

# 9

# Containerizing Your Application with Docker

In the last few years, one technology in particular has been making headlines on the net, on social networks, and at events—Docker.

Docker is a containerization tool that became open source in 2013. It allows you to isolate an application from its host system so that the application becomes portable, and the code tested on a developer's workstation can be deployed to production without any concerns about execution runtime dependencies. We'll talk a little about application containerization in this chapter.

A container is a system that embeds an application and its dependencies. Unlike a **virtual machine (VM)**, a container contains only a light **operating system (OS)** and the elements required for the OS, such as system libraries, binaries, and code dependencies.

To learn more about the differences between VMs and containers, and why containers will replace VMs in the future, I suggest you read this blog article: <https://blog.docker.com/2018/08/containers-replacing-virtual-machines/>.

The principal difference between VMs and containers is that each VM that is hosted on a hypervisor contains a complete OS and is therefore completely independent of the guest OS that is on the hypervisor.

Containers, however, don't contain a complete OS—only a few binaries—but they are dependent on the guest OS, and use its resources (**central processing unit (CPU)**, **random-access memory (RAM)**, and network).

In this chapter, we will learn how to install Docker on different platforms, how to create a Docker image, and how to register it in Docker Hub. Then, we'll discuss an example of a **continuous integration/continuous deployment (CI/CD)** pipeline that deploys a Docker image in **Azure Container Instances (ACI)**. After that, we will show how to use Docker to run tools with **command-line interfaces (CLIs)**.

Finally, we will also learn the basic notions about **Docker Compose** and how to deploy Docker Compose containers in ACI.

This chapter covers the following topics:

- Installing Docker
- Creating a Dockerfile
- Building and running a container on a local machine
- Pushing an image to Docker Hub
- Pushing a Docker image to a private registry (ACR)
- Deploying a container to ACI with a CI/CD pipeline
- Using Docker for running command-line tools
- Getting started with Docker Compose
- Deploying Docker Compose containers in ACI

## Technical requirements

This chapter has the following technical requirements:

- An Azure subscription. You can get a free account here: <https://azure.microsoft.com/en-us/free/>.
- For some Azure commands, we will use the Azure CLI. Refer to the documentation here: <https://docs.microsoft.com/en-us/cli/azure/install-azure-cli>.

- In the last part of this chapter, in the *Creating a CI/CD pipeline for the container* section, we will discuss Terraform and the CI/CD pipeline, which were explained in *Chapter 2, Provisioning Cloud Infrastructure with Terraform*, and *Chapter 7, Continuous Integration and Continuous Deployment*.

All of the source code for the scripts included in this chapter is available here:

<https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP09>.

Check out the following video to see the code in action:

<https://bit.ly/3t1Jov8>

## Installing Docker

Docker's daemon is free and very well suited to developers and small teams—it's what we'll use in this book.

Docker is a cross-platform tool that can be installed on Windows, Linux, or macOS and is also natively present on some cloud providers, such as **Amazon Web Services (AWS)** and Azure.

To operate, Docker needs the following elements:

- **The Docker client:** This allows you to perform various operations on the command line.
- **The Docker daemon:** This is Docker's engine.
- **Docker Registry:** This is a public registry (Docker Hub) or private registry of Docker images.

Before installing Docker, we will first create an account on Docker Hub.

## Registering on Docker Hub

Docker Hub is a public space called a **registry**, containing more than 2 million public Docker images that have been deposited by companies, communities, and even individual users.

To register on Docker Hub and list public Docker images, perform the following steps:

1. Go to <https://hub.docker.com/>, where you will see the following screen:

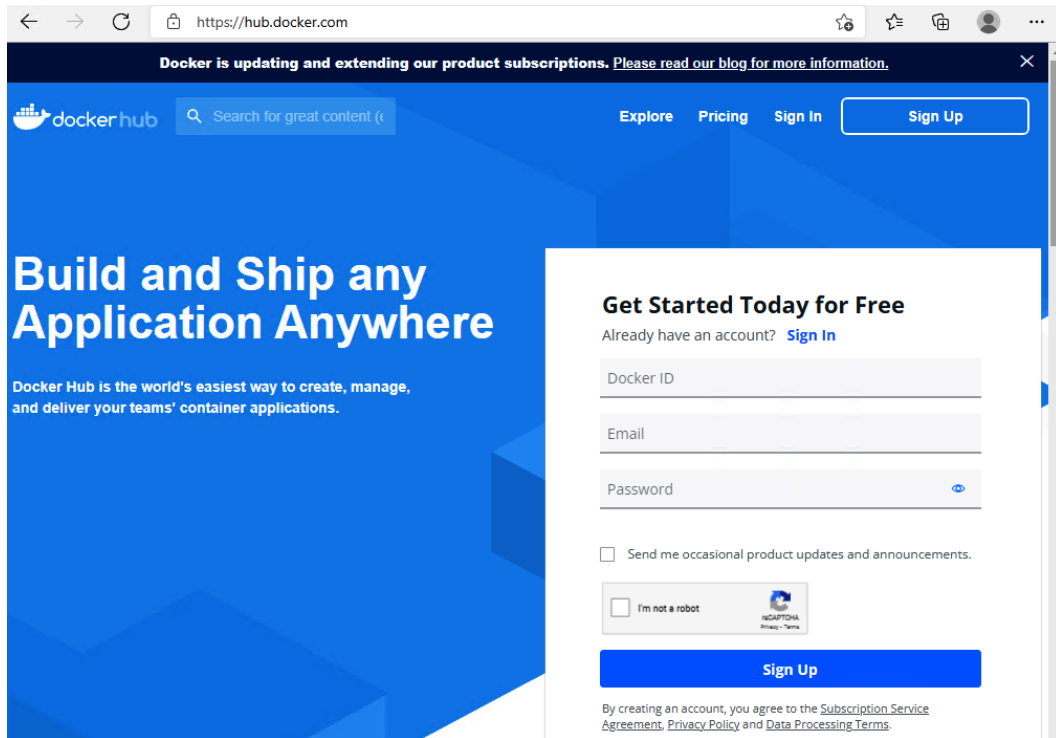


Figure 9.1 – Docker Hub login page

2. Fill in the form with a unique **identifier (ID)**, an email, and a password. Then, click on the **Sign Up** button.
3. Once your account is created, you can then log in to the site, and this account will allow you to upload custom images and download **Docker Desktop**.
4. To view and explore the images available from Docker Hub, go to the **Explore** section, as indicated in the following screenshot:

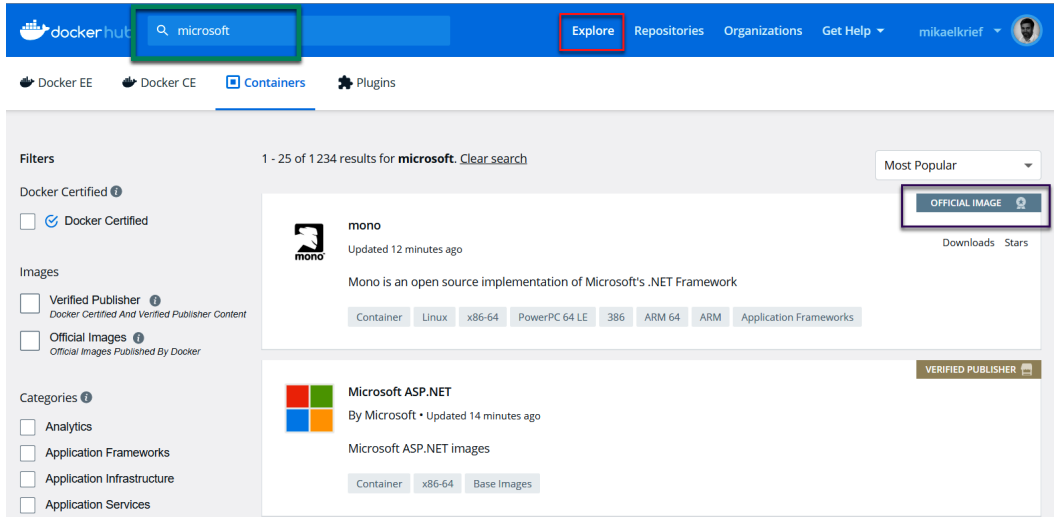


Figure 9.2 – Docker Hub Explore page

A list of Docker images is displayed with a search filter that you can use to search for official images or images from verified publishers, as well as images certified by Docker.

Having created an account on Docker Hub, we will now look at installing Docker on Windows.

## Docker installation

We'll now discuss the installation of Docker on Windows in detail.

Before installing Docker Desktop on Windows or macOS, we need to check all license options. For more information about Docker Desktop licensing, read the pricing page (<https://www.docker.com/pricing>) and the **frequently asked questions (FAQ)** page (<https://www.docker.com/pricing/faq>).

To install Docker Desktop on a Windows machine, it is necessary to first check the hardware requirements, which are outlined here:

- Windows 10/11 64-bit with at least 4 **gigabytes (GB)** of RAM
- **Windows Subsystem for Linux 2 (WSL 2)** backend or Hyper-V enabled. You can refer to this documentation in the event of any problems: <https://docs.docker.com/docker-for-windows/troubleshoot/#virtualization-must-be-enabled>.

**Note**

For more information about WSL, read the documentation here:

`https://docs.microsoft.com/en-us/windows/wsl/install`

More details about Docker Desktop requirements are specified here:

`https://docs.docker.com/desktop/windows/install/`

To install Docker Desktop, which is the same binary as the Docker installer for Windows and macOS, follow these steps:

1. First, download Docker Desktop by clicking on the **Docker Desktop for Windows** button on the install documentation page at `https://docs.docker.com/desktop/windows/install/`, as indicated in the following screenshot:



Figure 9.3 – Download link for Docker Desktop

2. Once that's downloaded, click on the downloaded **executable (EXE)** file.
3. Then, take the single configuration step, which is a possibility to install required components for the WSL 2 backend, as illustrated in the following screenshot:



Figure 9.4 – Docker Desktop configuration

In our case, we will check this option to install Windows components using WSL 2 as the backend.

4. Once the installation is complete, we'll get a confirmation message and a button to close the installation, as illustrated in the following screenshot:

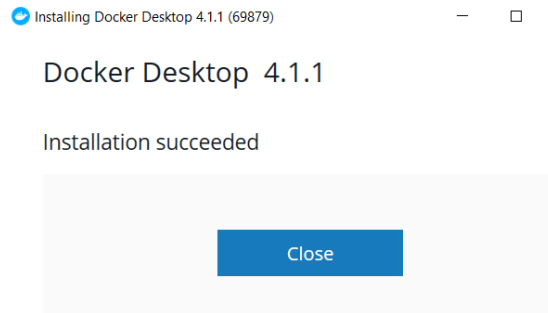


Figure 9.5 – Docker Desktop end installation

5. Finally, to start Docker, launch the Docker Desktop program. An icon will appear in the notification bar indicating that Docker is starting. It will then ask you to log in to Docker Hub via a small window. The startup steps of Docker Desktop are shown in the following screenshot:

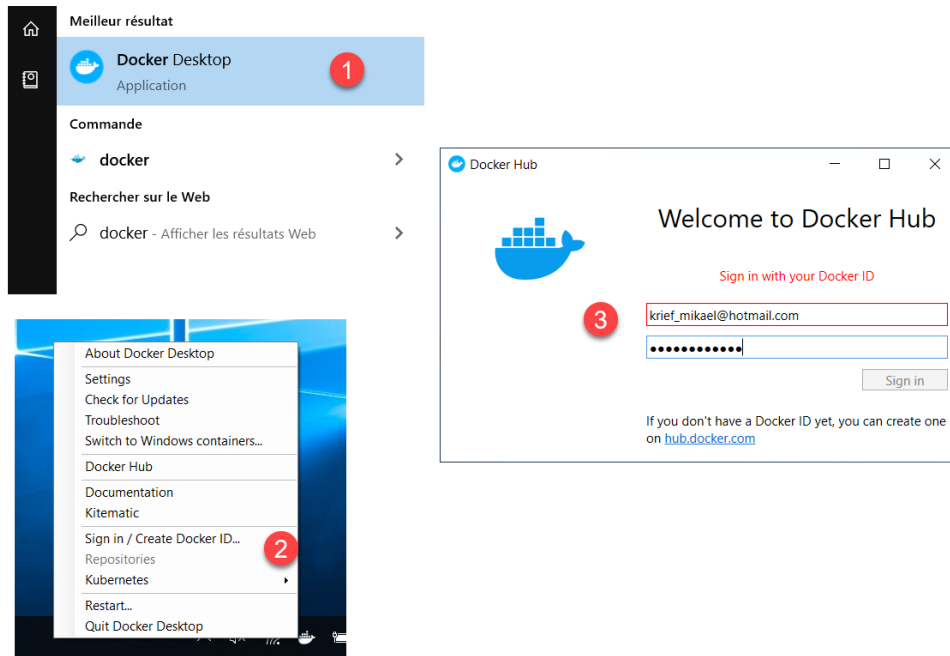


Figure 9.6 – Docker Hub sign-in from Docker Desktop

That's it! We've installed and started Docker on Windows.

To install Docker on another OS, you can read the documentation for each OS at <https://docs.docker.com/get-docker/>. Afterward, you can choose the desired target OS from this page, as shown in the following screenshot:

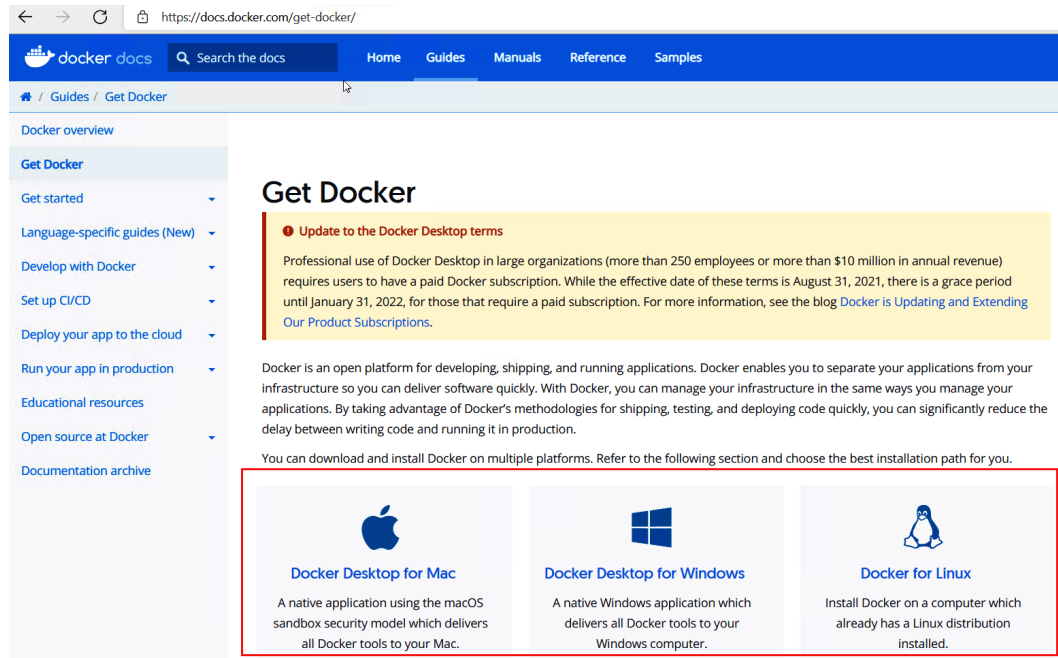


Figure 9.7 – Docker installation documentation

To check your Docker installation, open the Terminal window (it will also work on a Windows PowerShell Terminal) and execute the following command:

```
docker --help
```

You should be able to see something like this:



```
PS C:\Users\mkrief> docker --help
Usage:
  docker [flags]
  docker [command]

Available Commands:
  compose      Docker Compose
  context       Manage contexts
  ecs
  exec         Run a command in a running container
  help         Help about any command
  inspect      Inspect containers
  kill         Kill one or more running containers
  login        Log in to a Docker registry or cloud backend
  logout       Log out from a Docker registry or cloud backend
  logs         Fetch the logs of a container
  prune        prune existing resources in current context
  ps           List containers
  rm           Remove containers
  run          Run a container
  secret       Manages secrets
  serve        Start an api server
  start        Start one or more stopped containers
  stop         Stop one or more running containers
  version      Show the Docker version information
  volume       Manages volumes
```

Figure 9.8 – docker --help command

As you can see in the preceding screenshot, the command displays the different operations available in the Docker client tool.

Before looking at the execution of Docker commands in detail, it is important to have an overview of Docker's concepts.

## An overview of Docker's elements

Before executing Docker commands, we will discuss some of Docker's fundamental elements, which are **Dockerfiles**, **containers**, and **volumes**.

First of all, it is important to know that a **Docker image** is a basic element of Docker and consists of a text document called a Dockerfile that contains the binaries and application files we want to containerize.

A **Docker registry** is a centralized storage system for shared Docker images. This registry can be public—as in the case of Docker Hub—or private, such as with **Azure Container Registry (ACR)** or JFrog Artifactory.

A container is an instance that is executed from a Docker image. It is possible to have several instances of the same image within a container that the application will run. Finally, a volume is a storage space that is physically located on the host OS (that is, outside the container), and it can be shared across multiple containers if required. This space will allow the storage of persistent elements such as files or databases.

To manipulate these elements, we will use command lines, which will be discussed as we progress through this chapter.

In this section, we discussed Docker Hub and the different steps for creating an account. Then, we looked at the steps for installing Docker Desktop locally, and finally, we finished with an overview of Docker elements.

We will now start working with Docker, and the first operation we will look at is the creation of a Docker image from a Dockerfile.

## Creating a Dockerfile

A basic Docker element is a file called a **Dockerfile**, which contains step-by-step instructions for building a Docker image.

To understand how to create a Dockerfile, we'll look at an example that allows us to build a Docker image that contains an Apache web server and a web application.

Let's start by writing a Dockerfile.

## Writing a Dockerfile

To write a Dockerfile, we will first create a **HyperText Markup Language (HTML)** page that will be our web application. So, we'll create a new `appdocker` directory and an `index.html` page in it, which includes the example code that displays welcome text on a web page, as follows:

```
<html>
  <body>
    <h1>Welcome to my new app</h1>
    This page is test for my demo Dockerfile.<br />
    Enjoy ...
  </body>
</html>
```

Then, in the same directory, we create a Dockerfile (without an extension) with the following content, which we will detail right after:

```
FROM httpd:latest
COPY index.html /usr/local/apache2/htdocs/
```

To create a Dockerfile, start with the `FROM` statement. The required `FROM` statement defines the base image, which we will use for our Docker image—any Docker image is built from another Docker image. This base image can be saved either in Docker Hub or in another registry, such as JFrog Artifactory, Nexus Repository, or ACR.

In our code example, we use the Apache `httpd` image tagged as the latest version, `https://hub.docker.com/_/httpd/`, and we use the `FROM httpd:latest` Dockerfile instruction.

Then, we use the `COPY` instruction to execute the image construction process. Docker copies the local `index.html` file that we just created into the `/usr/local/apache2/htdocs/` directory of the image.

#### Note

The source code for this Dockerfile and the HTML page can be found here: <https://github.com/PacktPublishing/Learning-DevOps-Second-Edition/tree/main/CHAP09/appdocker>.

We have just looked at the `FROM` and `COPY` instructions of the Dockerfile, but there are other instructions as well that we'll cover in the following section.

## Dockerfile instructions overview

We previously mentioned that a Dockerfile file is comprised of instructions, and we also looked at a concrete example with the `FROM` and `COPY` instructions. There are other instructions that will allow you to build a Docker image. Here is an overview of the principal instructions that can be used for this purpose:

- **FROM:** This instruction is used to define the base image for our image, as shown in the example detailed in the preceding *Writing a Dockerfile* section.
- **COPY and ADD:** These are used to copy one or more local files into an image. The `ADD` instruction supports an extra two functionalities, to refer to a **Uniform Resource Locator (URL)** and to extract compressed files.

**Note**

For more details about the differences between COPY and ADD, you can read this article: <https://nickjanetakis.com/blog/docker-tip-2-the-difference-between-copy-and-add-in-a-dockerfile>.

- **RUN and CMD:** These instructions take a command as a parameter that will be executed during the construction of the image. The RUN instruction creates a layer so that it can be cached and versioned. The CMD instruction defines a default command to be executed during the call to run the image. The CMD instruction can be overwritten at runtime with an extra parameter provided.

You can write the following example of the RUN instruction in a Dockerfile to execute the `apt-get` command:

```
RUN apt-get update
```

With the preceding instruction, we update the `apt` packages that are already present in the image and create a layer. We can also use the CMD instruction in the following example, which will display a `docker` message during execution:

```
CMD "echo docker"
```

- **ENV:** This instruction allows you to instantiate environment variables that can be used to build an image. These environment variables will persist throughout the life of the container, as follows:

```
ENV myvar=mykey
```

The preceding command sets a `myvar` environment variable with the `mykey` value to the container.

- **WORKDIR:** This instruction gives the execution directory of the container, as follows:

```
WORKDIR usr/local/apache2
```

That was an overview of Dockerfile instructions. There are other instructions that are commonly used, such as EXPOSE, ENTRYPOINT, and VOLUME, which you can find in the official documentation at <https://docs.docker.com/engine/reference/builder/>.

We have just observed that the writing of a Dockerfile is performed with different instructions, such as `FROM`, `COPY`, and `RUN`, which are used to create a Docker image. Now, let's look at how to run Docker in order to build a Docker image from a Dockerfile, and run that image locally to test it.

## Building and running a container on a local machine

So far in the chapter, we have discussed Docker elements and have looked at an example of a Dockerfile that is used to containerize a web application. Now, we have all the elements to run Docker.

The execution of Docker is performed by different operations, as outlined here:

- Building a Docker image from a Dockerfile
- Instantiating a new container locally from this image
- Testing our locally containerized application

Let's take a deep dive into each operation.

### Building a Docker image

We'll build a Docker image from our previously created Dockerfile that contains the following instructions:

```
FROM httpd:latest
COPY index.html /usr/local/apache2/htdocs/
```

We'll go to a terminal to head into the directory that contains the Dockerfile, and then execute the `docker build` command with the following syntax:

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

The `-t` argument indicates the name of the image and its tag. Here, in our example, we call our image `demobook`, and the tag we've added is `v1`.

The . (dot) at the end of the command specifies that we will use the files in the current directory. The following screenshot shows the execution of this command:

```

PS C:\Users\...\.Learning-DevOps-Second-Edition\CHAP09\apddocker> docker build -t demobook:v1 .
[+] Building 40.5s (7/7) FINISHED
=> [internal] load build definition from Dockerfile                                0.3s
=> => transferring dockerfile: 98B                                                0.1s
=> [internal] load .dockerignore                                                  0.1s
=> => transferring context: 28                                                    0.0s
=> [internal] load metadata for docker.io/library/httpd:latest                  5.6s
=> [internal] load build context                                                 0.1s
=> => transferring context: 191B                                                  0.0s
=> [1/2] FROM docker.io/library/httpd:latest@sha256:f70876d78442771406d7245b8d3425e8b0a86891c79811af94fb2e12af0fadeb 33.9s
=> => resolve docker.io/library/httpd:latest@sha256:f70876d78442771406d7245b8d3425e8b0a86891c79811af94fb2e12af0fadeb 0.1s
=> => sha256:4482565671564bb0b369534aa4040f113c5fe4eee6aable2da04d144f663eed4 913.73kB / 913.73kB 0.6s
=> => sha256:f70876d78442771406d7245b8d3425e8b0a86891c79811af94fb2e12af0fadeb 1.86kB / 1.86kB 0.0s
=> => sha256:73c9b78280a693058838e9e3519e7f5723d742ada3e42c45f10744b4d88f486e 1.36kB / 1.36kB 0.0s
=> => sha256:1132a4fc88faaf5c19959f03535c1356d3004ced1978cb9c3f32e73d9c139532 8.78kB / 8.78kB 0.0s
=> => sha256:7d63c13d9b9b6ec5f05a2b07daadacaa9c610d01102a662ae9b1d082105f1ffa 31.36MB / 31.36MB 17.4s
=> => sha256:ca52f3eeea665ce537eec1840e21d7d024ab70fb555a609cd748e710779db9e0 176B / 176B 0.5s
=> => sha256:21d69ac90caf9d24441bfa860ed24c4bf82e421f95d9a2abf957c9b111978c03 24.11MB / 24.11MB 15.3s
=> => sha256:462e88bc307455be86d7af71d19421a240793468d7ab879e36c86b54d8e0ec7d 296B / 296B 0.8s
=> => extracting sha256:7d63c13d9b9b6ec5f05a2b07daadacaa9c610d01102a662ae9b1d082105f1ffa 6.0s
=> => extracting sha256:ca52f3eeea665ce537eec1840e21d7d024ab70fb555a609cd748e710779db9e0 0.0s
=> => extracting sha256:4482565671564bb0b369534aa4040f113c5fe4eee6aable2da04d144f663eed4 0.4s
=> => extracting sha256:21d69ac90caf9d24441bfa860ed24c4bf82e421f95d9a2abf957c9b111978c03 4.3s
=> => extracting sha256:462e88bc307455be86d7af71d19421a240793468d7ab879e36c86b54d8e0ec7d 0.0s
=> [2/2] COPY index.html /usr/local/apache2/htdocs/ 0.1s
=> => exporting to image 0.1s
=> => exporting layers 0.1s
=> => writing image sha256:9a3862a66c65d0a431b70b464d8deedff1a73927fad7475ac520f876644c3301 0.0s
=> => naming to docker.io/library/demobook:v1 0.0s
  
```

Figure 9.9 – docker build command

We can see in this preceding execution the three steps of the Docker image builder, as follows:

1. Docker downloads the defined base image.
2. Docker copies the `index.html` file in the image.
3. Docker creates and tags the image.

When you execute the `docker build` command, it downloads the base image indicated in the Dockerfile from Docker Hub, and then Docker executes the various instructions that are mentioned in the Dockerfile.

#### Note

Note that if during the first execution of the `docker build` command you get a `Get https://registry-1.docker.io/v2/library/httpd/manifests/latest: unauthorized: incorrect username or password error`, then execute the `docker logout` command. Next, restart the `docker build` command, as indicated in this article: <https://medium.com/@blacksourcez/fix-docker-error-unauthorized-incorrect-username-or-password-in-docker-f80c45951b6b>.

At the end of the execution, we obtain a locally stored Docker `demobook` image.

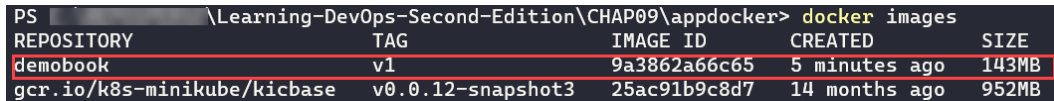
**Note**

The Docker image is stored in a local folder system depending on your OS. For more information about the location of Docker images, you can read this article: <http://www.scmgalaxy.com/tutorials/location-of-dockers-images-in-all-operating-systems/>.

We can also check if the image is successfully created by executing the following Docker command:

```
docker images
```

Here is the output of the preceding command:



REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
demobook	v1	9a3862a66c65	5 minutes ago	143MB
gcr.io/k8s-minikube/kicbase	v0.0.12-snapshot3	25ac91b9c8d7	14 months ago	952MB

Figure 9.10 – docker images command

This command displays a list of Docker images on my local machine, and we can see the demobook image we just created. So, the next time the image is built, we will not need to download the httpd image again.

Now that we have created a Docker image of our application, we will instantiate a new container of this image.

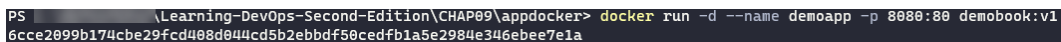
## Instantiating a new container of an image

To instantiate a new container of our Docker image, we will execute the `docker run` command in our Terminal, with the following syntax:

```
docker run -d --name demoapp -p 8080:80 demobook:v1
```

The `-d` parameter indicates that the container will run in the background. In the `--name` parameter, we specify the name of the container we want. In the `-p` parameter, we indicate the desired port translation. In our example, this would mean port 80 of the container will be translated to port 8080 on our local machine. And finally, the last parameter of the command is the name of the image and its tag.

The execution of this command is shown in the following screenshot:



```
PS \Learning-DevOps-Second-Edition\CHAP09\appdocker> docker run -d --name demoapp -p 8080:80 demobook:v1
6cce2099b174cbe29fcd408d044cd5b2ebddf50cedfb1a5e2984e346ebee7e1a
```

Figure 9.11 – docker run command

At the end of its execution, this command displays the ID of the container, and the container runs in the background. It is also possible to display a list of containers running on the local machine by executing the following command:

```
docker ps
```

The following screenshot shows the execution with our container:

```
PS C:\Users\mkrief> docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
6cce2099b174	demobook:v1	"httpd-foreground"	2 weeks ago	Up About a minute	0.0.0.0:8080->80/tcp	demoapp

Figure 9.12 – docker ps command

After the execution of each container, we have its shortcut ID, its associated image, its name, its execution command, and its translation port information displayed.

So, we have built a Docker image and instantiated a new container of that image locally. We will now see how to run a web application that is in the local container.

## Testing a container locally

*Everything that runs in a container remains inside it*—this is the principle of container isolation. However, in the port translation that we did previously, you can test your container on your local machine with the run command.

To do this, open a web browser and enter `http://localhost:8080` with 8080, which represents the translation port indicated in the command. You should be able to see the following result:

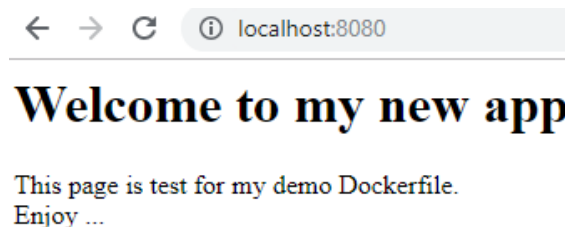


Figure 9.13 – Docker application launched

We can see the content of our `index.html` page displayed.

In this section, we looked at the different Docker commands that can be used to build a Docker image. Then, we instantiated a new container from that image, and finally, we tested it locally.

In the next section, we will see how to publish a Docker image in Docker Hub.



## Pushing an image to Docker Hub

The goal of creating a Docker image that contains an application is to be able to use it on servers that contain Docker and host the company's applications, just as with a VM.

In order for an image to be downloaded to another computer, it must be saved in a Docker image registry. As already mentioned in this chapter, there are several Docker registries that can be installed on-premises, which is the case for JFrog Artifactory and Nexus Repository.

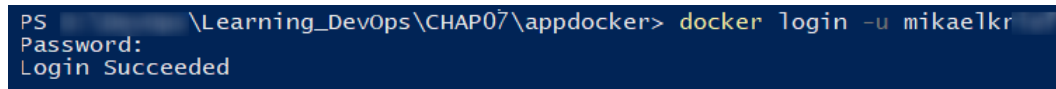
If you want to create a public image, you can push (or upload) it to Docker Hub, which is Docker's public (and free, depending on your license) registry. We will now see how to upload the image we created in the previous section to Docker Hub. To do this, you need to have an account on Docker Hub, which we created prior to installing Docker Desktop.

To push a Docker image to Docker Hub, perform the following steps:

1. **Sign in to Docker Hub:** Log in to Docker Hub using the following command:

```
docker login -u <your dockerhub login>
```

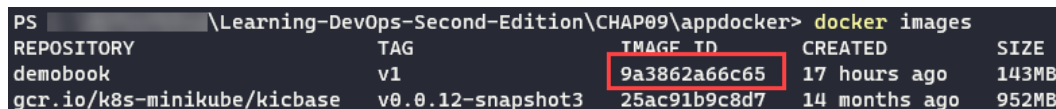
When executing the command, you will be asked to enter your Docker Hub password and indicate that you are connected to the Docker registry, as shown in the following screenshot:



```
PS \Learning_DevOps\CHAP07\appdocker> docker login -u mikaelkr
Password:
Login Succeeded
```

Figure 9.14 – The docker login command

2. **Retrieve the image ID:** The next step consists of retrieving the ID of the image that has been created. To do so, we will execute the `docker images` command to display a list of images with their ID.



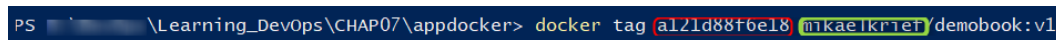
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
demobook	v1	9a3862a66c65	17 hours ago	143MB
gcr.io/k8s-minikube/kicbase	v0.0.12-snapshot3	25ac91b9c8d7	14 months ago	952MB

Figure 9.15 – Docker images list

3. **Tag the image for Docker Hub:** With the ID of the image we retrieved, we will now tag the image for Docker Hub. To do so, the following command is executed:

```
docker tag <image ID> <dockerhub login>/demobook:v1
```

The following screenshot shows the execution of this command on the created image:



```
PS \Learning_DevOps\CHAP07\appdocker> docker tag a121d88f6e18 mikaelkr/demobook:v1
```

Figure 9.16 – docker tag

4. **Push the Docker image to Docker Hub:** After tagging the image, the last step is to push the tagged image to Docker Hub.

For this purpose, we will execute the following command:

```
docker push docker.io/<dockerhub login>/demobook:v1
```

The following screenshot shows the execution of the preceding command:

```
PS > (Learning-DevOps-Second-Edition\CHAP09\appdocker> docker push docker.io/mikaelkrief/demobook:v1
The push refers to repository [docker.io/mikaelkrief/demobook]
f083a28f9cfa: Pushed
4dcdec0b7a0e: Mounted from library/httpd
c86537ee54f9: Mounted from library/httpd
ecd2b49ef243: Mounted from library/httpd
7511c367f47a: Mounted from library/httpd
e8b689711f21: Mounted from library/httpd
v1: digest: sha256:380d9e0b2beb20c495b496c7047bd3d808f048307a5b5c84cc1e1de3fe119e79 size: 1572
```

Figure 9.17 – docker push image

We can see from this execution that the image is uploaded to Docker Hub.

To view the pushed image in Docker Hub, we connect to the Docker Hub web portal at <https://hub.docker.com/> and see that the image is present, as shown in the following screenshot:

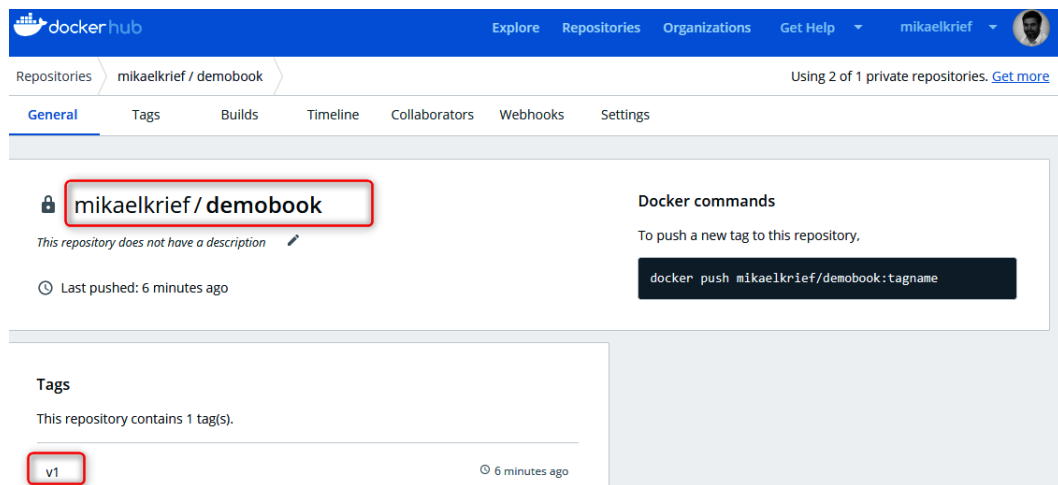


Figure 9.18 – The pushed image in Docker Hub with tag

By default, the image pushed to Docker Hub is in public mode—everybody can view it in the explorer and use it.

We can access this image in Docker Hub in the Docker Hub search engine, as shown in the following screenshot:

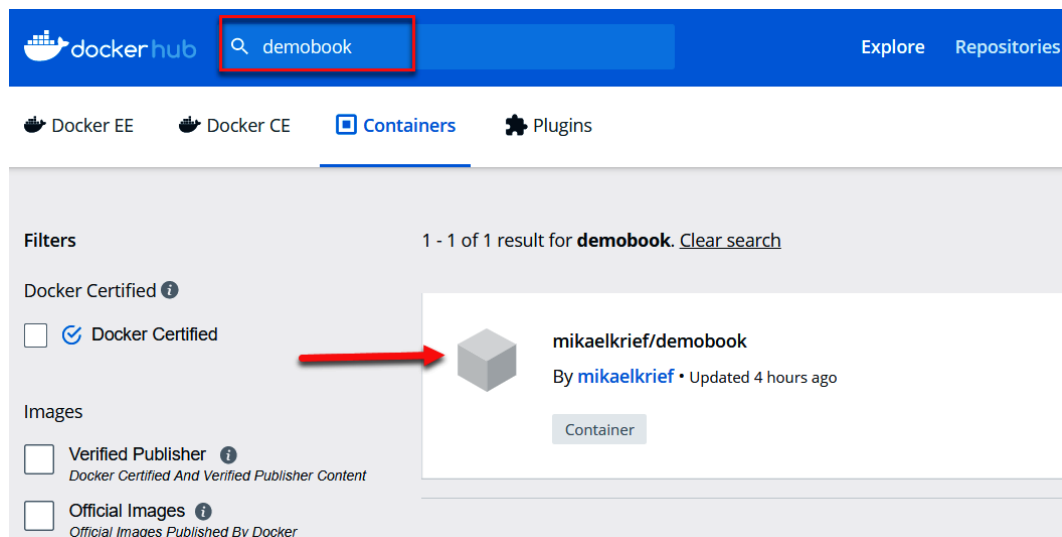


Figure 9.19 – Finding the image in Docker Hub

To make this image private—meaning only you are authenticated to use it—you must go to the **Settings** tab of the image and click on the **Make private** button, as shown in the following screenshot:

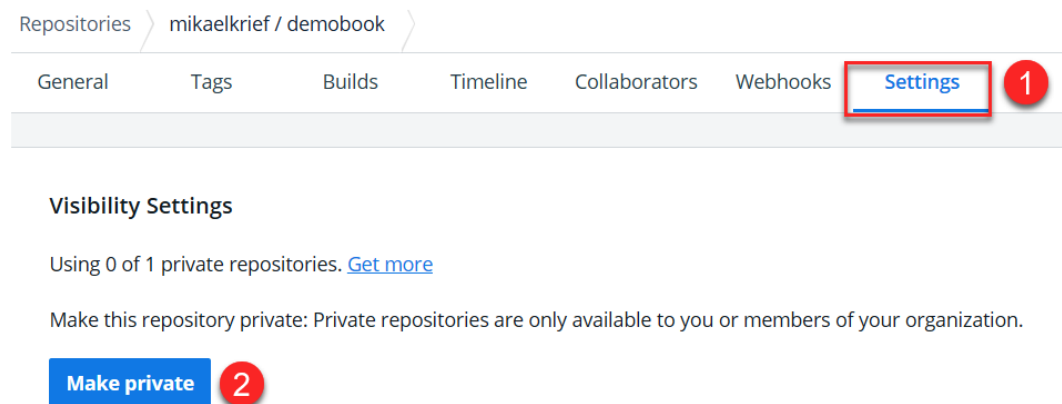


Figure 9.20 – Making a Docker image private

In this section, we looked at the steps and Docker commands for logging in to Docker Hub via the command line, and then we looked at the `tag` and `push` commands for uploading a Docker image to Docker Hub.

In the next section, we will see how to push a Docker image to a private Docker registry using an example ACR instance.