FEBRUARY 1, 2021 / #DOCKER

The Docker Handbook – Learn Docker for Beginners

Farhan Hasin Chowdhury

The concept of containerization itself is pretty old. But the emergence of the Docker Engine in 2013 has made it much easier to containerize your applications.

According to the Stack Overflow Developer Survey - 2020, Docker is the #1 most wanted platform, #2 most loved platform, and also the #3 most popular platform.

As in-demand as it may be, getting started can seem a bit intimidating at first. So in this book, we'll be learning everything from the basics to a more intermediate level of containerization. After going through the entire book, you should be able to:

- Containerize (almost) any application
- Upload custom Docker Images to online registries
- Work with multiple containers using Docker Compose

Prerequisites

JavaScript)

Table of Contents

- Introduction to Containerization and Docker
- How to Install Docker
  - How to Install Docker on macOS
  - How to Install Docker on Windows
  - How to Install Docker on Linux
- Hello World in Docker - Intro to Docker Basics
  - What is a Container?
  - What is a Docker Image?
  - What is a Docker Registry?
  - Docker Architecture Overview
  - The Full Picture
- Docker Container Manipulation Basics
  - How to Run a Container
  - How to Publish a Port
  - How to Use Detached Mode
  - How to List Containers
  - How to Name or Rename a Container
  - How to Stop or Kill a Running Container
  - How to Restart a Container
  - How to Create a Container Without Running

  - How to Execute Commands Inside a Container
  - How to Work With Executable Images
- Docker Image Manipulation Basics
  - How to Create a Docker Image
  - How to Tag Docker Images
  - How to List and Remove Docker Images
  - How to Understand the Many Layers of a Docker Image
  - How to Build NGINX from Source
  - How to Optimize Docker Images
  - Embracing Alpine Linux
  - How to Create Executable Docker Images
  - How to Share Your Docker Images Online
- How to Containerize a JavaScript Application
  - How to Write the Development Dockerfile
  - How to Work With Bind Mounts in Docker
  - How to Work With Anonymous Volumes in Docker
  - How to Perform Multi-Staged Builds in Docker
  - How to Ignore Unnecessary Files
- Network Manipulation Basics in Docker
  - Docker Network Basics
  - How to Create a User-Defined Bridge in Docker
  - How to Attach a Container to a Network in Docker
  - How to Detach Containers from a Network in Docker

Application

- How to Run the Database Server
- How to Work with Named Volumes in Docker
- How to Access Logs from a Container in Docker
- How to Create a Network and Attaching the Database Server in Docker
- How to Write the Dockerfile
- How to Execute Commands in a Running Container
- How to Write Management Scripts in Docker

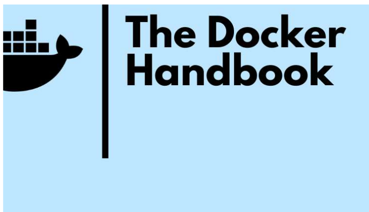
- How to Compose Projects Using Docker-Compose
  - Docker Compose Basics
  - How to Start Services in Docker Compose
  - How to List Services in Docker Compose
  - How to Execute Commands Inside a Running Service in Docker Compose
  - How to Access Logs from a Running Service in Docker Compose
  - How to Stop Services in Docker Compose
  - How to Compose a Full-stack Application in Docker Compose
- Conclusion

Project Code

Code for the example projects can be found in the following repository:

Project codes used in “The Docker Handbook” :notebook: - fhsinchy/docker-handbook-projects

fhsinchy • GitHub



spare a ⭐ to keep me motivated

You can find the complete code in the `completed` branch.

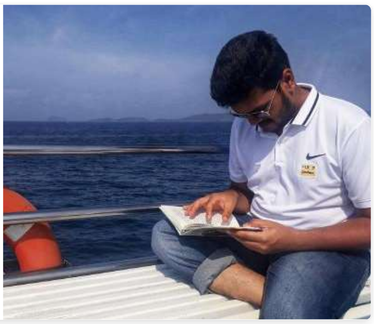
Contributions

This book is completely open-source and quality contributions are more than welcome. You can find the full content in the following repository:

fhsinchy/the-docker-handbook

Open-source book on Docker. Contribute to fhsinchy/the-docker-handbook development by…

fhsinchy • GitHub



spare a ⭐ to keep me motivated

I usually do my changes and updates on the GitBook version of the book first and then publish them on freeCodeCamp. You can find the always updated and often unstable version of the book at the following link:

The Docker Handbook

'he Docker Handbook

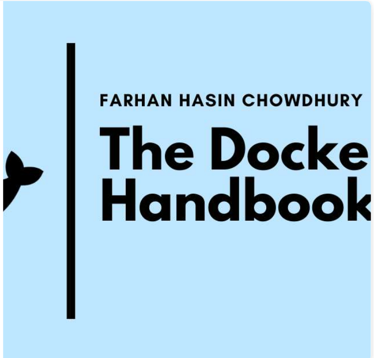
don't forget to leave ⭐ on the repository

If you're looking for a frozen but stable version of the book, then freeCodeCamp will be the best place to go:

The Docker Handbook

The concept of containerization itself is pretty old, but the emergence of the Docker Engine...

favicon Farhan Hasin Chowdhury • freeCodeCamp.org



sharing with others may help

Whichever version of the book you end up reading though, don't forget to let me know your opinion. Constructive criticism is always welcomed.

Introduction to Containerization and Docker

According to IBM,

> Containerization involves encapsulating or packaging up software code and all its dependencies so that it can run uniformly and consistently on any infrastructure.

In other words, containerization lets you bundle up your software along with all its dependencies in a self-contained package so that it an awesome book management application that can store information regarding all the books you own, and can also serve the purpose of a book lending system for your friends.

If you make a list of the dependencies, that list may look as follows:

- Node.js
- Express.js
- SQLite3

Well, theoretically this should be it. But practically there are some other things as well. Turns out Node.js uses a build tool known as `node-gyp` for building native add-ons. And according to the installation instruction in the official repository, this build tool requires Python 2 or 3 and a proper C/C++ compiler tool-chain.

Taking all these into account, the final list of dependencies is as follows:

- Node.js
- Express.js
- SQLite3
- Python 2 or 3
- C/C++ tool-chain

Installing Python 2 or 3 is pretty straightforward regardless of the platform you're on. Setting up the C/C++ tool-chain is pretty easy on Linux, but on Windows and Mac it's a painful task.

Tools for Xcode package.

Regardless of the one you install, it still may break on OS updates. In fact, the problem is so prevalent that there are Installation notes for macOS Catalina available on the official repository.

Let's assume that you've gone through all the hassle of setting up the dependencies and have started working on the project. Does that mean you're out of danger now? Of course not.

What if you have a teammate who uses Windows while you're using Linux. Now you have to consider the inconsistencies of how these two different operating systems handle paths. Or the fact that popular technologies like nginx are not well optimized to run on Windows. Some technologies like Redis don't even come pre-built for Windows.

Even if you get through the entire development phase, what if the person responsible for managing the servers follows the wrong deployment procedure?

All these issues can be solved if only you could somehow:

- Develop and run the application inside an isolated environment (known as a container) that matches your final deployment environment.
- Put your application inside a single file (known as an image) along with all its dependencies and necessary deployment configurations.
- And share that image through a central server (known as a registry) that is accessible by anyone with proper

Your teammates will then be able to download the image from the registry, run the application as it is within an isolated environment free from the platform specific inconsistencies, or even deploy directly on a server, since the image comes with all the proper production configurations.

That is the idea behind containerization: putting your applications inside a self-contained package, making it portable and reproducible across various environments.

Now the question is "What role does Docker play here?"

As I've already explained, containerization is an idea that solves a myriad of problems in software development by putting things into boxes.

This very idea has quite a few implementations. Docker is such an implementation. It's an open-source containerization platform that allows you to containerize your applications, share them using public or private registries, and also to orchestrate them.

Now, Docker is not the only containerization tool on the market, it's just the most popular one. Another containerization engine that I love is called Podman developed by Red Hat. Other tools like Kaniko by Google, rkt by CoreOS are amazing, but they're not ready to be a drop-in replacement for Docker just yet.

Also, if you want a history lesson, you may read the amazing A Brief History of Containers: From the 1970s Till Now which covers most of the major turning points for the technology.

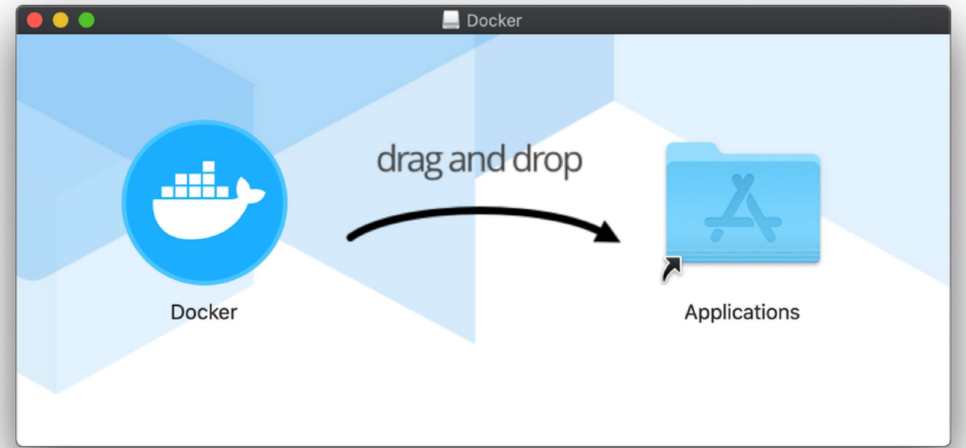
system you’re using. But it's universally simple across the board.

Docker runs flawlessly on all three major platforms, Mac, Windows, and Linux. Among the three, the installation process on Mac is the easiest, so we'll start there.

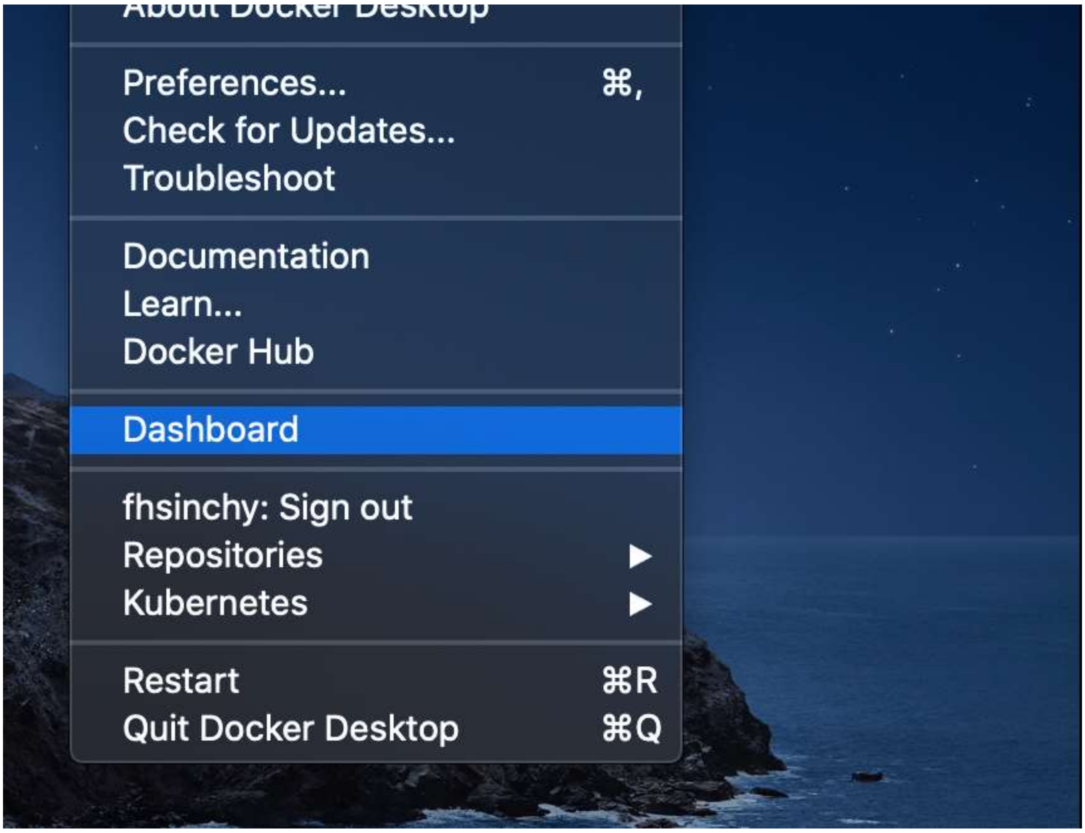
How to Install Docker on macOS

On a mac, all you have to do is navigate to the official download page and click the *Download for Mac (stable)* button.

You’ll get a regular looking *Apple Disk Image* file and inside the file, there will be the application. All you have to do is drag the file and drop it in your Applications directory.



You can start Docker by simply double-clicking the application icon. Once the application starts, you'll see the Docker icon appear on your menu-bar.



Now, open up the terminal and execute `docker --version` and `docker-compose --version` to ensure the success of the installation.

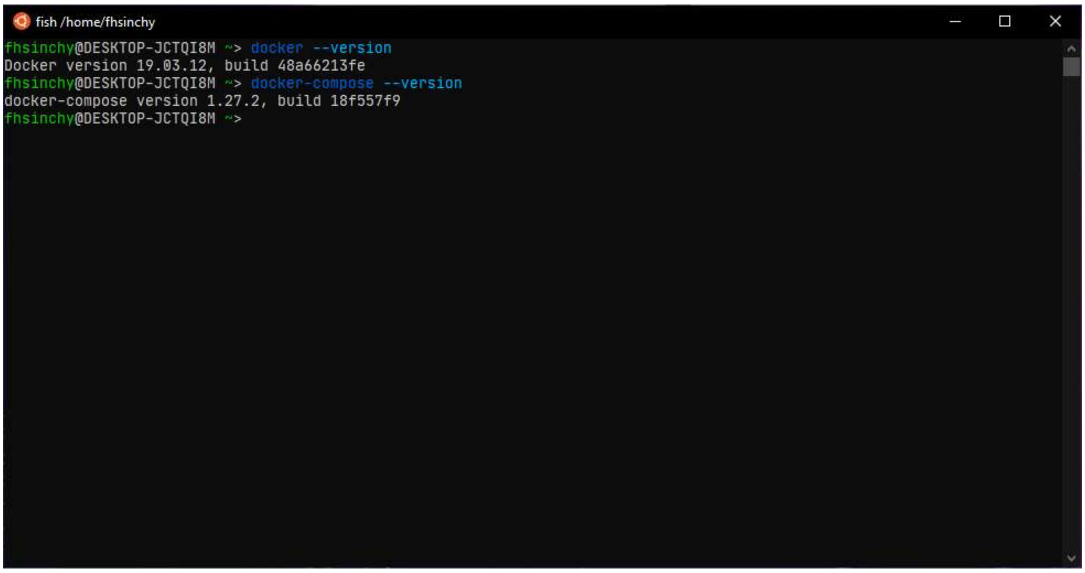
How to Install Docker on Windows

On Windows, the procedure is almost the same, except there are a few extra steps that you'll need to go through. The installation steps are as follows:

1. Navigate to this site and follow the instructions for installing WSL2 on Windows 10.
2. Then navigate to the official download page and click the *Download for Windows (stable)* button.
3. Double-click the downloaded installer and go through the installation with the defaults.

start menu or your desktop. The docker icon should show up on your taskbar.

Now, open up Ubuntu or whatever distribution you've installed from Microsoft Store. Execute the `docker --version` and `docker-compose --version` commands to make sure that the installation was successful.



You can access Docker from your regular Command Prompt or PowerShell as well. It's just that I prefer using WSL2 over any other command line on Windows.

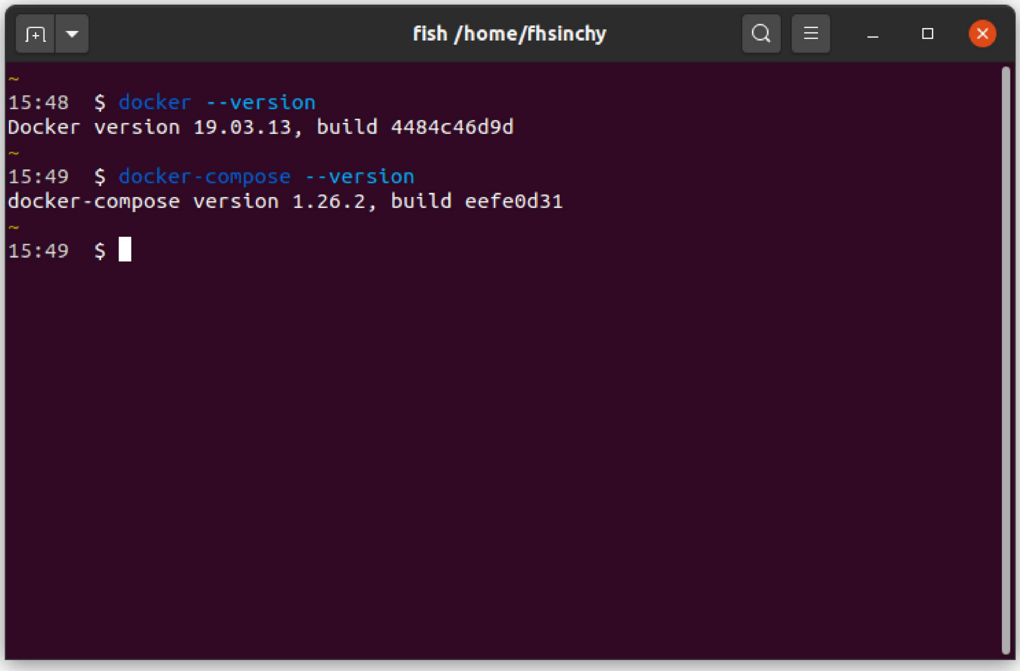
depending on the distribution you're on, it may vary even more. But to be honest, the installation is just as easy (if not easier) as the other two platforms.

The Docker Desktop package on Windows or Mac is a collection of tools like `Docker Engine`, `Docker Compose`, `Docker Dashboard`, `Kubernetes` and a few other goodies.

On Linux however, you don't get such a bundle. Instead you install all the necessary tools you need manually. Installation procedures for different distributions are as follows:

- If you're on Ubuntu, you may follow the Install Docker Engine on Ubuntu section from the official docs.
- For other distributions, *installation per distro* guides are available on the official docs.
    - Install Docker Engine on Debian
    - Install Docker Engine on Fedora
    - Install Docker Engine on CentOS
- If you're on a distribution that is not listed in the docs, you may follow the Install Docker Engine from binaries guide instead.
- Regardless of the procedure you follow, you'll have to go through some Post-installation steps for Linux which are very important.
- Once you're done with the docker installation, you'll have to install another tool named Docker Compose. You may follow the Install Docker Compose guide from the official docs.

success of the installation.



Although Docker performs quite well regardless of the platform you're on, I prefer Linux over the others. Throughout the book, I'll be switching between my Ubuntu 20.10 and Fedora 33 workstations.

Another thing that I would like to clarify right from the get go, is that I won't be using any GUI tool for working with Docker throughout the entire book.

I'm aware of the nice GUI tools available for different platforms, but learning the common docker commands is one of the primary goals of this book.

Now that you have Docker up and running on your machine, it's time for you to run your first container. Open up the terminal and run the following command:

```
docker run hello-world

# Unable to find image 'hello-world:latest' locally
# latest: Pulling from library/hello-world
# 0e03bdcc26d7: Pull complete
# Digest: sha256:4cf9c47f86df71d48364001ede3a4fcd85ae80ce02ebad7
# Status: Downloaded newer image for hello-world:latest
#
# Hello from Docker!
# This message shows that your installation appears to be workin
#
# To generate this message, Docker took the following steps:
#  1. The Docker client contacted the Docker daemon.
#  2. The Docker daemon pulled the "hello-world" image from the 
#     (amd64)
#  3. The Docker daemon created a new container from that image 
#     executable that produces the output you are currently read
#  4. The Docker daemon streamed that output to the Docker clien
#     to your terminal.
#
# To try something more ambitious, you can run an Ubuntu contain
#  $ docker run -it ubuntu bash
#
# Share images, automate workflows, and more with a free Docker 
#  https://hub.docker.com/
#
# For more examples and ideas, visit:
#  https://docs.docker.com/get-started/
```

The hello-world image is an example of minimal containerization with Docker. It has a single program compiled from a hello.c file responsible for printing out the message you're seeing on your terminal.

run in the past:

```
docker ps -a

# CONTAINER ID        IMAGE               COMMAND             CRI
# 128ec8ceab71        hello-world         "/hello"            14
```

In the output, a container named `exciting_chebyshev` was run with the container id of `128ec8ceab71` using the `hello-world` image. It has `Exited (0) 13 seconds ago` where the `(0)` exit code means no error was produced during the runtime of the container.

Now in order to understand what just happened behind the scenes, you'll have to get familiar with the Docker Architecture and three very fundamental concepts of containerization in general, which are as follows:

- Container
- Image
- Registry

I've listed the three concepts in alphabetical order and will begin my explanations with the first one on the list.

What is a Container?

In the world of containerization, there can not be anything more fundamental than the concept of a container.

The official Docker resources site says -

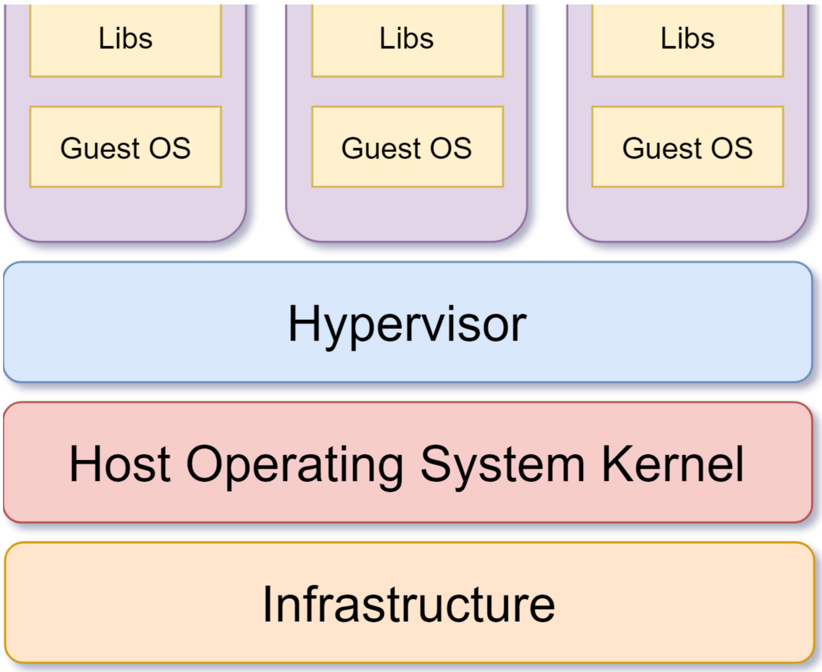
> virtualize the host operating system only.

You may consider containers to be the next generation of virtual machines.

Just like virtual machines, containers are completely isolated environments from the host system as well as from each other. They are also a lot lighter than the traditional virtual machine, so a large number of containers can be run simultaneously without affecting the performance of the host system.

Containers and virtual machines are actually different ways of virtualizing your physical hardware. The main difference between these two is the method of virtualization.

Virtual machines are usually created and managed by a program known as a hypervisor, like Oracle VM VirtualBox, VMware Workstation, KVM, Microsoft Hyper-V and so on. This hypervisor program usually sits between the host operating system and the virtual machines to act as a medium of communication.

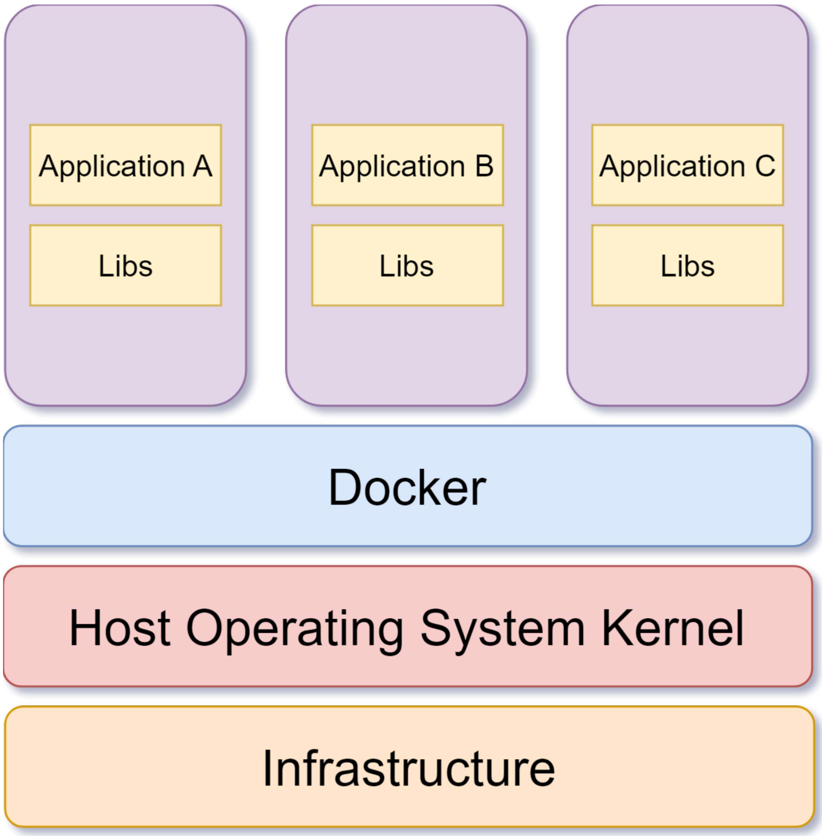


Each virtual machine comes with its own guest operating system which is just as heavy as the host operating system.

The application running inside a virtual machine communicates with the guest operating system, which talks to the hypervisor, which then in turn talks to the host operating system to allocate necessary resources from the physical infrastructure to the running application.

As you can see, there is a long chain of communication between applications running inside virtual machines and the physical infrastructure. The application running inside the virtual machine may take only a small amount of resources, but the guest operating system adds a noticeable overhead.

Unlike a virtual machine, a container does the job of virtualization in a smarter way. Instead of having a complete guest operating system inside a container, it just utilizes the host operating system via the



The container runtime, that is Docker, sits between the containers and the host operating system instead of a hypervisor. The containers then communicate with the container runtime which then communicates with the host operating system to get necessary resources from the physical infrastructure.

As a result of eliminating the entire guest operating system layer, containers are much lighter and less resource-hogging than traditional virtual machines.

As a demonstration of the point, look at the following code block:

In the code block above, I have executed the `uname -a` command on my host operating system to print out the kernel details. Then on the next line I've executed the same command inside a container running Alpine Linux.

As you can see in the output, the container is indeed using the kernel from my host operating system. This goes to prove the point that containers virtualize the host operating system instead of having an operating system of their own.

If you're on a Windows machine, you'll find out that all the containers use the WSL2 kernel. It happens because WSL2 acts as the back-end for Docker on Windows. On macOS the default back-end is a VM running on HyperKit hypervisor.

What is a Docker Image?

Images are multi-layered self-contained files that act as the template for creating containers. They are like a frozen, read-only copy of a container. Images can be exchanged through registries.

In the past, different container engines had different image formats. But later on, the Open Container Initiative (OCI) defined a standard specification for container images which is complied by the major containerization engines out there. This means that an image built with Docker can be used with another runtime like Podman without any additional hassle.

Containers are just images in running state. When you obtain an image from the internet and run a container using that image, you

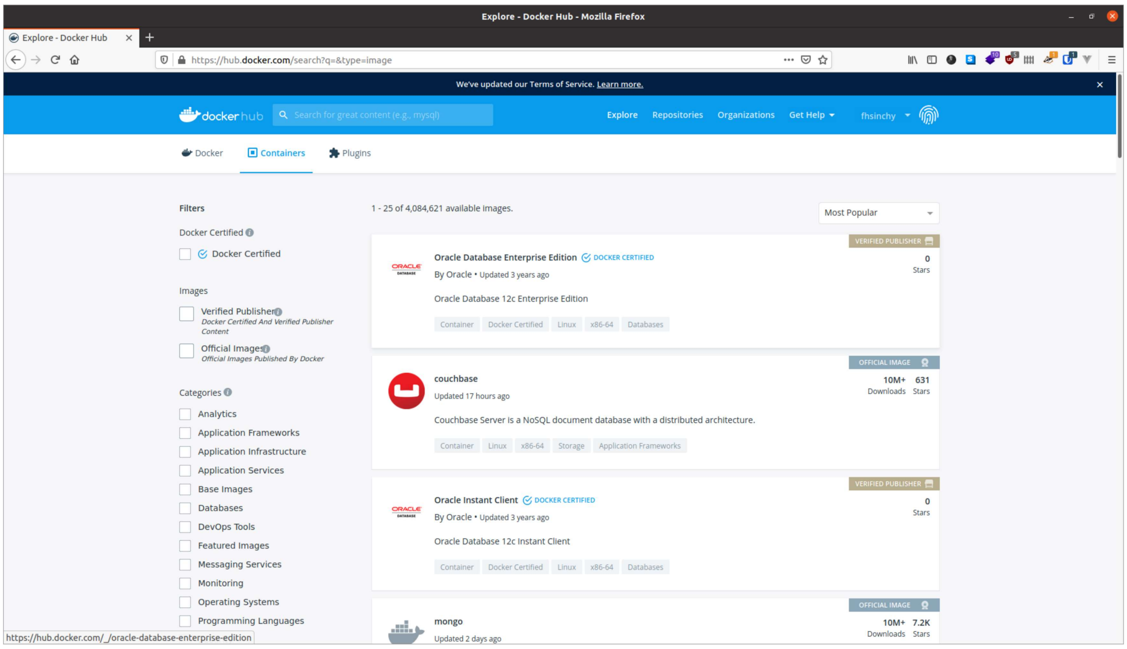
This concept will become a lot clearer in upcoming sections of this book. But for now, just keep in mind that images are multi-layered read-only files carrying your application in a desired state inside them.

What is a Docker Registry?

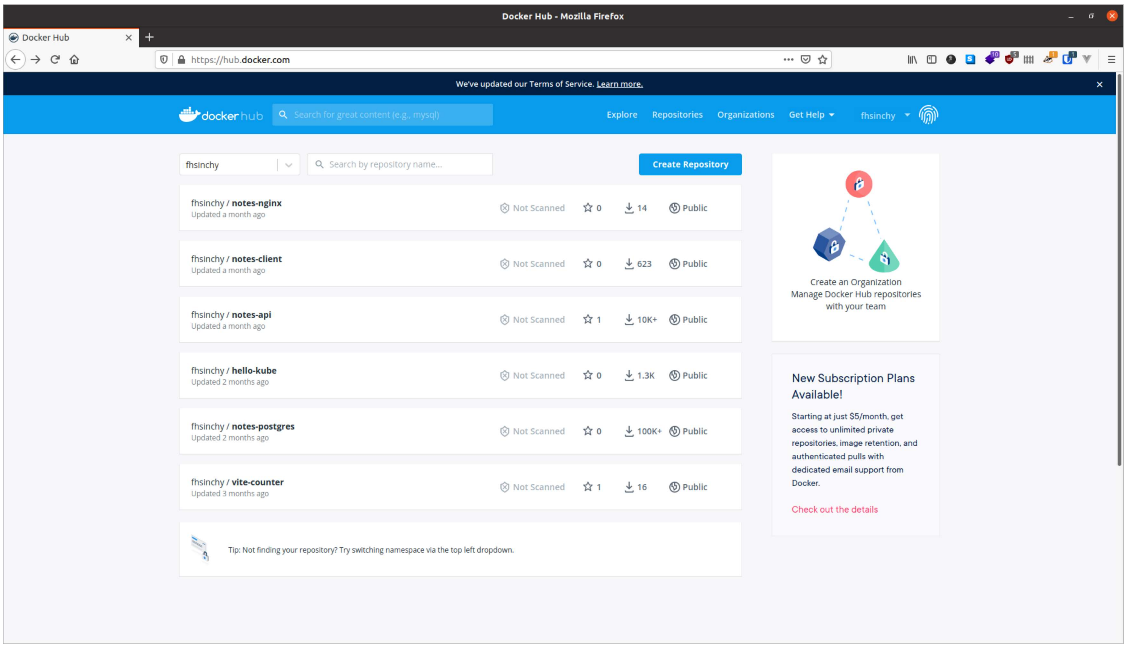
You've already learned about two very important pieces of the puzzle, *Containers* and *Images*. The final piece is the *Registry*.

An image registry is a centralized place where you can upload your images and can also download images created by others. Docker Hub is the default public registry for Docker. Another very popular image registry is Quay by Red Hat.

Throughout this book I'll be using Docker Hub as my registry of choice.



(fhsinchy) page.



Apart from Docker Hub or Quay, you can also create your own image registry for hosting private images. There is also a local registry that runs within your computer that caches images pulled from remote registries.

Docker Architecture Overview

Now that you've become familiar with most of the fundamental concepts regarding containerization and Docker, it's time for you to understand how Docker as a software was designed.

The engine consists of three major components:

1. **Docker Daemon:** The daemon (`dockerd`) is a process that keeps running in the background and waits for commands

2. **Docker Client:** The client (`docker`) is a command-line interface program mostly responsible for transporting commands issued by users.
3. **REST API:** The REST API acts as a bridge between the daemon and the client. Any command issued using the client passes through the API to finally reach the daemon.

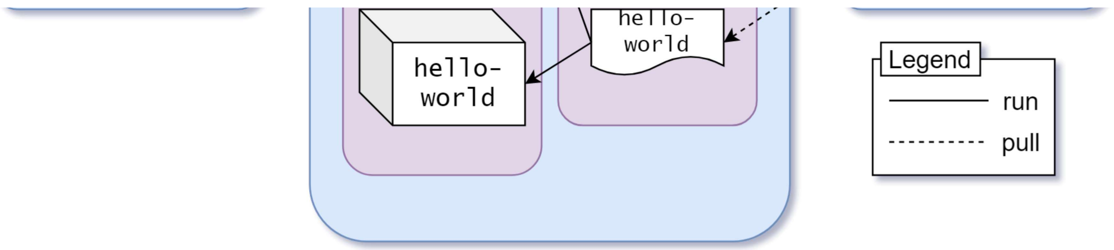
According to the official docs,

> "Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers".

You as a user will usually execute commands using the client component. The client then use the REST API to reach out to the long running daemon and get your work done.

The Full Picture

Okay, enough talking. Now it's time for you to understand how all these pieces of the puzzle you just learned about work in harmony. Before I dive into the explanation of what really happens when you run the `docker run hello-world` command, let me show you a little diagram I've made:



This image is a slightly modified version of the one found in the official docs. The events that occur when you execute the command are as follows:

1. You execute `docker run hello-world` command where `hello-world` is the name of an image.
2. Docker client reaches out to the daemon, tells it to get the `hello-world` image and run a container from that.
3. Docker daemon looks for the image within your local repository and realizes that it's not there, resulting in the `Unable to find image 'hello-world:latest' locally` that's printed on your terminal.
4. The daemon then reaches out to the default public registry which is Docker Hub and pulls in the latest copy of the `hello-world` image, indicated by the `latest: Pulling from library/hello-world` line in your terminal.
5. Docker daemon then creates a new container from the freshly pulled image.
6. Finally Docker daemon runs the container created using the `hello-world` image outputting the wall of text on your terminal.

It's the default behavior of Docker daemon to look for images in the hub that are not present locally. But once an image has been

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:d58e752213a51785838f9eed2b7a498ffa1cb3aa7f946dda1
Status: Downloaded newer image for hello-world:latest
```

If there is a newer version of the image available on the public registry, the daemon will fetch the image again. That `:latest` is a tag. Images usually have meaningful tags to indicate versions or builds. You'll learn about this in greater detail later on.

Docker Container Manipulation Basics

In the previous sections, you've learned about the building blocks of Docker and have also run a container using the `docker run` command.

In this section, you'll be learning about container manipulation in a lot more detail. Container manipulation is one of the most common task you'll be performing every single day, so having a proper understanding of the various commands is crucial.

Keep in mind, though, that this is not an exhaustive list of all the commands you can execute on Docker. I'll be talking only about the most common ones. Anytime you want to learn more about the available commands, just visit the official reference for the Docker command-line.

using the `hello-world` image. The generic syntax for this command is as follows:

```
docker run <image name>
```

Although this is a perfectly valid command, there is a better way of dispatching commands to the `docker` daemon.

Prior to version `1.13`, Docker had only the previously mentioned command syntax. Later on, the command-line was restructured to have the following syntax:

```
docker <object> <command> <options>
```

In this syntax:

- `object` indicates the type of Docker object you'll be manipulating. This can be a `container`, `image`, `network` or `volume` object.
- `command` indicates the task to be carried out by the daemon, that is the `run` command.
- `options` can be any valid parameter that can override the default behavior of the command, like the `--publish` option for port mapping.

Now, following this syntax, the `run` command can be written as follows:

The `image name` can be of any image from an online registry or your local system. As an example, you can try to run a container using the fhsinchy/hello-dock image. This image contains a simple Vue.js application that runs on port 80 inside the container.

To run a container using this image, execute following command on your terminal:

```
docker container run --publish 8080:80 fhsinchy/hello-dock

# /docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, wil
# /docker-entrypoint.sh: Looking for shell scripts in /docker-en
# /docker-entrypoint.sh: Launching /docker-entrypoint.d/10-liste
# 10-listen-on-ipv6-by-default.sh: Getting the checksum of /etc/
# 10-listen-on-ipv6-by-default.sh: Enabled listen on IPv6 in /et
# /docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsu
# /docker-entrypoint.sh: Configuration complete; ready for start
```

The command is pretty self-explanatory. The only portion that may require some explanation is the `--publish 8080:80` portion which will be explained in the next sub-section.

How to Publish a Port

Containers are isolated environments. Your host system doesn't know anything about what's going on inside a container. Hence, applications running inside a container remain inaccessible from the outside.

follows:

```
--publish <host port>:<container port>
```

When you wrote `--publish 8080:80` in the previous sub-section, it meant any request sent to port 8080 of your host system will be forwarded to port 80 inside the container.

Now to access the application on your browser, visit `http://127.0.0.1:8080`.

You can stop the container by simply hitting the `ctrl + c` key combination while the terminal window is in focus or closing off the terminal window completely.

or `-d` option. In the example above, in order for the container to keep running, you had to keep the terminal window open. Closing the terminal window also stopped the running container.

This is because, by default, containers run in the foreground and attach themselves to the terminal like any other normal program invoked from the terminal.

In order to override this behavior and keep a container running in background, you can include the `--detach` option with the `run` command as follows:

```
docker container run --detach --publish 8080:80 fhsinchy/hello-d

# 9f21cb77705810797c4b847dbd330d9c732ffddba14fb435470567a7a3f46c
```

Unlike the previous example, you won't get a wall of text thrown at you this time. Instead what you'll get is the ID of the newly created container.

The order of the options you provide doesn't really matter. If you put the `--publish` option before the `--detach` option, it'll work just the same. One thing that you have to keep in mind in case of the `run` command is that the image name must come last. If you put anything after the image name then that'll be passed as an argument to the container entry-point (explained in the Executing Commands Inside a Container sub-section) and may result in unexpected situations.

are currently running. To do so execute following command:

```
docker container ls

# CONTAINER ID        IMAGE                 COMMAND
# 9f21cb777058        fhsinchy/hello-dock   "/docker-entrypoint.
```

A container named `gifted_sammet` is running. It was created `5 seconds ago` and the status is `Up 5 seconds,` which indicates that the container has been running fine since its creation.

The `CONTAINER ID` is `9f21cb777058` which is the first 12 characters of the full container ID. The full container ID is `9f21cb77705810797c4b847dbd330d9c732ffddba14fb435470567a7a3f46cdc` which is 64 characters long. This full container ID was printed as the output of the `docker container run` command in the previous section.

Listed under the `PORTS` column, port 8080 from your local network is pointing towards port 80 inside the container. The name `gifted_sammet` is generated by Docker and can be something completely different in your computer.

The `container ls` command only lists the containers that are currently running on your system. In order to list out the containers that have run in the past you can use the `--all` or `-a` option.

As you can see, the second container in the list `reverent_torvalds` was created earlier and has exited with the status code 0, which indicates that no error was produced during the runtime of the container.

How to Name or Rename a Container

By default, every container has two identifiers. They are as follows:

- `CONTAINER ID` - a random 64 character-long string.
- `NAME` - combination of two random words, joined with an underscore.

Referring to a container based on these two random identifiers is kind of inconvenient. It would be great if the containers could be referred to using a name defined by you.

Naming a container can be achieved using the `--name` option. To run another container using the `fhsinchy/hello-dock` image with the name `hello-dock-container` you can execute the following command:

```
docker container run --detach --publish 8888:80 --name hello-doc

# b1db06e400c4c5e81a93a64d30acc1bf821bed63af36cab5cdb95d25e114f5
```

The 8080 port on local network is occupied by the `gifted_sammet` container (the container created in the previous sub-section). That's

```
docker container ls

# CONTAINER ID         IMAGE                  COMMAND
# b1db06e400c4         fhsinchy/hello-dock    "/docker-entrypoint.
# 9f21cb777058         fhsinchy/hello-dock    "/docker-entrypoint.
```

A new container with the name of `hello-dock-container` has been started.

You can even rename old containers using the `container rename` command. Syntax for the command is as follows:

```
docker container rename <container identifier> <new name>
```

To rename the `gifted_sammet` container to `hello-dock-container-2`, execute following command:

```
docker container rename gifted_sammet hello-dock-container-2
```

The command doesn't yield any output but you can verify that the changes have taken place using the `container ls` command. The `rename` command works for containers both in running state and stopped state.

How to Stop or Kill a Running Container

way.

There are two commands that deal with this task. The first one is the `container stop` command. Generic syntax for the command is as follows:

```
docker container stop <container identifier>
```

Where `container identifier` can either be the id or the name of the container.

I hope that you remember the container you started in the previous section. It's still running in the background. Get the identifier for that container using `docker container ls` (I'll be using `hello-dock-container` container for this demo). Now execute the following command to stop the container:

```
docker container stop hello-dock-container

# hello-dock-container
```

If you use the name as identifier, you'll get the name thrown back to you as output. The `stop` command shuts down a container gracefully by sending a `SIGTERM` signal. If the container doesn't stop within a certain period, a `SIGKILL` signal is sent which shuts down the container immediately.

instead. The `container kill` command follows the same syntax as the `stop` command.

```
docker container kill hello-dock-container-2

# hello-dock-container-2
```

How to Restart a Container

When I say restart I mean two scenarios specifically. They are as follows:

- Restarting a container that has been previously stopped or killed.
- Rebooting a running container.

As you've already learned from a previous sub-section, stopped containers remain in your system. If you want you can restart them. The `container start` command can be used to start any stopped or killed container. The syntax of the command is as follows:

```
docker container start <container identifier>
```

You can get the list of all containers by executing the `container ls --all` command. Then look for the containers with `Exited` status.

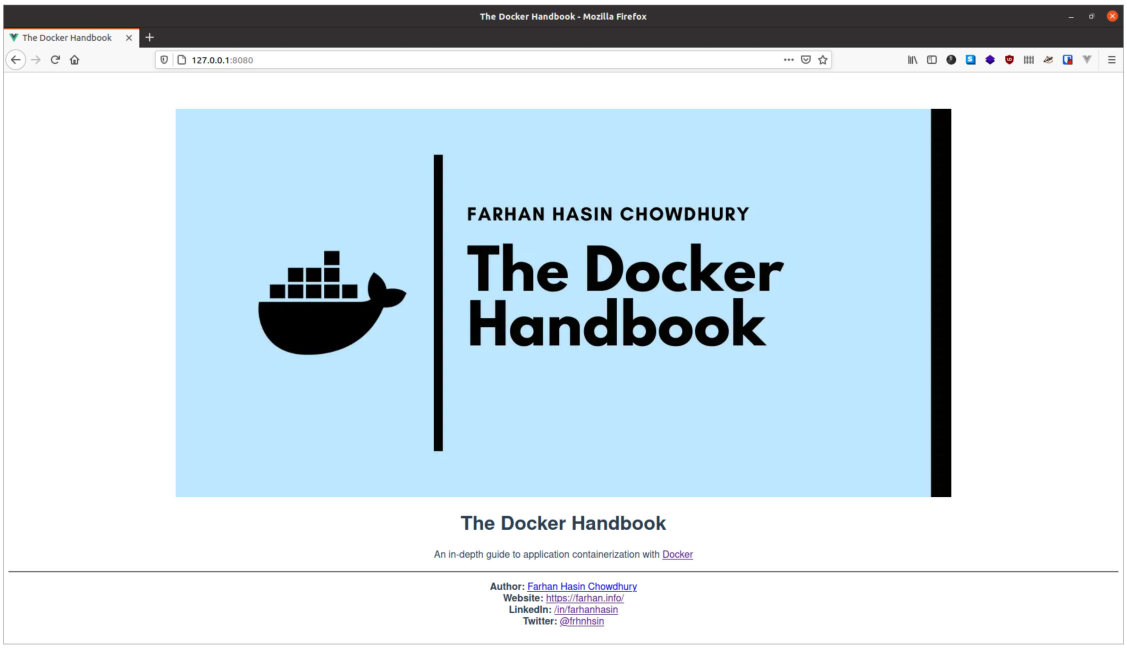
```
# 128ec8ceab71        hello-world           "/hello"
```

Now to restart the `hello-dock-container` container, you may execute the following command:

```
docker container start hello-dock-container

# hello-dock-container
```

Now you can ensure that the container is running by looking at the list of running containers using the `container ls` command.

The `container start` command starts any container in detached mode by default and retains any port configurations made previously. So if you visit `http://127.0.0.1:8080` now, you should be able to access the `hello-dock` application just like before.



`container restart` command follows the exact syntax as the `container start` command.

```
docker container restart hello-dock-container-2

# hello-dock-container-2
```

The main difference between the two commands is that the `container restart` command attempts to stop the target container and then starts it back up again, whereas the start command just starts an already stopped container.

In case of a stopped container, both commands are exactly the same. But in case of a running container, you must use the `container restart` command.

How to Create a Container Without Running

So far in this section, you've started containers using the `container run` command which is in reality a combination of two separate commands. These commands are as follows:

- `container create` command creates a container from a given image.
- `container start` command starts a container that has been already created.

Now, to perform the demonstration shown in the Running Containers section using these two commands, you can do

```
docker container create --publish 8080:80 fhsinchy/hello-dock

# 2e7ef5098bab92f4536eb9a372d9b99ed852a9a816c341127399f51a6d0538

docker container ls --all

# CONTAINER ID        IMAGE                 COMMAND
# 2e7ef5098bab        fhsinchy/hello-dock   "/docker-entrypoint.
```

Evident by the output of the `container ls --all` command, a container with the name of `hello-dock` has been created using the `fhsinchy/hello-dock` image. The `STATUS` of the container is `Created` at the moment, and, given that it's not running, it won't be listed without the use of the `--all` option.

Once the container has been created, it can be started using the `container start` command.

```
docker container start hello-dock

# hello-dock

docker container ls

# CONTAINER ID        IMAGE                 COMMAND
# 2e7ef5098bab        fhsinchy/hello-dock   "/docker-entrypoint.
```

The container `STATUS` has changed from `Created` to `Up 29 seconds` which indicates that the container is now in running state. The port configuration has also shown up in the `PORTS` column which was previously empty.

command.

How to Remove Dangling Containers

As you've already seen, containers that have been stopped or killed remain in the system. These dangling containers can take up space or can conflict with newer containers.

In order to remove a stopped container you can use the `container rm` command. The generic syntax is as follows:

```
docker container rm <container identifier>
```

To find out which containers are not running, use the `container ls --all` command and look for containers with `Exited` status.

```
docker container ls --all

# CONTAINER ID        IMAGE                  COMMAND
# b1db06e400c4        fhsinchy/hello-dock    "/docker-entrypoint.
# 9f21cb777058        fhsinchy/hello-dock    "/docker-entrypoint.
# 6cf52771dde1        fhsinchy/hello-dock    "/docker-entrypoint.
# 128ec8ceab71        hello-world            "/hello"
```

As can be seen in the output, the containers with ID `6cf52771dde1` and `128ec8ceab71` are not running. To remove the `6cf52771dde1` you can execute the following command:

You can check if the container was deleted or not by using the `container ls` command. You can also remove multiple containers at once by passing their identifiers one after another separated by spaces.

Or, instead of removing individual containers, if you want to remove all dangling containers at one go, you can use the `container prune` command.

You can check the container list using the `container ls --all` command to make sure that the dangling containers have been removed:

```
docker container ls --all

# CONTAINER ID        IMAGE                   COMMAND
# b1db06e400c4        fhsinchy/hello-dock     "/docker-entrypoint.
# 9f21cb777058        fhsinchy/hello-dock     "/docker-entrypoint.
```

If you are following the book exactly as written so far, you should only see the `hello-dock-container` and `hello-dock-container-2` in the list. I would suggest stopping and removing both containers before going on to the next section.

There is also the `--rm` option for the `container run` and `container start` commands which indicates that you want the containers removed as soon as they're stopped. To start another

```
docker container run --rm --detach --publish 8888:80 --name hell

# 0d74e14091dc6262732bee226d95702c21894678efb4043663f7911c53fb79
```

You can use the `container ls` command to verify that the container is running:

```
docker container ls

# CONTAINER ID   IMAGE                 COMMAND
# 0d74e14091dc   fhsinchy/hello-dock   "/docker-entrypoint.…"
```

Now if you stop the container and then check again with the `container ls --all` command:

```
docker container stop hello-dock-volatile

# hello-dock-volatile

docker container ls --all

# CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS
```

The container has been removed automatically. From now on I'll use the `--rm` option for most of the containers. I'll explicitly mention where it's not needed.

So far you've only run containers created from either the hello-world image or the fhsinchy/hello-dock image. These images are made for executing simple programs that are not interactive.

Well, all images are not that simple. Images can encapsulate an entire Linux distribution inside them.

Popular distributions such as Ubuntu, Fedora, and Debian all have official Docker images available in the hub. Programming languages such as python, php, go or run-times like node and deno all have their official images.

These images do not just run some pre-configured program. These are instead configured to run a shell by default. In case of the operating system images it can be something like `sh` or `bash` and in case of the programming languages or run-times, it is usually their default language shell.

As you may have already learned from your previous experiences with computers, shells are interactive programs. An image configured to run such a program is an interactive image. These images require a special `-it` option to be passed in the `container run` command.

As an example, if you run a container using the `ubuntu` image by executing `docker container run ubuntu` you'll see nothing happens. But if you execute the same command with the `-it` option, you should land directly on bash inside the Ubuntu container.

```
# NAME="Ubuntu"
# VERSION="20.04.1 LTS (Focal Fossa)"
# ID=ubuntu
# ID_LIKE=debian
# PRETTY_NAME="Ubuntu 20.04.1 LTS"
# VERSION_ID="20.04"
# HOME_URL="https://www.ubuntu.com/"
# SUPPORT_URL="https://help.ubuntu.com/"
# BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
# PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-pol
# VERSION_CODENAME=focal
# UBUNTU_CODENAME=focal
```

As you can see from the output of the `cat /etc/os-release` command, I am indeed interacting with the bash running inside the Ubuntu container.

The `-it` option sets the stage for you to interact with any interactive program inside a container. This option is actually two separate options mashed together.

- The `-i` or `--interactive` option connects you to the input stream of the container, so that you can send inputs to bash.
- The `-t` or `--tty` option makes sure that you get some good formatting and a native terminal-like experience by allocating a pseudo-tty.

You need to use the `-it` option whenever you want to run a container in interactive mode. Another example can be running the `node` image as follows:

```
# Type ".help" for more information.
# > ['farhan', 'hasin', 'chowdhury'].map(name => name.toUpperCas
# [ 'FARHAN', 'HASIN', 'CHOWDHURY' ]
```

Any valid JavaScript code can be executed in the node shell. Instead of writing `-it` you can be more verbose by writing `--interactive` `--tty` separately.

How to Execute Commands Inside a Container

In the Hello World in Docker section of this book, you've seen me executing a command inside an Alpine Linux container. It went something like this:

```
docker run alpine uname -a
# Linux f08dbbe9199b 5.8.0-22-generic #23-Ubuntu SMP Fri Oct 9 0
```

In this command, I've executed the `uname -a` command inside an Alpine Linux container. Scenarios like this (where all you want to do is to execute a certain command inside a certain container) are pretty common.

Assume that you want encode a string using the `base64` program. This is something that's available in almost any Linux or Unix based operating system (but not on Windows).

In this situation you can quickly spin up a container using images like busybox and let it do the job.

```
echo -n my-secret | base64

# bXktc2VjcmV0
```

And the generic syntax for passing a command to a container that is not running is as follows:

```
docker container run <image name> <command>
```

To perform the base64 encoding using the busybox image, you can execute the following command:

```
docker container run --rm busybox sh -c "echo -n my-secret | bas

# bXktc2VjcmV0
```

What happens here is that, in a `container run` command, whatever you pass after the image name gets passed to the default entry point of the image.

An entry point is like a gateway to the image. Most of the images except the executable images (explained in the Working With Executable Images sub-section) use shell or `sh` as the default entry-point. So any valid shell command can be passed to them as arguments.

These images are designed to behave like executable programs.

Take for example my rmbyext project. This is a simple Python script capable of recursively deleting files of given extensions. To learn more about the project, you can checkout the repository:

fhsinchy/rmbyext

Recursively removes all files with given extension(s). - fhsinchy/rmbyext

fhsinchy • GitHub

spare a ⭐ to keep me motivated

If you have both Git and Python installed, you can install this script by executing the following command:

```
pip install git+https://github.com/fhsinchy/rmbyext.git#egg=rmby
```

Assuming Python has been set up properly on your system, the script should be available anywhere through the terminal. The generic syntax for using this script is as follows:

```
rmbyext <file extension>
```

with the following files:

```
touch a.pdf b.pdf c.txt d.pdf e.txt

ls

# a.pdf  b.pdf  c.txt  d.pdf  e.txt
```

To delete all the `pdf` files from this directory, you can execute the following command:

```
rmbyext pdf

# Removing: PDF
# b.pdf
# a.pdf
# d.pdf
```

An executable image for this program should be able to take extensions of files as arguments and delete them just like the `rmbyext` program did.

The [fhsinchy/rmbyext]() image behaves in a similar manner. This image contains a copy of the `rmbyext` script and is configured to run the script on a directory `/zone` inside the container.

Now the problem is that containers are isolated from your local system, so the `rmbyext` program running inside the container doesn't have any access to your local file system. So, if somehow you can map the local directory containing the `pdf` files to the `/zone`

One way to grant a container direct access to your local file system is by using bind mounts.

A bind mount lets you form a two way data binding between the content of a local file system directory (source) and another directory inside a container (destination). This way any changes made in the destination directory will take effect on the source directory and vise versa.

Let's see a bind mount in action. To delete files using this image instead of the program itself, you can execute the following command:

```
docker container run --rm -v $(pwd):/zone fhsinchy/rmbyext pdf

# Removing: PDF
# b.pdf
# a.pdf
# d.pdf
```

As you may have already guessed by seeing the `-v $(pwd):/zone` part in the command, the `-v` or `--volume` option is used for creating a bind mount for a container. This option can take three fields separated by colons (`:`). The generic syntax for the option is as follows:

```
--volume <local file system directory absolute path>:<container
```

the container.

The source directory in my case is `/home/fhsinchy/the-zone`. Given that my terminal is opened inside the directory, `$(pwd)` will be replaced with `/home/fhsinchy/the-zone` which contains the previously mentioned `.pdf` and `.txt` files.

You can learn more about command substitution here if you want to.

The `--volume` or `-v` option is valid for the `container run` as well as the `container create` commands. We'll explore volumes in greater detail in the upcoming sections so don't worry if you didn't understand them very well here.

The difference between a regular image and an executable one is that the entry-point for an executable image is set to a custom program instead of `sh`, in this case the `rmbyext` program. And as you've learned in the previous sub-section, anything you write after the image name in a `container run` command gets passed to the entry-point of the image.

So in the end the `docker container run --rm -v $(pwd):/zone fhsinchy/rmbyext pdf` command translates to `rmbyext pdf` inside the container. Executable images are not that common in the wild but can be very useful in certain cases.

Docker Image Manipulation Basics

Now that you have a solid understanding of how to run containers using publicly available images, it's time for you to learn about running containers using them, and sharing them online.

I would suggest you to install Visual Studio Code with the official Docker Extension from the marketplace. This will greatly help your development experience.

How to Create a Docker Image

As I've already explained in the Hello World in Docker section, images are multi-layered self-contained files that act as the template for creating Docker containers. They are like a frozen, read-only copy of a container.

In order to create an image using one of your programs you must have a clear vision of what you want from the image. Take the official nginx image, for example. You can start a container using this image simply by executing the following command:

```
docker container run --rm --detach --name default-nginx --publis

# b379ecd5b6b9ae27c144e4fa12bdc5d0635543666f75c14039eea8d5f38e3f

docker container ls

# CONTAINER ID        IMAGE               COMMAND
# b379ecd5b6b9        nginx               "/docker-entrypoint.…"
```

Now, if you visit `http://127.0.0.1:8080` in the browser, you'll see a default response page.

That's all nice and good, but what if you want to make a custom NGINX image which functions exactly like the official one, but that's built by you? That's a completely valid scenario to be honest. In fact, let's do that.

In order to make a custom NGINX image, you must have a clear picture of what the final state of the image will be. In my opinion the image should be as follows:

- The image should have NGINX pre-installed which can be done using a package manager or can be built from source.
- The image should start NGINX automatically upon running.

That's simple. If you've cloned the project repository linked in this book, go inside the project root and look for a directory named `custom-nginx` in there.

Now, create a new file named `Dockerfile` inside that directory. A `Dockerfile` is a collection of instructions that, once processed by the daemon, results in an image. Content for the `Dockerfile` is as follows:

```
EXPOSE 80

RUN apt-get update && \
    apt-get install nginx -y && \
    apt-get clean && rm -rf /var/lib/apt/lists/*

CMD ["nginx", "-g", "daemon off;"]
```

Images are multi-layered files and in this file, each line (known as instructions) that you've written creates a layer for your image.

- Every valid `Dockerfile` starts with a `FROM` instruction. This instruction sets the base image for your resultant image. By setting `ubuntu:latest` as the base image here, you get all the goodness of Ubuntu already available in your custom image, so you can use things like the `apt-get` command for easy package installation.
- The `EXPOSE` instruction is used to indicate the port that needs to be published. Using this instruction doesn't mean that you won't need to `--publish` the port. You'll still need to use the `--publish` option explicitly. This `EXPOSE` instruction works like a documentation for someone who's trying to run a container using your image. It also has some other uses that I won't be discussing here.
- The `RUN` instruction in a `Dockerfile` executes a command inside the container shell. The `apt-get update && apt-get install nginx -y` command checks for updated package versions and installs NGINX. The `apt-get clean && rm -rf /var/lib/apt/lists/*` command is used for clearing the package cache because you don't want any unnecessary baggage in your image. These two commands are simple Ubuntu stuff, nothing fancy. The `RUN` instructions here are

- Finally the `CMD` instruction sets the default command for your image. This instruction is written in `exec` form here comprising of three separate parts. Here, `nginx` refers to the NGINX executable. The `-g` and `daemon off` are options for NGINX. Running NGINX as a single process inside containers is considered a best practice hence the usage of this option. The `CMD` instruction can also be written in `shell` form. You can consult the official reference for more information.

Now that you have a valid `Dockerfile` you can build an image out of it. Just like the container related commands, the image related commands can be issued using the following syntax:

```
docker image <command> <options>
```

To build an image using the `Dockerfile` you just wrote, open up your terminal inside the `custom-nginx` directory and execute the following command:

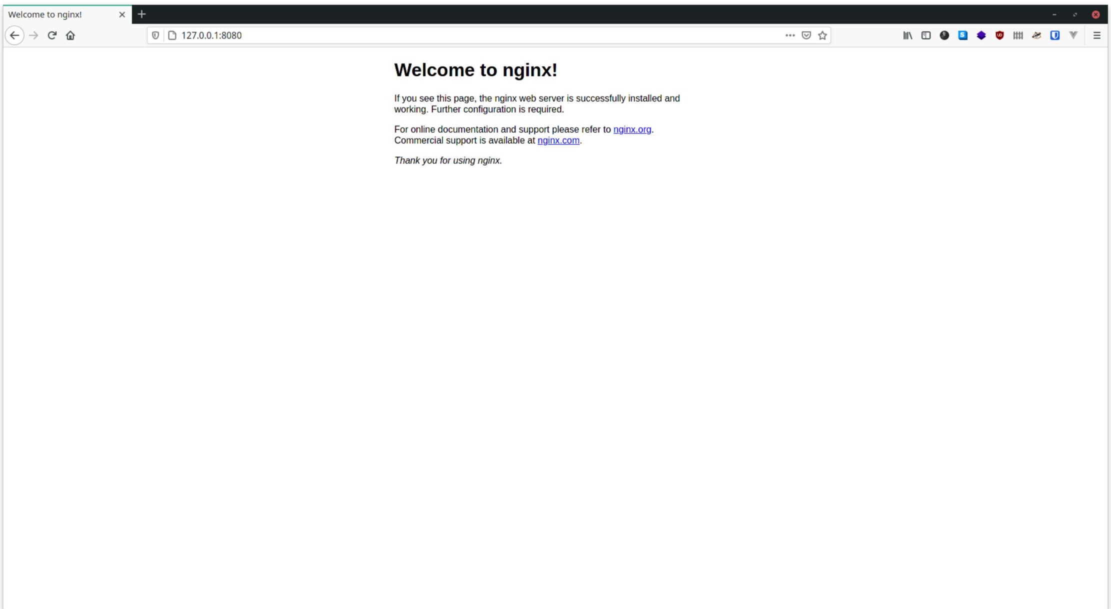
```
# Step 4/4 : CMD ["nginx", "-g", "daemon off;"]
#  ---> Running in 4792e4691660
# Removing intermediate container 4792e4691660
#  ---> 3199372aa3fc
# Successfully built 3199372aa3fc
```

To perform an image build, the daemon needs two very specific pieces of information. These are the name of the `Dockerfile` and the build context. In the command issued above:

- `docker image build` is the command for building the image. The daemon finds any file named `Dockerfile` within the context.
- The `.` at the end sets the context for this build. The context means the directory accessible by the daemon during the build process.

Now to run a container using this image, you can use the `container run` command coupled with the image ID that you received as the result of the build process. In my case the id is `3199372aa3fc` evident by the `Successfully built 3199372aa3fc` line in the previous code block.

```
docker container run --rm --detach --name custom-nginx-packaged

# ec09d4e1f70c903c3b954c8d7958421cdd1ae3d079b57f929e44131fbf8069

docker container ls

# CONTAINER ID        IMAGE               COMMAND
# ec09d4e1f70c        3199372aa3fc        "nginx -g 'daemon of…"
```



How to Tag Docker Images

Just like containers, you can assign custom identifiers to your images instead of relying on the randomly generated ID. In case of an image, it's called tagging instead of naming. The `--tag` or `-t` option is used in such cases.

Generic syntax for the option is as follows:

```
--tag <image repository>:<image tag>
```

The repository is usually known as the image name and the tag indicates a certain build or version.

Take the official mysql image, for example. If you want to run a container using a specific version of MySQL, like 5.7, you can

In order to tag your custom NGINX image with `custom-nginx:packaged` you can execute the following command:

```
docker image build --tag custom-nginx:packaged .

# Sending build context to Docker daemon  1.055MB
# Step 1/4 : FROM ubuntu:latest
#  ---> f63181f19b2f
# Step 2/4 : EXPOSE 80
#  ---> Running in 53ab370b9efc
# Removing intermediate container 53ab370b9efc
#  ---> 6d6460a74447
# Step 3/4 : RUN apt-get update &&     apt-get install nginx -y 
#  ---> Running in b4951b6b48bb
### LONG INSTALLATION STUFF GOES HERE ###
# Removing intermediate container b4951b6b48bb
#  ---> fdc6cdd8925a
# Step 4/4 : CMD ["nginx", "-g", "daemon off;"]
#  ---> Running in 3bdbd2af4f0e
# Removing intermediate container 3bdbd2af4f0e
#  ---> f8837621b99d
# Successfully built f8837621b99d
# Successfully tagged custom-nginx:packaged
```

Nothing will change except the fact that you can now refer to your image as `custom-nginx:packaged` instead of some long random string.

In cases where you forgot to tag an image during build time, or maybe you want to change the tag, you can use the `image tag` command to do that:

How to List and Remove Docker Images

Just like the `container ls` command, you can use the `image ls` command to list all the images in your local system:

```
docker image ls

# REPOSITORY      TAG        IMAGE ID       CREATED          SIZE
# <none>          <none>     3199372aa3fc   7 seconds ago    132MB
# custom-nginx    packaged   f8837621b99d   4 minutes ago    132MB
```

Images listed here can be deleted using the `image rm` command. The generic syntax is as follows:

```
docker image rm <image identifier>
```

The identifier can be the image ID or image repository. If you use the repository, you'll have to identify the tag as well. To delete the `custom-nginx:packaged` image, you may execute the following command:

You can also use the `image prune` command to cleanup all un-tagged dangling images as follows:

```
docker image prune --force

# Deleted Images:
# deleted: sha256:ba9558bdf2beda81b9acc652ce4931a85f0fc7f69dbc91
# deleted: sha256:ad9cc3ff27f0d192f8fa5fadebf813537e02e6ad472f65
# deleted: sha256:f1e9b82068d43c1bb04ff3e4f0085b9f8903a12b27196d
# deleted: sha256:ec16024aa036172544908ec4e5f842627d04ef99ee9b8d

# Total reclaimed space: 59.19MB
```

The `--force` or `-f` option skips any confirmation questions. You can also use the `--all` or `-a` option to remove all cached images in your local registry.

How to Understand the Many Layers of a Docker Image

From the very beginning of this book, I've been saying that images are multi-layered files. In this sub-section I'll demonstrate the various layers of an image and how they play an important role in the build process of that image.

For this demonstration, I'll be using the `custom-nginx:packaged` image from the previous sub-section.

To visualize the many layers of an image, you can use the `image history` command. The various layers of the `custom-`

```
docker image history custom-nginx:packaged

# IMAGE                CREATED             CREATED BY
# 7f16387f7307         5 minutes ago       /bin/sh -c #(nop)  CMD
# 587c805fe8df         5 minutes ago       /bin/sh -c apt-get upd
# 6fe4e51e35c1         6 minutes ago       /bin/sh -c #(nop)  EXP
# d70eaf7277ea         17 hours ago        /bin/sh -c #(nop)  CMD
# <missing>            17 hours ago        /bin/sh -c mkdir -p /r
# <missing>            17 hours ago        /bin/sh -c [ -z "$(apt
# <missing>            17 hours ago        /bin/sh -c set -xe   &
# <missing>            17 hours ago        /bin/sh -c #(nop) ADD
```

There are eight layers of this image. The upper most layer is the latest one and as you go down the layers get older. The upper most layer is the one that you usually use for running containers.

Now, let's have a closer look at the images beginning from image `d70eaf7277ea` down to `7f16387f7307`. I'll ignore the bottom four layers where the `IMAGE` is `<missing>` as they are not of our concern.

- `d70eaf7277ea` was created by `/bin/sh -c #(nop)  CMD ["/bin/bash"]` which indicates that the default shell inside Ubuntu has been loaded successfully.
- `6fe4e51e35c1` was created by `/bin/sh -c #(nop)  EXPOSE 80` which was the second instruction in your code.
- `587c805fe8df` was created by `/bin/sh -c apt-get update && apt-get install nginx -y && apt-get clean && rm -rf /var/lib/apt/lists/*` which was the third instruction in your code. You can also see that this image has a size of `60MB` given all necessary packages were installed during the execution of this instruction.

which sets the default command for this image.

As you can see, the image comprises of many read-only layers, each recording a new set of changes to the state triggered by certain instructions. When you start a container using an image, you get a new writable layer on top of the other layers.

This layering phenomenon that happens every time you work with Docker has been made possible by an amazing technical concept called a union file system. Here, union means union in set theory. According to Wikipedia -

> It allows files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system. Contents of directories which have the same path within the merged branches will be seen together in a single merged directory, within the new, virtual filesystem.

By utilizing this concept, Docker can avoid data duplication and can use previously created layers as a cache for later builds. This results in compact, efficient images that can be used everywhere.

How to Build NGINX from Source

In the previous sub-section, you learned about the `FROM`, `EXPOSE`, `RUN` and `CMD` instructions. In this sub-section you'll be learning a lot more about other instructions.

In this sub-section you'll again create a custom NGINX image. But the twist is that you'll be building NGINX from source instead of

In order to build NGINX from source, you first need the source of NGINX. If you've cloned my projects repository you'll see a file named `nginx-1.19.2.tar.gz` inside the `custom-nginx` directory. You'll use this archive as the source for building NGINX.

Before diving into writing some code, let's plan out the process first. The image creation process this time can be done in seven steps. These are as follows:

- Get a good base image for building the application, like ubuntu.
- Install necessary build dependencies on the base image.
- Copy the `nginx-1.19.2.tar.gz` file inside the image.
- Extract the contents of the archive and get rid of it.
- Configure the build, compile and install the program using the `make` tool.
- Get rid of the extracted source code.
- Run `nginx` executable.

Now that you have a plan, let's begin by opening up old `Dockerfile` and updating its contents as follows:

```
FROM ubuntu:latest

RUN apt-get update && \
    apt-get install build-essential\
                    libpcre3 \
                    libpcre3-dev \
                    zlib1g \
                    zlib1g-dev \
```

```
COPY nginx-1.19.2.tar.gz .

RUN tar -xvf nginx-1.19.2.tar.gz && rm nginx-1.19.2.tar.gz

RUN cd nginx-1.19.2 && \
    ./configure \
        --sbin-path=/usr/bin/nginx \
        --conf-path=/etc/nginx/nginx.conf \
        --error-log-path=/var/log/nginx/error.log \
        --http-log-path=/var/log/nginx/access.log \
        --with-pcre \
        --pid-path=/var/run/nginx.pid \
        --with-http_ssl_module && \
    make && make install

RUN rm -rf /nginx-1.19.2

CMD ["nginx", "-g", "daemon off;"]
```

As you can see, the code inside the `Dockerfile` reflects the seven steps I talked about above.

- The `FROM` instruction sets Ubuntu as the base image making an ideal environment for building any application.
- The `RUN` instruction installs standard packages necessary for building NGINX from source.
- The `COPY` instruction here is something new. This instruction is responsible for copying the the `nginx-1.19.2.tar.gz` file inside the image. The generic syntax for the `COPY` instruction is `COPY <source> <destination>` where source is in your local filesystem and the destination is inside your image. The `.` as the destination means the working directory inside the image which is by default `/` unless set otherwise.

- The archive file contains a directory called `nginx-1.19.2` containing the source code. So on the next step, you'll have to `cd` inside that directory and perform the build process. You can read the How to Install Software from Source Code… and Remove it Afterwards article to learn more on the topic.
- Once the build and installation is complete, you remove the `nginx-1.19.2` directory using `rm` command.
- On the final step you start NGINX in single process mode just like you did before.

Now to build an image using this code, execute the following command:

```
# Removing intermediate container 3110c7fdbd57
#  ---> eae55f7369d3
# Successfully built eae55f7369d3
# Successfully tagged custom-nginx:built
```

This code is alright but there are some places where we can make improvements.

- Instead of hard coding the filename like `nginx-1.19.2.tar.gz`, you can create an argument using the `ARG` instruction. This way, you'll be able to change the version or filename by just changing the argument.
- Instead of downloading the archive manually, you can let the daemon download the file during the build process. There is another instruction like `COPY` called the `ADD` instruction which is capable of adding files from the internet.

Open up the `Dockerfile` file and update its content as follows:

```
RUN cd ${FILENAME} && \
    ./configure \
        --sbin-path=/usr/bin/nginx \
        --conf-path=/etc/nginx/nginx.conf \
        --error-log-path=/var/log/nginx/error.log \
        --http-log-path=/var/log/nginx/access.log \
        --with-pcre \
        --pid-path=/var/run/nginx.pid \
        --with-http_ssl_module && \
    make && make install

RUN rm -rf /${FILENAME}}

CMD ["nginx", "-g", "daemon off;"]
```

The code is almost identical to the previous code block except for a new instruction called `ARG` on line 13, 14 and the usage of the `ADD` instruction on line 16. Explanation for the updated code is as follows:

- The `ARG` instruction lets you declare variables like in other languages. These variables or arguments can later be accessed using the `${argument name}` syntax. Here, I've put the filename `nginx-1.19.2` and the file extension `tar.gz` in two separate arguments. This way I can switch between newer versions of NGINX or the archive format by making a change in just one place. In the code above, I've added default values to the variables. Variable values can be passed as options of the `image build` command as well. You can consult the official reference for more details.
- In the `ADD` instruction, I've formed the download URL dynamically using the arguments declared above. The `https://nginx.org/download/${FILENAME}.${EXTENSION}`

extension by changing it in just one place thanks to the `ARG` instruction.

- The `ADD` instruction doesn't extract files obtained from the internet by default, hence the usage of `tar` on line 18.

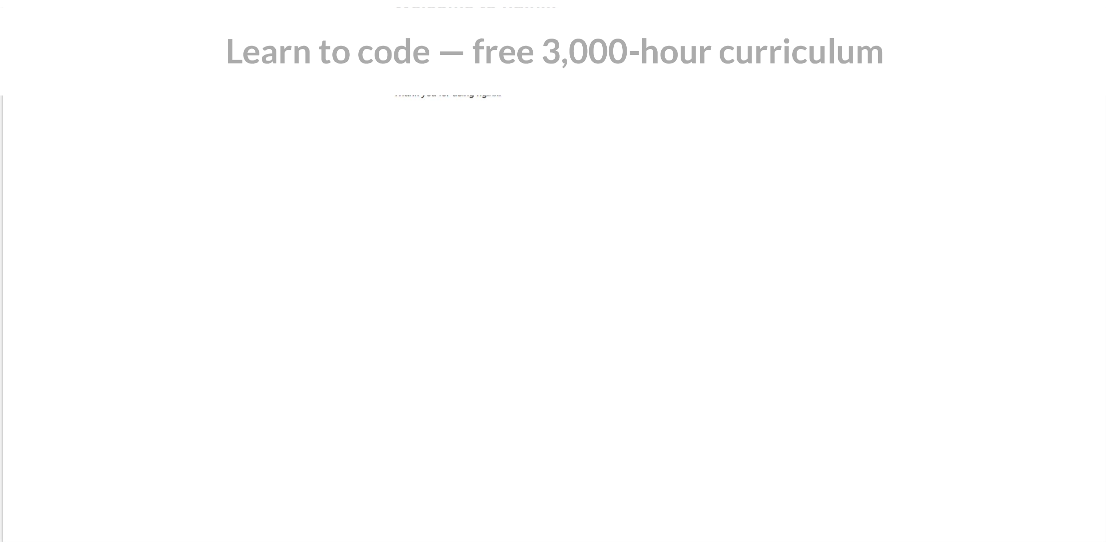
The rest of the code is almost unchanged. You should be able to understand the usage of the arguments by yourself now. Finally let's try to build an image from this updated code.

```
#  ---> Running in 63ee44b571bb
# Removing intermediate container 63ee44b571bb
#  ---> 4ce79556db1b
# Successfully built 4ce79556db1b
# Successfully tagged custom-nginx:built
```

Now you should be able to run a container using the `custom-nginx:built` image.

```
docker container run --rm --detach --name custom-nginx-built --p

# 90ccdbc0b598dddc4199451b2f30a942249d85a8ed21da3c8d14612f17eed0

docker container ls

# CONTAINER ID        IMAGE                 COMMAND
# 90ccdbc0b598        custom-nginx:built    "nginx -g 'daemon of…
```

A container using the `custom-nginx:built-v2` image has been successfully run. The container should be accessible at `http://127.0.0.1:8080` now.



And here is the trusty default response page from NGINX. You can visit the official reference site to learn more about the available instructions.

How to Optimize Docker Images

The image we built in the last sub-section is functional but very unoptimized. To prove my point let's have a look at the size of the image using the `image ls` command:

```
docker image ls

# REPOSITORY      TAG      IMAGE ID       CREATED          S
# custom-nginx    built    1f3aaf40bb54   16 minutes ago   3
```

For an image containing only NGINX, that's too much. If you pull the official image and check its size, you'll see how small it is:

```
# 30c386181b68: Pull complete
# b15158e9ebbe: Pull complete
# Digest: sha256:ebd0fd56eb30543a9195280eb81af2a9a8e6143496accd6a
# Status: Downloaded newer image for nginx:stable
# docker.io/library/nginx:stable

docker image ls

# REPOSITORY          TAG        IMAGE ID       CREATED          SI
# custom-nginx        built      1f3aaf40bb54   25 minutes ago   34
# nginx               stable     b9e1dc12387a   11 days ago      13
```

In order to find out the root cause, let's have a look at the `Dockerfile` first:

```
        --with-http_ssl_module && \
    make && make install

RUN rm -rf /${FILENAME}}

CMD ["nginx", "-g", "daemon off;"]
```

As you can see on line 3, the `RUN` instruction installs a lot of stuff. Although these packages are necessary for building NGINX from source, they are not necessary for running it.

Out of the 6 packages that we installed, only two are necessary for running NGINX. These are `libpcre3` and `zlib1g`. So a better idea would be to uninstall the other packages once the build process is done.

To do so, update your `Dockerfile` as follows:

```
        --conf-path=/etc/nginx/nginx.conf \
        --error-log-path=/var/log/nginx/error.log \
        --http-log-path=/var/log/nginx/access.log \
        --with-pcre \
        --pid-path=/var/run/nginx.pid \
        --with-http_ssl_module && \
    make && make install && \
    cd / && rm -rfv /${FILENAME} && \
    apt-get remove build-essential \
                    libpcre3-dev \
                    zlib1g-dev \
                    libssl-dev \
                    -y && \
    apt-get autoremove -y && \
    apt-get clean && rm -rf /var/lib/apt/lists/*

CMD ["nginx", "-g", "daemon off;"]
```

As you can see, on line 10 a single `RUN` instruction is doing all the necessary heavy-lifting. The exact chain of events is as follows:

- From line 10 to line 17, all the necessary packages are being installed.
- On line 18, the source code is being extracted and the downloaded archive gets removed.
- From line 19 to line 28, NGINX is configured, built, and installed on the system.
- On line 29, the extracted files from the downloaded archive get removed.
- From line 30 to line 36, all the unnecessary packages are being uninstalled and cache cleared. The `libpcre3` and `zlib1g` packages are needed for running NGINX so we keep them.

If you install packages and then remove them in separate `RUN` instructions, they'll live in separate layers of the image. Although the final image will not have the removed packages, their size will still be added to the final image since they exist in one of the layers consisting the image. So make sure you make these kind of changes on a single layer.

Let's build an image using this `Dockerfile` and see the differences.

```
# REPOSITORY          TAG       IMAGE ID       CREATED
# custom-nginx        built     512aa6a95a93   About a minute ago
# nginx               stable    b9e1dc12387a   11 days ago
```

As you can see, the image size has gone from being 343MB to 81.6MB. The official image is 133MB. This is a pretty optimized build, but we can go a bit further in the next sub-section.

Embracing Alpine Linux

If you've been fiddling around with containers for some time now, you may have heard about something called Alpine Linux. It's a full-featured Linux distribution like Ubuntu, Debian or Fedora.

But the good thing about Alpine is that it's built around `musl libc` and `busybox` and is lightweight. Where the latest ubuntu image weighs at around 28MB, alpine is 2.8MB.

Apart from the lightweight nature, Alpine is also secure and is a much better fit for creating containers than the other distributions.

Although not as user friendly as the other commercial distributions, the transition to Alpine is still very simple. In this sub-section you'll learn about recreating the `custom-nginx` image using the Alpine image as its base.

Open up your `Dockerfile` and update its content as follows:

```
ADD https://nginx.org/download/${FILENAME}.${EXTENSION} .

RUN apk add --no-cache pcre zlib && \
    apk add --no-cache \
            --virtual .build-deps \
            build-base \
            pcre-dev \
            zlib-dev \
            openssl-dev && \
    tar -xvf ${FILENAME}.${EXTENSION} && rm ${FILENAME}.${EXTENSION} &
    cd ${FILENAME} && \
    ./configure \
        --sbin-path=/usr/bin/nginx \
        --conf-path=/etc/nginx/nginx.conf \
        --error-log-path=/var/log/nginx/error.log \
        --http-log-path=/var/log/nginx/access.log \
        --with-pcre \
        --pid-path=/var/run/nginx.pid \
        --with-http_ssl_module && \
    make && make install && \
    cd / && rm -rfv /${FILENAME} && \
    apk del .build-deps

CMD ["nginx", "-g", "daemon off;"]
```

The code is almost identical except for a few changes. I'll be listing the changes and explaining them as I go:

- Instead of using `apt-get install` for installing packages, we use `apk add`. The `--no-cache` option means that the downloaded package won't be cached. Likewise we'll use `apk del` instead of `apt-get remove` to uninstall packages.
- The `--virtual` option for the `apk add` command is used for bundling a bunch of packages into a single virtual package for easier management. Packages that are needed only for building the program are labeled as `.build-deps` which are

official docs.

- The package names are a bit different here. Usually every Linux distribution has its package repository available to everyone where you can search for packages. If you know the packages required for a certain task, then you can just head over to the designated repository for a distribution and search for it. You can look up Alpine Linux packages here.

Now build a new image using this `Dockerfile` and see the difference in file size:

```
docker image ls

# REPOSITORY          TAG       IMAGE ID       CREATED          SI
# custom-nginx        built     3e186a3c6830   8 seconds ago    12
```

Where the ubuntu version was 81.6MB, the alpine one has come down to 12.8MB which is a massive gain. Apart from the `apk` package manager, there are some other things that differ in Alpine from Ubuntu but they're not that big a deal. You can just search the internet whenever you get stuck.

How to Create Executable Docker Images

In the previous section you worked with the fhsinchy/rmbyext image. In this section you'll learn how to make such an executable image.

To begin with, open up the directory where you've cloned the repository that came with this book. The code for the `rmbyext` application resides inside the sub-directory with the same name.

Before you start working on the `Dockerfile` take a moment to plan out what the final output should be. In my opinion it should be like something like this:

- The image should have Python pre-installed.
- It should contain a copy of my `rmbyext` script.
- A working directory should be set where the script will be executed.

To build the above mentioned image, take the following steps:

- Get a good base image for running Python scripts, like python.
- Set-up the working directory to an easily accessible directory.
- Install Git so that the script can be installed from my GitHub repository.
- Install the script using Git and pip.
- Get rid of the build's unnecessary packages.
- Set `rmbyext` as the entry-point for this image.

Now create a new `Dockerfile` inside the `rmbyext` directory and put the following code in it:

```
FROM python:3-alpine

WORKDIR /zone

RUN apk add --no-cache git && \
    pip install git+https://github.com/fhsinchy/rmbyext.git#egg=rmbyex
    apk del git

ENTRYPOINT [ "rmbyext" ]
```

The explanation for the instructions in this file is as follows:

- The `FROM` instruction sets python as the base image, making an ideal environment for running Python scripts. The `3-`

- The `WORKDIR` instruction sets the default working directory to `/zone` here. The name of the working directory is completely random here. I found zone to be a fitting name, you may use anything you want.
- Given the `rmbyext` script is installed from GitHub, `git` is an install time dependency. The `RUN` instruction on line 5 installs `git` then installs the `rmbyext` script using Git and pip. It also gets rid of `git` afterwards.
- Finally on line 9, the `ENTRYPOINT` instruction sets the `rmbyext` script as the entry-point for this image.

In this entire file, line 9 is the magic that turns this seemingly normal image into an executable one. Now to build the image you can execute following command:

```
#  ---> 1746b0cedbc7
# Successfully built 1746b0cedbc7
# Successfully tagged rmbyext:latest

docker image ls

# REPOSITORY           TAG         IMAGE ID       CREATED          S
# rmbyext              latest      1746b0cedbc7   4 minutes ago    5
```

Here I haven't provided any tag after the image name, so the image has been tagged as `latest` by default. You should be able to run the image as you saw in the previous section. Remember to refer to the actual image name you've set, instead of `fhsinchy/rmbyext` here.

How to Share Your Docker Images Online

Now that you know how to make images, it's time to share them with the world. Sharing images online is easy. All you need is an account at any of the online registries. I'll be using Docker Hub here.

Navigate to the Sign Up page and create a free account. A free account allows you to host unlimited public repositories and one private repository.

Once you've created the account, you'll have to sign in to it using the docker CLI. So open up your terminal and execute the following command to do so:

```
# Login Succeeded
```

You'll be prompted for your username and password. If you input them properly, you should be logged in to your account successfully.

In order to share an image online, the image has to be tagged. You've already learned about tagging in a previous sub-section. Just to refresh your memory, the generic syntax for the `--tag` or `-t` option is as follows:

```
--tag <image repository>:<image tag>
```

As an example, let's share the `custom-nginx` image online. To do so, open up a new terminal window inside the `custom-nginx` project directory.

To share an image online, you'll have to tag it following the `<docker hub username>/<image name>:<image tag>` syntax. My username is `fhsinchy` so the command will look like this:

```
#  ---> 36efdf6efacc
# Step 5/9 : ADD https://nginx.org/download/${FILENAME}.${EXTENS
# Downloading [==================================================
#  ---> dba252f8d609
# Step 6/9 : RUN tar -xvf ${FILENAME}.${EXTENSION} && rm ${FILEN
#  ---> Running in 2f5b091b2125
### LONG EXTRACTION STUFF GOES HERE ###
# Removing intermediate container 2f5b091b2125
#  ---> 2c9a325d74f1
# Step 7/9 : RUN cd ${FILENAME} &&     ./configure         --sbi
#  ---> Running in 11cc82dd5186
### LONG CONFIGURATION AND BUILD STUFF GOES HERE ###
# Removing intermediate container 11cc82dd5186
#  ---> 6c122e485ec8
# Step 8/9 : RUN rm -rf /${FILENAME}}
#  ---> Running in 04102366960b
# Removing intermediate container 04102366960b
#  ---> 6bfa35420a73
# Step 9/9 : CMD ["nginx", "-g", "daemon off;"]
#  ---> Running in 63ee44b571bb
# Removing intermediate container 63ee44b571bb
#  ---> 4ce79556db1b
# Successfully built 4ce79556db1b
# Successfully tagged fhsinchy/custom-nginx:latest
```

In this command the `fhsinchy/custom-nginx` is the image repository and `latest` is the tag. The image name can be anything you want and can not be changed once you've uploaded the image. The tag can be changed whenever you want and usually reflects the version of the software or different kind of builds.

Take the `node` image as an example. The `node:lts` image refers to the long term support version of Node.js whereas the `node:lts-alpine` version refers to the Node.js version built for Alpine Linux, which is much smaller than the regular one.

older version of the image as `latest` , then Docker will not make any extra effort to cross check that.

Once the image has been built, you can them upload it by executing the following command:

```
docker image push <image repository>:<image tag>
```

So in my case the command will be as follows:

```
docker image push fhsinchy/custom-nginx:latest

# The push refers to repository [docker.io/fhsinchy/custom-nginx
# 4352b1b1d9f5: Pushed
# a4518dd720bd: Pushed
# 1d756dc4e694: Pushed
# d7a7e2b6321a: Pushed
# f6253634dc78: Mounted from library/ubuntu
# 9069f84dbbe9: Mounted from library/ubuntu
# bacd3af13903: Mounted from library/ubuntu
# latest: digest: sha256:ffe93440256c9edb2ed67bf3bba3c204fec3a46
```

Depending on the image size, the upload may take some time. Once it's done you should able to find the image in your hub profile page.

How to Containerize a JavaScript Application

In this sub-section, you'll be working with the source code of the fhsinchy/hello-dock image that you worked with on a previous section. In the process of containerizing this very simple application, you'll be introduced to volumes and multi-staged builds, two of the most important concepts in Docker.

How to Write the Development Dockerfile

To begin with, open up the directory where you've cloned the repository that came with this book. Code for the `hello-dock` application resides inside the sub-directory with the same name.

This is a very simple JavaScript project powered by the vitejs/vite project. Don't worry though, you don't need to know JavaScript or vite in order to go through this sub-section. Having a basic understanding of Node.js and npm will suffice.

Just like any other project you've done in the previous sub-section, you'll begin by making a plan of how you want this application to run. In my opinion, the plan should be as follows:

- Get a good base image for running JavaScript applications, like node.
- Set the default working directory inside the image.
- Copy the `package.json` file into the image.
- Install necessary dependencies.
- Copy the rest of the project files.
- Start the `vite` development server by executing `npm run dev` command.

needs to be run.

Now if you put the above mentioned plan inside `Dockerfile.dev` , the file should look like as follows:

```
FROM node:lts-alpine

EXPOSE 3000

USER node

RUN mkdir -p /home/node/app

WORKDIR /home/node/app

COPY ./package.json .
RUN npm install

COPY . .

CMD [ "npm", "run", "dev" ]
```

The explanation for this code is as follows:

- The `FROM` instruction here sets the official Node.js image as the base, giving you all the goodness of Node.js necessary to run any JavaScript application. The `lts-alpine` tag indicates that you want to use the Alpine variant, long term support version of the image. Available tags and necessary documentation for the image can be found on the node hub page.
- The `USER` instruction sets the default user for the image to `node` . By default Docker runs containers as the root user.

non-root user named `node` which you can set as the default user using the `USER` instruction.

- The `RUN mkdir -p /home/node/app` instruction creates a directory called `app` inside the home directory of the `node` user. The home directory for any non-root user in Linux is usually `/home/<user name>` by default.
- Then the `WORKDIR` instruction sets the default working directory to the newly created `/home/node/app` directory. By default the working directory of any image is the root. You don't want any unnecessary files sprayed all over your root directory, do you? Hence you change the default working directory to something more sensible like `/home/node/app` or whatever you like. This working directory will be applicable to any subsequent `COPY`, `ADD`, `RUN` and `CMD` instructions.
- The `COPY` instruction here copies the `package.json` file which contains information regarding all the necessary dependencies for this application. The `RUN` instruction executes the `npm install` command which is the default command for installing dependencies using a `package.json` file in Node.js projects. The `.` at the end represents the working directory.
- The second `COPY` instruction copies the rest of the content from the current directory (`.`) of the host filesystem to the working directory (`.`) inside the image.
- Finally, the `CMD` instruction here sets the default command for this image which is `npm run dev` written in `exec` form.
- The `vite` development server by default runs on port `3000`, and adding an `EXPOSE` command seemed like a good idea, so

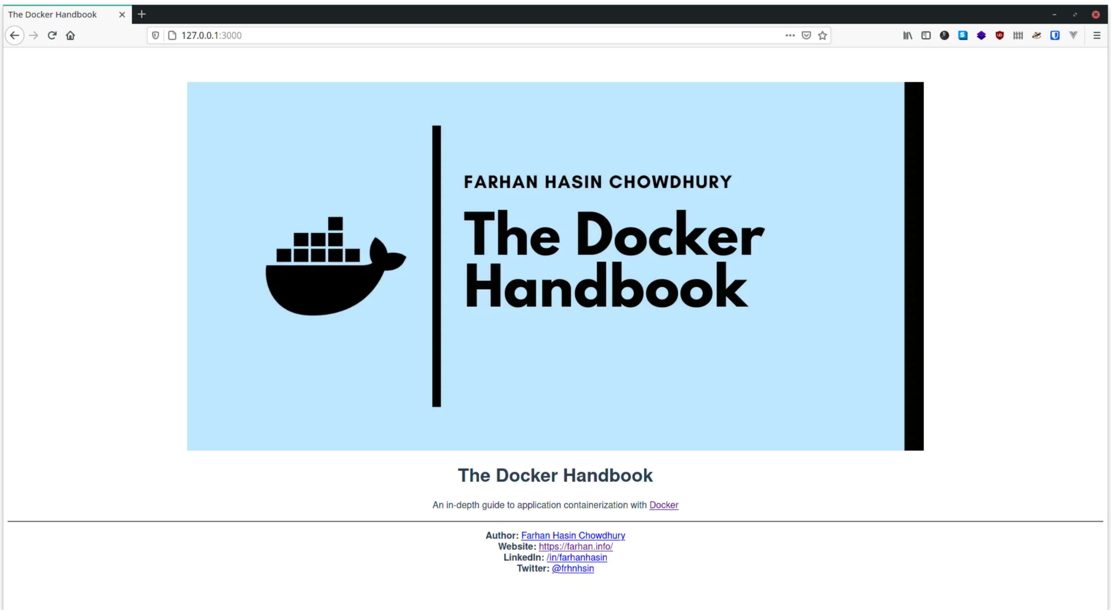
Now, to build an image from this `Dockerfile.dev` you can execute the following command:

```
docker image build --file Dockerfile.dev --tag hello-dock:dev .

# Step 1/7 : FROM node:lts
#  ---> b90fa0d7cbd1
# Step 2/7 : EXPOSE 3000
#  ---> Running in 722d639badc7
# Removing intermediate container 722d639badc7
#  ---> e2a8aa88790e
# Step 3/7 : WORKDIR /app
#  ---> Running in 998e254b4d22
# Removing intermediate container 998e254b4d22
#  ---> 6bd4c42892a4
# Step 4/7 : COPY ./package.json .
#  ---> 24fc5164a1dc
# Step 5/7 : RUN npm install
#  ---> Running in 23b4de3f930b
### LONG INSTALLATION STUFF GOES HERE ###
# Removing intermediate container 23b4de3f930b
#  ---> c17ecb19a210
# Step 6/7 : COPY . .
#  ---> afb6d9a1bc76
# Step 7/7 : CMD [ "npm", "run", "dev" ]
#  ---> Running in a7ff529c28fe
# Removing intermediate container a7ff529c28fe
#  ---> 1792250adb79
# Successfully built 1792250adb79
# Successfully tagged hello-dock:dev
```

Given the filename is not `Dockerfile` you have to explicitly pass the filename using the `--file` option. A container can be run using this image by executing the following command:

```
    --publish 3000:3000 \
    --name hello-dock-dev \
    hello-dock:dev

# 21b9b1499d195d85e81f0e8bce08f43a64b63d589c5f15cbbd0b9c0cb07ae2
```

Now visit `http://127.0.0.1:3000` to see the `hello-dock` application in action.

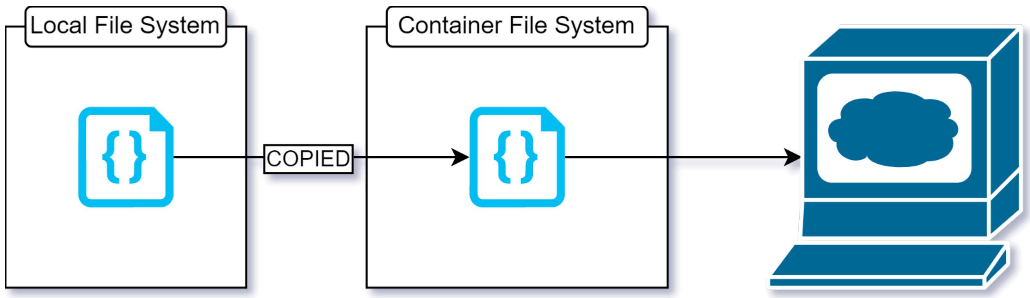


Congratulations on running your first real-world application inside a container. The code you've just written is okay but there is one big issue with it and a few places where it can be improved. Let's begin with the issue first.

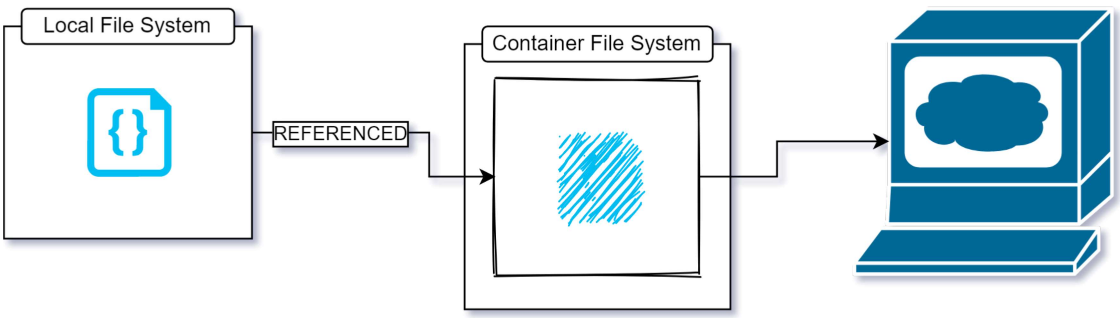
How to Work With Bind Mounts in Docker

If you've worked with any front-end JavaScript framework before, you should know that the development servers in these frameworks usually come with a hot reload feature. That is if you make a change

But if you make any changes in your code right now, you'll see nothing happening to your application running in the browser. This is because you're making changes in the code that you have in your local file system but the application you're seeing in the browser resides inside the container file system.



To solve this issue, you can again make use of a bind mount. Using bind mounts, you can easily mount one of your local file system directories inside a container. Instead of making a copy of the local file system, the bind mount can reference the local file system directly from inside the container.



This way, any changes you make to your local source code will reflect immediately inside the container, triggering the hot reload feature of the `vite` development server. Changes made to the file

You've already learned in the Working With Executable Images sub-section, bind mounts can be created using the `--volume` or `-v` option for the `container run` or `container start` commands. Just to remind you, the generic syntax is as follows:

```
--volume <local file system directory absolute path>:<container
```

Stop your previously started `hello-dock-dev` container, and start a new container by executing the following command:

```
docker container run \
    --rm \
    --publish 3000:3000 \
    --name hello-dock-dev \
    --volume $(pwd):/home/node/app \
    hello-dock:dev

# sh: 1: vite: not found
# npm ERR! code ELIFECYCLE
# npm ERR! syscall spawn
# npm ERR! file sh
# npm ERR! errno ENOENT
# npm ERR! hello-dock@0.0.0 dev: `vite`
# npm ERR! spawn ENOENT
# npm ERR!
# npm ERR! Failed at the hello-dock@0.0.0 dev script.
# npm ERR! This is probably not a problem with npm. There is lik
# npm WARN Local package.json exists, but node_modules missing,
```

Keep in mind, I've omitted the `--detach` option and that's to demonstrate a very important point. As you can see, the application hot reloads, it introduces another problem. If you have any previous experience with Node.js, you may know that the dependencies of a Node.js project live inside the `node_modules` directory on the project root.

Now that you're mounting the project root on your local file system as a volume inside the container, the content inside the container gets replaced along with the `node_modules` directory containing all the dependencies. This means that the `vite` package has gone missing.

How to Work With Anonymous Volumes in Docker

This problem can be solved using an anonymous volume. An anonymous volume is identical to a bind mount except that you don't need to specify the source directory here. The generic syntax for creating an anonymous volume is as follows:

```
--volume <container file system directory absolute path>:<read w
```

So the final command for starting the `hello-dock` container with both volumes should be as follows:

```
# 53d1cfdb3ef148eb6370e338749836160f75f076d0fbec3c2a9b059a8992de
```

Here, Docker will take the entire `node_modules` directory from inside the container and tuck it away in some other directory managed by the Docker daemon on your host file system and will mount that directory as `node_modules` inside the container.

How to Perform Multi-Staged Builds in Docker

So far in this section, you've built an image for running a JavaScript application in development mode. Now if you want to build the image in production mode, some new challenges show up.

In development mode the `npm run serve` command starts a development server that serves the application to the user. That server not only serves the files but also provides the hot reload feature.

In production mode, the `npm run build` command compiles all your JavaScript code into some static HTML, CSS, and JavaScript files. To run these files you don't need node or any other runtime dependencies. All you need is a server like `nginx` for example.

To create an image where the application runs in production mode, you can take the following steps:

- Use `node` as the base image and build the application.
- Install `nginx` inside the node image and use that to serve the static files.

- Use `node` image as the base and build the application.
- Copy the files created using the `node` image to an `nginx` image.
- Create the final image based on `nginx` and discard all `node` related stuff.

This way your image only contains the files that are needed and becomes really handy.

This approach is a multi-staged build. To perform such a build, create a new `Dockerfile` inside your `hello-dock` project directory and put the following content in it:

```
FROM node:lts-alpine as builder

WORKDIR /app

COPY ./package.json ./
RUN npm install

COPY . .
RUN npm run build

FROM nginx:stable-alpine

EXPOSE 80

COPY --from=builder /app/dist /usr/share/nginx/html
```

As you can see the `Dockerfile` looks a lot like your previous ones with a few oddities. The explanation for this file is as follows:

- From line 3 to line 9, it's standard stuff that you've seen many times before. The `RUN npm run build` command actually compiles the entire application and tucks it inside `/app/dist` directory where `/app` is the working directory and `/dist` is the default output directory for `vite` applications.
- Line 11 starts the second stage of the build using `nginx:stable-alpine` as the base image.
- The NGINX server runs on port 80 by default so the line `EXPOSE 80` is added.
- The last line is a `COPY` instruction. The `--from=builder` part indicates that you want to copy some files from the `builder` stage. After that it's a standard copy instruction where `/app/dist` is the source and `/usr/share/nginx/html` is the destination. The destination used here is the default site path for NGINX so any static file you put inside there will be automatically served.

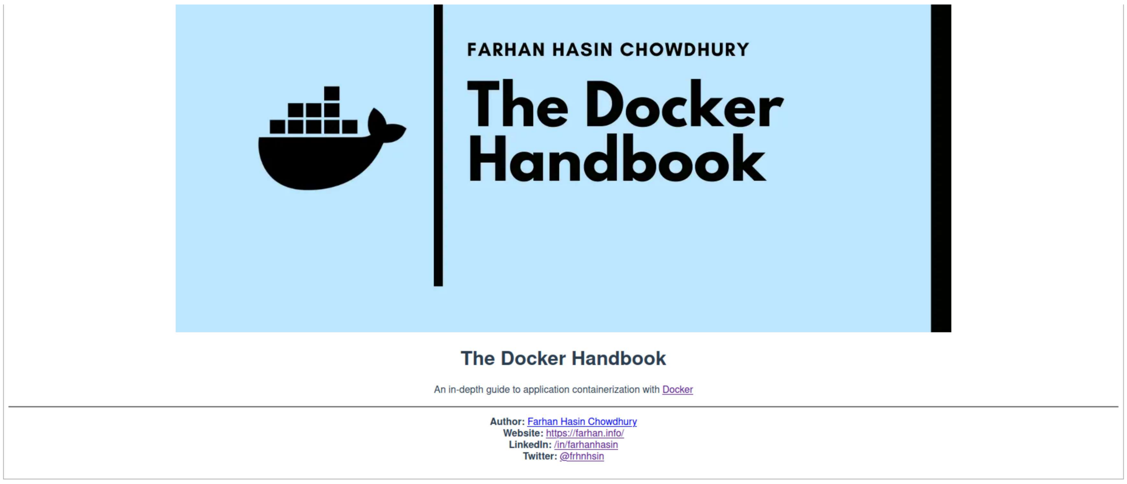
As you can see, the resulting image is a `nginx` base image containing only the files necessary for running the application. To build this image execute the following command:

```
# eea03077c87e: Pull complete
# eba7631b45c5: Pull complete
# Digest: sha256:2eea9f5d6fff078ad6cc6c961ab11b8314efd91fb8480b5
# Status: Downloaded newer image for nginx:stable
#  ---> 05f64a802c26
# Step 9/9 : COPY --from=builder /app/dist /usr/share/nginx/html
#  ---> 8c6dfc34a10d
# Successfully built 8c6dfc34a10d
# Successfully tagged hello-dock:prod
```

Once the image has been built, you may run a new container by executing the following command:

```
docker container run \
    --rm \
    --detach \
    --name hello-dock-prod \
    --publish 8080:80 \
    hello-dock:prod

# 224aaba432bb09aca518fdd0365875895c2f5121eb668b2e7b2d5a99c019b9
```

The running application should be available on `http://127.0.0.1:8080`:



Here you can see my `hello-dock` application in all its glory. Multi-staged builds can be very useful if you're building large applications with a lot of dependencies. If configured properly, images built in multiple stages can be very optimized and compact.

How to Ignore Unnecessary Files

If you've been working with `git` for some time now, you may know about the `.gitignore` files in projects. These contain a list of files and directories to be excluded from the repository.

Well, Docker has a similar concept. The `.dockerignore` file contains a list of files and directories to be excluded from image builds. You can find a pre-created `.dockerignore` file in the `hello-dock` directory.

```
.git
*Dockerfile*
*docker-compose*
node_modules
```

effect. I've added `.dockerignore` files where necessary in the project repository.

Network Manipulation Basics in Docker

So far in this book, you've only worked with single container projects. But in real life, the majority of projects that you'll have to work with will have more than one container. And to be honest, working with a bunch of containers can be a little difficult if you don't understand the nuances of container isolation.

So in this section of the book, you'll get familiar with basic networking with Docker and you'll work hands on with a small multi-container project.

Well you've already learned in the previous section that containers are isolated environments. Now consider a scenario where you have a `notes-api` application powered by Express.js and a PostgreSQL database server running in two separate containers.

These two containers are completely isolated from each other and are oblivious to each other's existence. **So how do you connect the two? Won't that be a challenge?**

You may think of two possible solutions to this problem. They are as follows:

- Accessing the database server using an exposed port.

The first one involves exposing a port from the `postgres` container and the `notes-api` will connect through that. Assume that the exposed port from the `postgres` container is 5432. Now if you try to connect to `127.0.0.1:5432` from inside the `notes-api` container, you'll find that the `notes-api` can't find the database server at all.

The reason is that when you're saying `127.0.0.1` inside the `notes-api` container, you're simply referring to the `localhost` of that container and that container only. The `postgres` server simply doesn't exist there. As a result the `notes-api` application failed to connect.

The second solution you may think of is finding the exact IP address of the `postgres` container using the `container inspect` command and using that with the port. Assuming the name of the `postgres` container is `notes-api-db-server` you can easily get the IP address by executing the following command:

```
docker container inspect --format='{{range .NetworkSettings.Netw

#  172.17.0.2
```

Now given that the default port for `postgres` is `5432`, you can very easily access the database server by connecting to `172.17.0.2:5432` from the `notes-api` container.

There are problems in this approach as well. Using IP addresses to refer to a container is not recommended. Also, if the container gets

Now that I've dismissed the possible wrong answers to the original question, the correct answer is, **you connect them by putting them under a user-defined bridge network.**

Docker Network Basics

A network in Docker is another logical object like a container and image. Just like the other two, there is a plethora of commands under the `docker network` group for manipulating networks.

To list out the networks in your system, execute the following command:

```
docker network ls

# NETWORK ID     NAME      DRIVER    SCOPE
# c2e59f2b96bd   bridge    bridge    local
# 124dccee067f   host      host      local
# 506e3822bf1f   none      null      local
```

You should see three networks in your system. Now look at the `DRIVER` column of the table here. These drivers are can be treated as the type of network.

By default, Docker has five networking drivers. They are as follows:

- `bridge` - The default networking driver in Docker. This can be used when multiple containers are running in standard mode and need to communicate with each other.
- `host` - Removes the network isolation completely. Any container running under a `host` network is basically

altogether. I haven't found any use-case for this yet.

- `overlay` - This is used for connecting multiple Docker daemons across computers and is out of the scope of this book.
- `macvlan` - Allows assignment of MAC addresses to containers, making them function like physical devices in a network.

There are also third-party plugins that allow you to integrate Docker with specialized network stacks. Out of the five mentioned above, you'll only work with the `bridge` networking driver in this book.

How to Create a User-Defined Bridge in Docker

Before you start creating your own bridge, I would like to take some time to discuss the default bridge network that comes with Docker. Let's begin by listing all the networks on your system:

```
docker network ls

# NETWORK ID     NAME      DRIVER    SCOPE
# c2e59f2b96bd   bridge    bridge    local
# 124dccee067f   host      host      local
# 506e3822bf1f   none      null      local
```

As you can see, Docker comes with a default bridge network named `bridge`. Any container you run will be automatically attached to this bridge network:

```
docker network inspect --format='{{range .Containers}}{{.Name}}{
# hello-dock
```

Containers attached to the default bridge network can communicate with each others using IP addresses which I have already discouraged in the previous sub-section.

A user-defined bridge, however, has some extra features over the default one. According to the official docs on this topic, some notable extra features are as follows:

- **User-defined bridges provide automatic DNS resolution between containers:** This means containers attached to the same network can communicate with each others using the container name. So if you have two containers named `notes-api` and `notes-db` the API container will be able to connect to the database container using the `notes-db` name.
- **User-defined bridges provide better isolation:** All containers are attached to the default bridge network by default which can cause conflicts among them. Attaching containers to a user-defined bridge can ensure better isolation.
- **Containers can be attached and detached from user-defined networks on the fly:** During a container's lifetime, you can connect or disconnect it from user-defined networks on the fly. To remove a container from the default bridge network, you need to stop the container and recreate it with different network options.

command is as follows:

```
docker network create <network name>
```

To create a network with the name `skynet` execute the following command:

```
docker network create skynet

# 7bd5f351aa892ac6ec15fed8619fc3bbb95a7dcdd58980c28304627c8f7eb0

docker network ls

# NETWORK ID     NAME      DRIVER    SCOPE
# be0cab667c4b   bridge    bridge    local
# 124dccee067f   host      host      local
# 506e3822bf1f   none      null      local
# 7bd5f351aa89   skynet    bridge    local
```

As you can see a new network has been created with the given name. No container is currently attached to this network. In the next sub-section, you'll learn about attaching containers to a network.

How to Attach a Container to a Network in Docker

There are mostly two ways of attaching a container to a network. First, you can use the network connect command to attach a

```
docker network connect <network identifier> <container identifie
```

To connect the `hello-dock` container to the `skynet` network, you can execute the following command:

```
docker network connect skynet hello-dock

docker network inspect --format='{{range .Containers}} {{.Name}}

#  hello-dock

docker network inspect --format='{{range .Containers}} {{.Name}}

#  hello-dock
```

As you can see from the outputs of the two `network inspect` commands, the `hello-dock` container is now attached to both the `skynet` and the default `bridge` network.

The second way of attaching a container to a network is by using the `--network` option for the `container run` or `container create` commands. The generic syntax for the option is as follows:

```
--network <network identifier>
```

```
docker container run --network skynet --rm --name alpine-box -it

# lands you into alpine linux shell

/ # ping hello-dock

# PING hello-dock (172.18.0.2): 56 data bytes
# 64 bytes from 172.18.0.2: seq=0 ttl=64 time=0.191 ms
# 64 bytes from 172.18.0.2: seq=1 ttl=64 time=0.103 ms
# 64 bytes from 172.18.0.2: seq=2 ttl=64 time=0.139 ms
# 64 bytes from 172.18.0.2: seq=3 ttl=64 time=0.142 ms
# 64 bytes from 172.18.0.2: seq=4 ttl=64 time=0.146 ms
# 64 bytes from 172.18.0.2: seq=5 ttl=64 time=0.095 ms
# 64 bytes from 172.18.0.2: seq=6 ttl=64 time=0.181 ms
# 64 bytes from 172.18.0.2: seq=7 ttl=64 time=0.138 ms
# 64 bytes from 172.18.0.2: seq=8 ttl=64 time=0.158 ms
# 64 bytes from 172.18.0.2: seq=9 ttl=64 time=0.137 ms
# 64 bytes from 172.18.0.2: seq=10 ttl=64 time=0.145 ms
# 64 bytes from 172.18.0.2: seq=11 ttl=64 time=0.138 ms
# 64 bytes from 172.18.0.2: seq=12 ttl=64 time=0.085 ms

--- hello-dock ping statistics ---
13 packets transmitted, 13 packets received, 0% packet loss
round-trip min/avg/max = 0.085/0.138/0.191 ms
```

As you can see, running `ping hello-dock` from inside the `alpine-box` container works because both of the containers are under the same user-defined bridge network and automatic DNS resolution is working.

Keep in mind, though, that in order for the automatic DNS resolution to work you must assign custom names to the containers. Using the randomly generated name will not work.

In the previous sub-section you learned about attaching containers to a network. In this sub-section, you'll learn about how to detach them.

You can use the `network disconnect` command for this task. The generic syntax for the command is as follows:

```
docker network disconnect <network identifier> <container identi
```

To detach the `hello-dock` container from the `skynet` network, you can execute the following command:

```
docker network disconnect skynet hello-dock
```

Just like the `network connect` command, the `network disconnect` command doesn't give any output.

How to Get Rid of Networks in Docker

Just like the other logical objects in Docker, networks can be removed using the `network rm` command. The generic syntax for the command is as follows:

```
docker network rm <network identifier>
```

```
docker network rm skynet
```

You can also use the `network prune` command to remove any unused networks from your system. The command also has the `-f` or `--force` and `-a` or `--all` options.

How to Containerize a Multi-Container JavaScript Application

Now that you've learned enough about networks in Docker, in this section you'll learn to containerize a full-fledged multi-container project. The project you'll be working with is a simple `notes-api` powered by Express.js and PostgreSQL.

In this project there are two containers in total that you'll have to connect using a network. Apart from this, you'll also learn about concepts like environment variables and named volumes. So without further ado, let's jump right in.

How to Run the Database Server

The database server in this project is a simple PostgreSQL server and uses the official postgres image.

According to the official docs, in order to run a container with this image, you must provide the `POSTGRES_PASSWORD` environment variable. Apart from this one, I'll also provide a name for the default database using the `POSTGRES_DB` environment variable. PostgreSQL by default listens on port `5432`, so you need to publish that as well.

```
docker container run \
    --detach \
    --name=notes-db \
    --env POSTGRES_DB=notesdb \
    --env POSTGRES_PASSWORD=secret \
    --network=notes-api-network \
    postgres:12

# a7b287d34d96c8e81a63949c57b83d7c1d71b5660c87f5172f074bd1606196

docker container ls

# CONTAINER ID   IMAGE         COMMAND                  CREATED
# a7b287d34d96   postgres:12   "docker-entrypoint.s…"   About a
```

The `--env` option for the `container run` and `container create` commands can be used for providing environment variables to a container. As you can see, the database container has been created successfully and is running now.

Although the container is running, there is a small problem. Databases like PostgreSQL, MongoDB, and MySQL persist their data in a directory. PostgreSQL uses the `/var/lib/postgresql/data` directory inside the container to persist data.

Now what if the container gets destroyed for some reason? You'll lose all your data. To solve this problem, a named volume can be used.

How to Work with Named Volumes in Docker

except that you can refer to a named volume using its name.

Volumes are also logical objects in Docker and can be manipulated using the command-line. The `volume create` command can be used for creating a named volume.

The generic syntax for the command is as follows:

```
docker volume create <volume name>
```

To create a volume named `notes-db-data` you can execute the following command:

```
docker volume create notes-db-data

# notes-db-data

docker volume ls

# DRIVER    VOLUME NAME
# local     notes-db-data
```

This volume can now be mounted to `/var/lib/postgresql/data` inside the `notes-db` container. To do so, stop and remove the `notes-db` container:

```
docker container stop notes-db

# notes-db
```

Now run a new container and assign the volume using the `--volume` or `-v` option.

```
docker container run \
    --detach \
    --volume notes-db-data:/var/lib/postgresql/data \
    --name=notes-db \
    --env POSTGRES_DB=notesdb \
    --env POSTGRES_PASSWORD=secret \
    --network=notes-api-network \
    postgres:12

# 37755e86d62794ed3e67c19d0cd1eba431e26ab56099b92a3456908c1d3467
```

Now inspect the `notes-db` container to make sure that the mounting was successful:

```
docker container inspect --format='{{range .Mounts}} {{ .Name }}

#  notes-db-data
```

Now the data will safely be stored inside the `notes-db-data` volume and can be reused in the future. A bind mount can also be used instead of a named volume here, but I prefer a named volume in such scenarios.

How to Access Logs from a Container in Docker

```
docker container logs <container identifier>
```

To access the logs from the `notes-db` container, you can execute the following command:

```
#
#
# /usr/local/bin/docker-entrypoint.sh: ignoring /docker-entrypoi
#
# 2021-01-25 13:39:22.008 UTC [47] LOG:  received fast shutdown
# waiting for server to shut down....2021-01-25 13:39:22.015 UTC
# 2021-01-25 13:39:22.017 UTC [47] LOG:  background worker "logi
# 2021-01-25 13:39:22.017 UTC [49] LOG:  shutting down
# 2021-01-25 13:39:22.056 UTC [47] LOG:  database system is shut
#  done
# server stopped
#
# PostgreSQL init process complete; ready for start up.
#
# 2021-01-25 13:39:22.135 UTC [1] LOG:  starting PostgreSQL 12.5
# 2021-01-25 13:39:22.136 UTC [1] LOG:  listening on IPv4 addres
# 2021-01-25 13:39:22.136 UTC [1] LOG:  listening on IPv6 addres
# 2021-01-25 13:39:22.147 UTC [1] LOG:  listening on Unix socket
# 2021-01-25 13:39:22.177 UTC [75] LOG:  database system was shu
# 2021-01-25 13:39:22.190 UTC [1] LOG:  database system is ready
```

Evident by the text in line 57, the database is up and ready to accept connections from the outside. There is also the `--follow` or `-f` option for the command which lets you attach the console to the logs output and get a continuous stream of text.

How to Create a Network and Attaching the Database Server in Docker

As you've learned in the previous section, the containers have to be attached to a user-defined bridge network in order to communicate with each other using container names. To do so, create a network named `notes-api-network` in your system:

Now attach the `notes-db` container to this network by executing the following command:

```
docker network connect notes-api-network notes-db
```

How to Write the Dockerfile

Go to the directory where you've cloned the project code. Inside there, go inside the `notes-api/api` directory, and create a new `Dockerfile`. Put the following code in the file:

This is a multi-staged build. The first stage is used for building and installing the dependencies using `node-gyp` and the second stage is for running the application. I'll go through the steps briefly:

- Stage 1 uses `node:lts-alpine` as its base and uses `builder` as the stage name.
- On line 5, we install `python`, `make`, and `g++`. The `node-gyp` tool requires these three packages to run.
- On line 7, we set `/app` directory as the `WORKDIR`.
- On line 9 and 10, we copy the `package.json` file to the `WORKDIR` and install all the dependencies.
- Stage 2 also uses `node-lts:alpine` as the base.
- On line 16, we set the `NODE_ENV` environment variable to `production`. This is important for the API to run properly.
- From line 18 to line 20, we set the default user to `node`, create the `/home/node/app` directory, and set that as the `WORKDIR`.
- On line 22, we copy all the project files and on line 23 we copy the `node_modules` directory from the `builder` stage. This directory contains all the built dependencies necessary for running the application.
- On line 25, we set the default command.

To build an image from this `Dockerfile`, you can execute the following command:

```
#  ---> 471e8b4eb0b2
# Step 7/14 : EXPOSE 3000
#  ---> Running in 4ea24f871747
# Removing intermediate container 4ea24f871747
#  ---> 1f0206f2f050
# Step 8/14 : ENV NODE_ENV=production
#  ---> Running in 5d40d6ac3b7e
# Removing intermediate container 5d40d6ac3b7e
#  ---> 31f62da17929
# Step 9/14 : USER node
#  ---> Running in 0963e1fb19a0
# Removing intermediate container 0963e1fb19a0
#  ---> 0f4045152b1c
# Step 10/14 : RUN mkdir -p /home/node/app
#  ---> Running in 0ac591b3adbd
# Removing intermediate container 0ac591b3adbd
#  ---> 5908373dfc75
# Step 11/14 : WORKDIR /home/node/app
#  ---> Running in 55253b62ff57
# Removing intermediate container 55253b62ff57
#  ---> 2883cdb7c77a
# Step 12/14 : COPY . .
#  ---> 8e60893a7142
# Step 13/14 : COPY --from=builder /app/node_modules  /home/node
#  ---> 27a85faa4342
# Step 14/14 : CMD [ "node", "bin/www" ]
#  ---> Running in 349c8ca6dd3e
# Removing intermediate container 349c8ca6dd3e
#  ---> 9ea100571585
# Successfully built 9ea100571585
# Successfully tagged notes-api:latest
```

Before you run a container using this image, make sure the database container is running, and is attached to the `notes-api-network`.

```
#                       "EndpointID": "5b8f8718ec9a5ec53e7a13cce3c
#                       "Gateway": "172.18.0.1",
#                       "IPAddress": "172.18.0.2",
#                       "IPPrefixLen": 16,
#                       "IPv6Gateway": "",
#                       "GlobalIPv6Address": "",
#                       "GlobalIPv6PrefixLen": 0,
#                       "MacAddress": "02:42:ac:12:00:02",
#                       "DriverOpts": {}
#                   }
#               }
#           }
#       }
# ]
```

I've shortened the output for easy viewing here. On my system, the `notes-db` container is running, uses the `notes-db-data` volume, and is attached to the `notes-api-network` bridge.

Once you're assured that everything is in place, you can run a new container by executing the following command:

```
docker container run \
    --detach \
    --name=notes-api \
    --env DB_HOST=notes-db \
    --env DB_DATABASE=notesdb \
    --env DB_PASSWORD=secret \
    --publish=3000:3000 \
    --network=notes-api-network \
    notes-api

# f9ece420872de99a060b954e3c236cbb1e23d468feffa7fed1e06985d99fb9
```

The `notes-api` application requires three environment variables to be set. They are as follows:

- `DB_HOST` - This is the host of the database server. Given that both the database server and the API are attached to the same user-defined bridge network, the database server can be refereed to using its container name which is `notes-db` in this case.
- `DB_DATABASE` - The database that this API will use. On Running the Database Server we set the default database name to `notesdb` using the `POSTGRES_DB` environment variable. We'll use that here.
- `DB_PASSWORD` - Password for connecting to the database. This was also set on Running the Database Server sub-section using the `POSTGRES_PASSWORD` environment variable.

To check if the container is running properly or not, you can use the `container ls` command:

```
docker container ls

# CONTAINER ID   IMAGE          COMMAND                  CREATED
# f9ece420872d   notes-api      "docker-entrypoint.s…"   12 minut
# 37755e86d627   postgres:12    "docker-entrypoint.s…"   17 hours
```

The container is running now. You can visit `http://127.0.0.1:3000/` to see the API in action.

The API has five routes in total that you can see inside the `/notes-api/api/api/routes/notes.js` file.

Although the container is running, there is one last thing that you'll have to do before you can start using it. You'll have to run the database migration necessary for setting up the database tables, and you can do that by executing `npm run db:migrate` command inside the container.

How to Execute Commands in a Running Container

You've already learned about executing commands in a stopped container. Another scenario is executing a command inside a running container.

For this, you'll have to use the `exec` command to execute a custom command inside a running container.

The generic syntax for the `exec` command is as follows:

To execute `npm run db:migrate` inside the `notes-api` container, you can execute the following command:

```
docker container exec notes-api npm run db:migrate

# > notes-api@ db:migrate /home/node/app
# > knex migrate:latest
#
# Using environment: production
# Batch 1 run: 1 migrations
```

In cases where you want to run an interactive command inside a running container, you'll have to use the `-it` flag. As an example, if you want to access the shell running inside the `notes-api` container, you can execute following the command:

```
docker container exec -it notes-api sh

# / # uname -a
# Linux b5b1367d6b31 5.10.9-201.fc33.x86_64 #1 SMP Wed Jan 20 16
```

How to Write Management Scripts in Docker

Managing a multi-container project along with the network and volumes and stuff means writing a lot of commands. To simplify the process, I usually have help from simple shell scripts and a Makefile.

- `boot.sh` - Used for starting the containers if they already exist.
- `build.sh` - Creates and runs the containers. It also creates the images, volumes, and networks if necessary.
- `destroy.sh` - Removes all containers, volumes and networks associated with this project.
- `stop.sh` - Stops all running containers.

There is also a `Makefile` that contains four targets named `start`, `stop`, `build` and `destroy`, each invoking the previously mentioned shell scripts.

If the container is in a running state in your system, executing `make stop` should stop all the containers. Executing `make destroy` should stop the containers and remove everything. Make sure you're running the scripts inside the `notes-api` directory:

```
make destroy

# ./shutdown.sh
# stopping api container --->
# notes-api
# api container stopped --->

# stopping db container --->
# notes-db
# db container stopped --->

# shutdown script finished

# ./destroy.sh
# removing api container --->
# notes-api
# api container removed --->
```

```
# removing db data volume --->
# notes-db-data
# db data volume removed --->

# removing network --->
# notes-api-network
# network removed --->

# destroy script finished
```

If you're getting a permission denied error, than execute `chmod +x` on the scripts:

```
chmod +x boot.sh build.sh destroy.sh shutdown.sh
```

I'm not going to explain these scripts because they're simple `if-else` statements along with some Docker commands that you've already seen many times. If you have some understanding of the Linux shell, you should be able to understand the scripts as well.

How to Compose Projects Using Docker-Compose

In the previous section, you've learned about managing a multi-container project and the difficulties of it. Instead of writing so many commands, there is an easier way to manage multi-container projects, a tool called Docker Compose.

According to the Docker documentation -

> configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Although Compose works in all environments, it's more focused on development and testing. Using Compose on a production environment is not recommended at all.

Docker Compose Basics

Go the directory where you've cloned the repository that came with this book. Go inside the `notes-api/api` directory and create a `Dockerfile.dev` file. Put the following code in it:

The code is almost identical to the `Dockerfile` that you worked with in the previous section. The three differences in this file are as follows:

- On line 10, we run `npm install` instead of `npm run install --only=prod` because we want the development dependencies also.
- On line 15, we set the `NODE_ENV` environment variable to `development` instead of `production`.
- On line 24, we use a tool called nodemon to get the hot-reload feature for the API.

You already know that this project has two containers:

- `notes-db` - A database server powered by PostgreSQL.
- `notes-api` - A REST API powered by Express.js

In the world of Compose, each container that makes up the application is known as a service. The first step in composing a multi-container project is to define these services.

Just like the Docker daemon uses a `Dockerfile` for building images, Docker Compose uses a `docker-compose.yaml` file to read service definitions from.

Head to the `notes-api` directory and create a new `docker-compose.yaml` file. Put the following code into the newly created file:

```
    db:
        image: postgres:12
        container_name: notes-db-dev
        volumes:
            - notes-db-dev-data:/var/lib/postgresql/data
        environment:
            POSTGRES_DB: notesdb
            POSTGRES_PASSWORD: secret
    api:
        build:
            context: ./api
            dockerfile: Dockerfile.dev
        image: notes-api:dev
        container_name: notes-api-dev
        environment:
            DB_HOST: db ## same as the database service name
            DB_DATABASE: notesdb
            DB_PASSWORD: secret
        volumes:
            - /home/node/app/node_modules
            - ./api:/home/node/app
        ports:
            - 3000:3000

volumes:
    notes-db-dev-data:
        name: notes-db-dev-data
```

Every valid `docker-compose.yaml` file starts by defining the file version. At the time of writing, `3.8` is the latest version. You can look up the latest version here.

Blocks in an YAML file are defined by indentation. I will go through each of the blocks and will explain what they do.

- The `services` block holds the definitions for each of the services or containers in the application. `db` and `api` are the two services that comprise this project.

run a container. For the `db` service we're using the official PostgreSQL image.

- Unlike the `db` service, a pre-built image for the `api` service doesn't exist. So we'll use the `Dockerfile.dev` file.
- The `volumes` block defines any name volume needed by any of the services. At the time it only enlists `notes-db-dev-data` volume used by the `db` service.

Now that have a high level overview of the `docker-compose.yaml` file, let's have a closer look at the individual services.

The definition code for the `db` service is as follows:

```
db:
    image: postgres:12
    container_name: notes-db-dev
    volumes:
        - db-data:/var/lib/postgresql/data
    environment:
        POSTGRES_DB: notesdb
        POSTGRES_PASSWORD: secret
```

- The `image` key holds the image repository and tag used for this container. We're using the `postgres:12` image for running the database container.
- The `container_name` indicates the name of the container. By default containers are named following <project directory name>_<service name> syntax. You can override that using `container_name`.

identical to what you've seen before.

- The `environment` map holds the values of the various environment variables needed for the service.

Definition code for the `api` service is as follows:

```
api:
    build:
        context: ./api
        dockerfile: Dockerfile.dev
    image: notes-api:dev
    container_name: notes-api-dev
    environment:
        DB_HOST: db ## same as the database service name
        DB_DATABASE: notesdb
        DB_PASSWORD: secret
    volumes:
        - /home/node/app/node_modules
        - ./api:/home/node/app
    ports:
        - 3000:3000
```

- The `api` service doesn't come with a pre-built image. Instead it has a build configuration. Under the `build` block we define the context and the name of the Dockerfile for building an image. You should have an understanding of context and Dockerfile by now so I won't spend time explaining those.
- The `image` key holds the name of the image to be built. If not assigned, the image will be named following the <project directory name>_<service name> syntax.

the `db` here, will be replaced by the IP address of the `api` service container. The `DB_DATABASE` and `DB_PASSWORD` variables have to match up with `POSTGRES_DB` and `POSTGRES_PASSWORD` respectively from the `db` service definition.

- In the `volumes` map, you can see an anonymous volume and a bind mount described. The syntax is identical to what you've seen in previous sections.
- The `ports` map defines any port mapping. The syntax, `<host port>:<container port>` is identical to the `--publish` option you used before.

Finally, the code for the `volumes` is as follows:

```
volumes:
    db-data:
        name: notes-db-dev-data
```

Any named volume used in any of the services has to be defined here. If you don't define a name, the volume will be named following the `<project directory name>_<volume key>` and the key here is `db-data`.

You can learn about the different options for volume configuration in the official docs.

How to Start Services in Docker Compose

command builds any missing images, creates containers, and starts them in one go.

Before you execute the command, though, make sure you've opened your terminal in the same directory where the `docker-compose.yaml` file is. This is very important for every `docker-compose` command you execute.

```
# Removing intermediate container 3ae30e6fb8b8
#  ---> c391cee4b92c
# Step 10/13 : WORKDIR /home/node/app
#  ---> Running in 6aa27f6b50c1
# Removing intermediate container 6aa27f6b50c1
#  ---> 761a7435dbca
# Step 11/13 : COPY . .
#  ---> b5d5c5bdf3a6
# Step 12/13 : COPY --from=builder /app/node_modules /home/node/a
#  ---> 9e1a19960420
# Step 13/13 : CMD [ "./node_modules/.bin/nodemon", "--config",
#  ---> Running in 5bdd62236994
# Removing intermediate container 5bdd62236994
#  ---> 548e178f1386
# Successfully built 548e178f1386
# Successfully tagged notes-api:dev
# Creating notes-api-dev ... done
# Creating notes-db-dev  ... done
```

The `--detach` or `-d` option here functions the same as the one you've seen before. The `--file` or `-f` option is only needed if the YAML file is not named `docker-compose.yaml` (but I've used here for demonstration purposes).

Apart from the the `up` command there is the `start` command. The main difference between these two is that the `start` command doesn't create missing containers, only starts existing containers. It's basically the same as the `container start` command.

The `--build` option for the `up` command forces a rebuild of the images. There are some other options for the `up` command that you can see in the official docs.

How to List Services in Docker Compose

```
docker-compose ps

#     Name                       Command                   State
# -----------------------------------------------------------------
# notes-api-dev   docker-entrypoint.sh ./nod ...   Up      0.0.0
# notes-db-dev    docker-entrypoint.sh postgres    Up      5432/
```

It's not as informative as the `container ls` output, but it's useful when you have tons of containers running simultaneously.

How to Execute Commands Inside a Running Service in Docker Compose

I hope you remember from the previous section that you have to run some migration scripts to create the database tables for this API.

Just like the `container exec` command, there is an `exec` command for `docker-compose`. Generic syntax for the command is as follows:

```
docker-compose exec <service name> <command>
```

To execute the `npm run db:migrate` command inside the `api` service, you can execute the following command:

```
docker-compose exec api npm run db:migrate
```

```
# Batch 1 run: 1 migrations
```

Unlike the `container exec` command, you don't need to pass the `-it` flag for interactive sessions. `docker-compose` does that automatically.

How to Access Logs from a Running Service in Docker Compose

You can also use the `logs` command to retrieve logs from a running service. The generic syntax for the command is as follows:

```
docker-compose logs <service name>
```

To access the logs from the `api` service, execute the following command:

This is just a portion from the log output. You can kind of hook into the output stream of the service and get the logs in real-time by using the `-f` or `--follow` option. Any later log will show up instantly in the terminal as long as you don't exit by pressing `ctrl + c` or closing the window. The container will keep running even if you exit out of the log window.

How to Stop Services in Docker Compose

To stop services, there are two approaches that you can take. The first one is the `down` command. The `down` command stops all running containers and removes them from the system. It also removes any networks:

```
docker-compose down --volumes

# Stopping notes-api-dev ... done
# Stopping notes-db-dev  ... done
# Removing notes-api-dev ... done
# Removing notes-db-dev  ... done
# Removing network notes-api_default
# Removing volume notes-db-dev-data
```

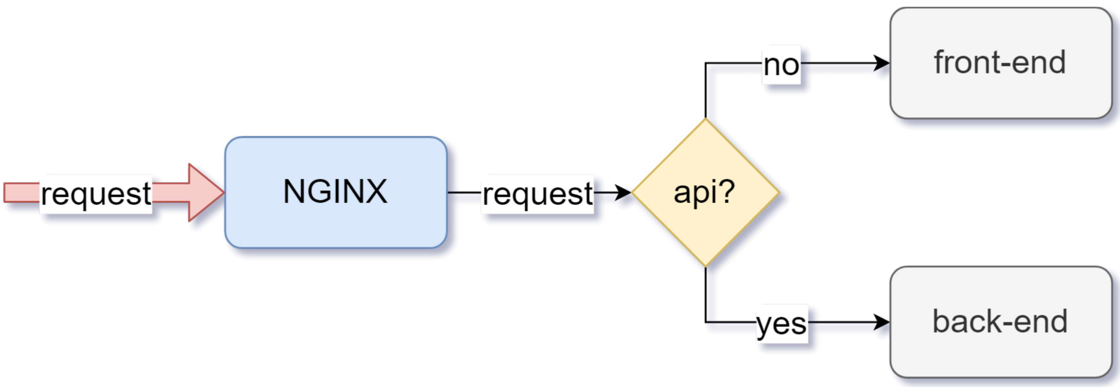
The `--volumes` option indicates that you want to remove any named volume(s) defined in the `volumes` block. You can learn about the additional options for the `down` command in the official docs.

Another command for stopping services is the `stop` command which functions identically to the `container stop` command. It stops all the containers for the application and keeps them. These containers can later be started with the `start` or `up` command.

In this sub-section, we'll be adding a front-end to our notes API and turning it into a complete full-stack application. I won't be explaining any of the `Dockerfile.dev` files in this sub-section (except the one for the `nginx` service) as they are identical to some of the others you've already seen in previous sub-sections.

If you've cloned the project code repository, then go inside the `fullstack-notes-application` directory. Each directory inside the project root contains the code for each service and the corresponding `Dockerfile`.

Before we start with the `docker-compose.yaml` file let's look at a diagram of how the application is going to work:



Instead of accepting requests directly like we previously did, in this application all the requests will be first received by an NGINX (lets call it router) service.

The router will then see if the requested end-point has `/api` in it. If yes, the router will route the request to the back-end or if not, the router will route the request to the front-end.

expected and the front-end application fails to find the `api` service.

NGINX, on the other hand, runs inside a container and can communicate with the different services across the entire application.

I will not get into the configuration of NGINX here. That topic is kinda out of the scope of this book. But if you want to have a look at it, go ahead and check out the `/notes-api/nginx/development.conf` and `/notes-api/nginx/production.conf` files. Code for the `/notes-api/nginx/Dockerfile.dev` is as follows:

```
FROM nginx:stable-alpine

COPY ./development.conf /etc/nginx/conf.d/default.conf
```

All it does is copy the configuration file to `/etc/nginx/conf.d/default.conf` inside the container.

Let's start writing the `docker-compose.yaml` file. Apart from the `api` and `db` services there will be the `client` and `nginx` services. There will also be some network definitions that I'll get into shortly.

```
version: "3.8"

services:
    db:
        image: postgres:12
        container_name: notes-db-dev
        volumes:
            - db-data:/var/lib/postgresql/data
```

```
            - backend
    api:
        build:
            context: ./api
            dockerfile: Dockerfile.dev
        image: notes-api:dev
        container_name: notes-api-dev
        volumes:
            - /home/node/app/node_modules
            - ./api:/home/node/app
        environment:
            DB_HOST: db ## same as the database service name
            DB_PORT: 5432
            DB_USER: postgres
            DB_DATABASE: notesdb
            DB_PASSWORD: secret
        networks:
            - backend
    client:
        build:
            context: ./client
            dockerfile: Dockerfile.dev
        image: notes-client:dev
        container_name: notes-client-dev
        volumes:
            - /home/node/app/node_modules
            - ./client:/home/node/app
        networks:
            - frontend
    nginx:
        build:
            context: ./nginx
            dockerfile: Dockerfile.dev
        image: notes-router:dev
        container_name: notes-router-dev
        restart: unless-stopped
        ports:
            - 8080:80
        networks:
            - backend
            - frontend

volumes:
    db-data:
        name: notes-db-dev-data
```

```
        driver: bridge
    backend:
        name: fullstack-notes-application-network-backend
        driver: bridge
```

The file is almost identical to the previous one you worked with. The only thing that needs some explanation is the network configuration. The code for the `networks` block is as follows:

```
networks:
    frontend:
        name: fullstack-notes-application-network-frontend
        driver: bridge
    backend:
        name: fullstack-notes-application-network-backend
        driver: bridge
```

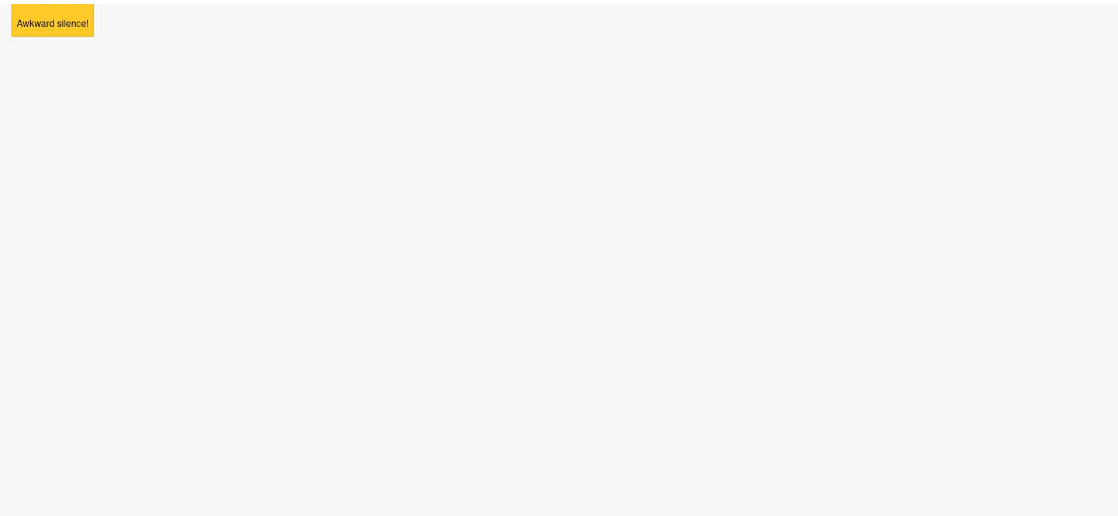
I've defined two bridge networks. By default, Compose creates a bridge network and attaches all containers to that. In this project, however, I wanted proper network isolation. So I defined two networks, one for the front-end services and one for the back-end services.

I've also added `networks` block in each of the service definitions. This way the the `api` and `db` service will be attached to one network and the `client` service will be attached to a separate network. But the `nginx` service will be attached to both the networks so that it can perform as router between the front-end and back-end services.

Start all the services by executing the following command:

```
# Building client
# Sending build context to Docker daemon  43.01kB
#
# Step 1/7 : FROM node:lts-alpine
#  ---> 471e8b4eb0b2
# Step 2/7 : USER node
#  ---> Using cache
#  ---> 4be5fb31f862
# Step 3/7 : RUN mkdir -p /home/node/app
#  ---> Using cache
#  ---> 1fefc7412723
# Step 4/7 : WORKDIR /home/node/app
#  ---> Using cache
#  ---> d1470d878aa7
# Step 5/7 : COPY ./package.json .
#  ---> Using cache
#  ---> bbcc49475077
# Step 6/7 : RUN npm install
#  ---> Using cache
#  ---> 860a4a2af447
# Step 7/7 : CMD [ "npm", "run", "serve" ]
#  ---> Using cache
#  ---> 11db51d5bee7
# Successfully built 11db51d5bee7
# Successfully tagged notes-client:dev
# Building nginx
# Sending build context to Docker daemon   5.12kB
#
# Step 1/2 : FROM nginx:stable-alpine
#  ---> f2343e2e2507
# Step 2/2 : COPY ./development.conf /etc/nginx/conf.d/default.c
#  ---> Using cache
#  ---> 02a55d005a98
# Successfully built 02a55d005a98
# Successfully tagged notes-router:dev
# Creating notes-client-dev ... done
# Creating notes-api-dev    ... done
# Creating notes-router-dev ... done
# Creating notes-db-dev     ... done
```

Now visit `http://localhost:8080` and voilà!



Try adding and deleting notes to see if the application works properly. The project also comes with shell scripts and a `Makefile`. Explore them to see how you can run this project without the help of `docker-compose` like you did in the previous section.

Conclusion

I would like to thank you from the bottom of my heart for the time you've spent reading this book. I hope you've enjoyed it and have learned all the essentials of Docker.

Apart from this one, I've written full-length handbooks on other complicated topics available for free on freeCodeCamp.

These handbooks are part of my mission to simplify hard to understand technologies for everyone. Each of these handbooks takes a lot of time and effort to write.

If you've enjoyed my writing and want to keep me motivated, consider leaving starts on GitHub and endorse me for relevant skills

I'm always open to suggestions and discussions on Twitter or LinkedIn. Hit me with direct messages.

In the end, consider sharing the resources with others, because

> Sharing knowledge is the most fundamental act of friendship. Because it is a way you can give something without loosing something. — Richard Stallman

Till the next one, stay safe and keep learning.

Farhan Hasin Chowdhury

Software developer with a knack for learning new things and writing about them

---

If you read this far, tweet to the author to show them you care.

Tweet a thanks

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

services, and staff.

You can make a tax-deductible donation here.