following command to build the image for the sample app using the Dockerfile. Don't forget the `.` at the end of the command. This command builds the image and stores it locally. The image is given the name `reservationsystem`. Verify that the image is built successfully. A warning about file and directory permissions will be displayed when the process completes. You can ignore these warnings for the purposes of this exercise.

bash

```
docker build -t reservationsystem .
```

2. Run the following command to verify that the image has been created and stored in the local registry.

   bash

   ```
   docker image list
   ```

   The image will have the name `reservationsystem`. You'll also see an image named *microsoft/dotnet*. This image contains the .NET Core SDK and was downloaded when the `reservationsystem` image was built using the Dockerfile.

   console

   ```
   REPOSITORY          TAG                 IMAGE ID            CREATED
   SIZE
   reservationsystem   latest              d2501f0f2ced        About a minute ago
   1.76GB
   microsoft/dotnet    2.1-sdk             c17aa78d71c2        8 days ago
   1.73GB
   ```

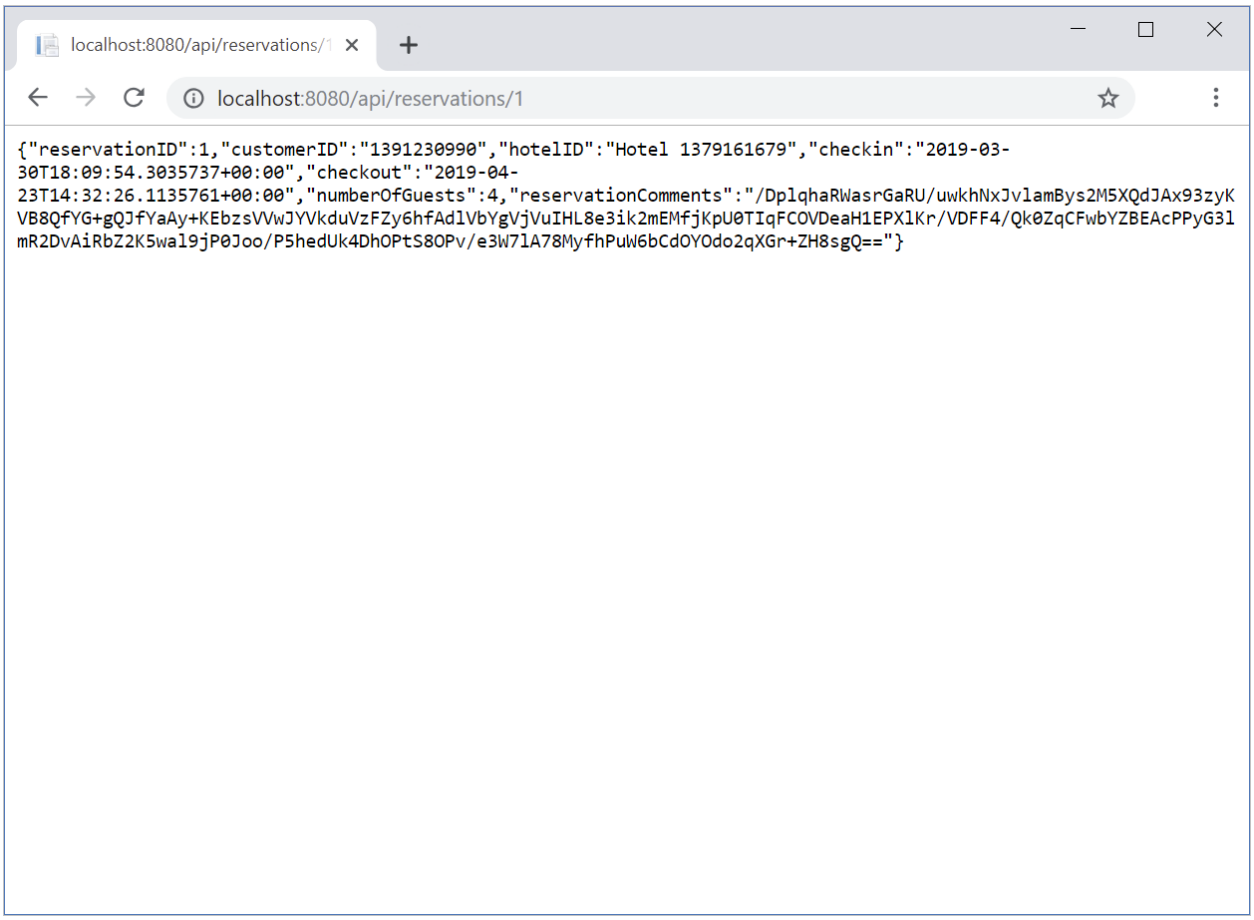
Test the web app

1. Run a container using the `reservationsystem` image using the following command. Docker will respond with a lengthy string of hex digits – the container runs in the background without any UI. Port 80 in the container is mapped to port 8080 on the host machine. The container is named `reservations`.

   bash

   ```
   docker run -p 8080:80 -d --name reservations reservationsystem
   ```

2. Start a web browser and navigate to http://localhost:8080/api/reservations/1. You should see a JSON document containing the data for reservation number 1 returned by the web app. You can replace the "1" with any reservation number, and you'll see the corresponding reservation details.



3. Examine the status of the container using the following command.

   bash

   ```
   docker ps -a
   ```

   Verify that the status of the container is *Up*.

   console

   ```
   CONTAINER ID        IMAGE               COMMAND                  CREATED
   STATUS              PORTS                  NAMES
   07b0d1de4db7        reservationsystem   "dotnet HotelReserva…"   5 minutes ago
   Up 5 minutes        0.0.0.0:8080->80/tcp   reservations
   ```

4. Stop the *reservations* container with the following command.

   bash

   ```
   docker container stop reservations
   ```

5. Delete the *reservations* container from the local registry.

   bash

   ```
   docker rm reservations
   ```

6. Leave the *reservationsystem* in the local registry. You will use this image in the next exercise.

You've now created an image for your web app and run it using a Docker container.