Table of Contents

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 can be run without going through a troublesome setup process.

Let's consider a real life scenario here. Assume you have developed 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.
On Windows, the C++ build tools package measures at gigabytes and takes quite some time to install. On a Mac, you can either install the gigantic Xcode application or the much smaller Command Line 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 authorization.

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.

How to Install Docker

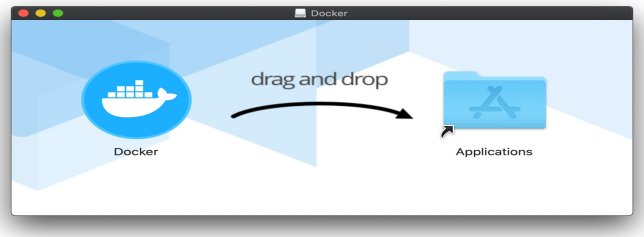
Installation of Docker varies greatly depending on the operating system you're using. But it's universally simple across the board.

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.

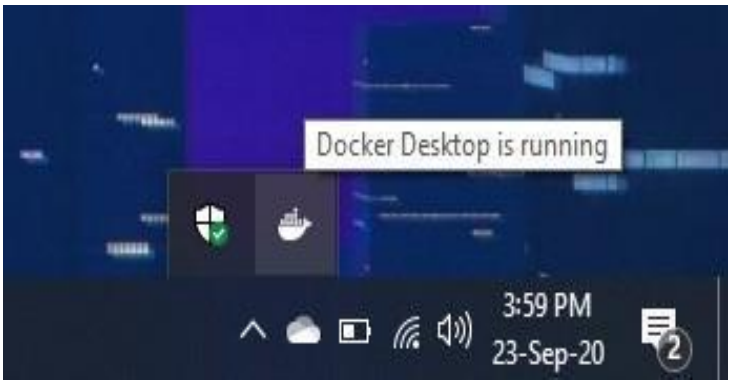
Now, open up the terminal and execute `docker --version` to ensure the success of the installation.

How to Install Docker on Windows

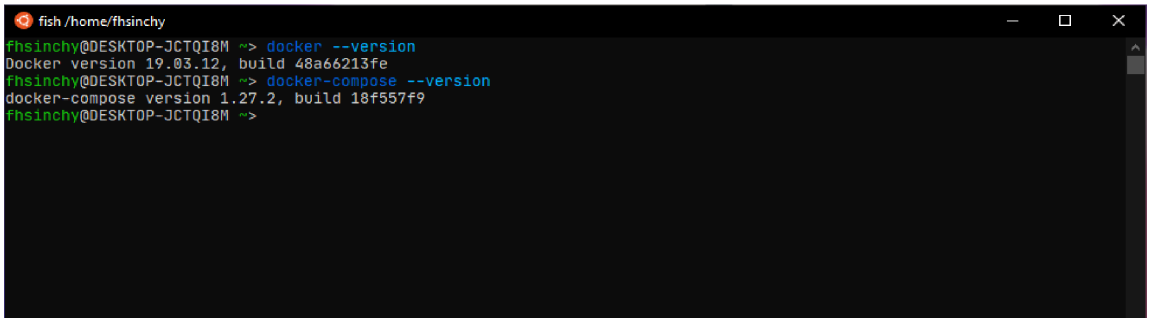
On Windows, the procedure is almost the same, except there are a few extra steps that you'll need to go
through. The installation steps are as follows:

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.

Once the installation is done, start *Docker Desktop* either from the 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 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.



How to Install Docker on Linux

Installing Docker on Linux is a bit of a different process, and 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.

Once the installation is done, open up the terminal and execute to docker --version ensure the 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.

Hello World in Docker – Intro to Docker Basics

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:
sha256:4cf9c47f86df71d48364001ede3a4fcd85ae80ce02ebad74156906caff
5378bc
# Status: Downloaded newer image for hello-world:latest #
# Hello from Docker!
# This message shows that your installation appears to be working
correctly.
#
# 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
Docker Hub.
#     (amd64)
#  3. The Docker daemon created a new container from that image
which runs the #     executable that produces the output you are
currently reading.
#  4. The Docker daemon streamed that output to the Docker
client, which sent it #     to your terminal.
#
# To try something more ambitious, you can run an Ubuntu
container with:
#  $ docker run -it ubuntu bash
#
# Share images, automate workflows, and more with a free Docker
ID: #  https://hub.docker.com/
#
# For more examples and ideas, visit:
```

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.

Now in your terminal, you can use the `docker ps -a` command to have a look at all the containers that are currently running or have run in the past:

```
docker ps -a
```

```
# CONTAINER ID   IMAGE         COMMAND    CREATED          STATUS
PORTS     NAMES
# 128ec8ceab71   hello-world   "/hello"   14 seconds ago   Exited (0) 13
seconds ago       exciting_chebyshev
```

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 -

A container is an abstraction at the application layer that packages code and dependencies together. Instead of virtualizing the entire physical machine, containers virtualize the host operating system only.

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

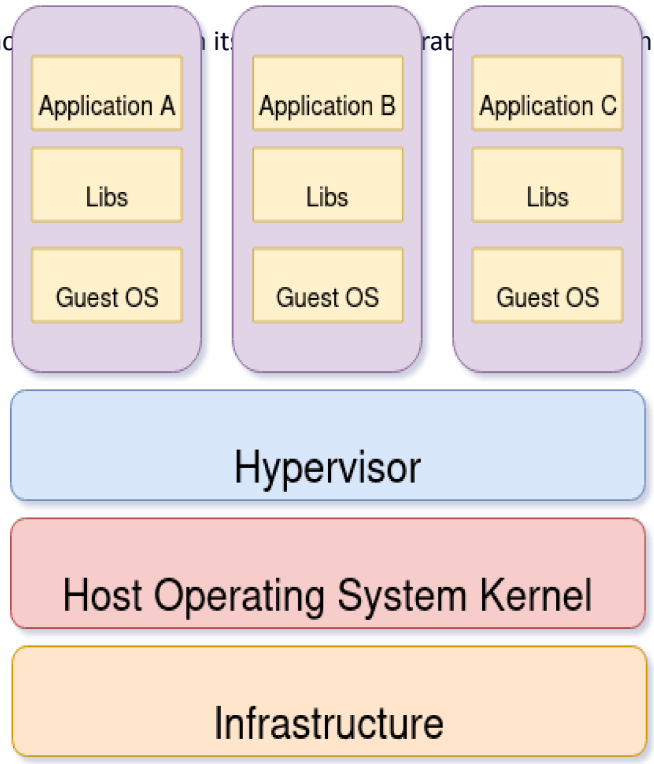
Just like virtual machines, containers are completely isolated environments from the host system as well as from each other. They are also a lot lighter than the traditional virtual machine, so a large number of containers can be run simultaneously without affecting the performance of the host system.

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 mac it rat n is just as heavy

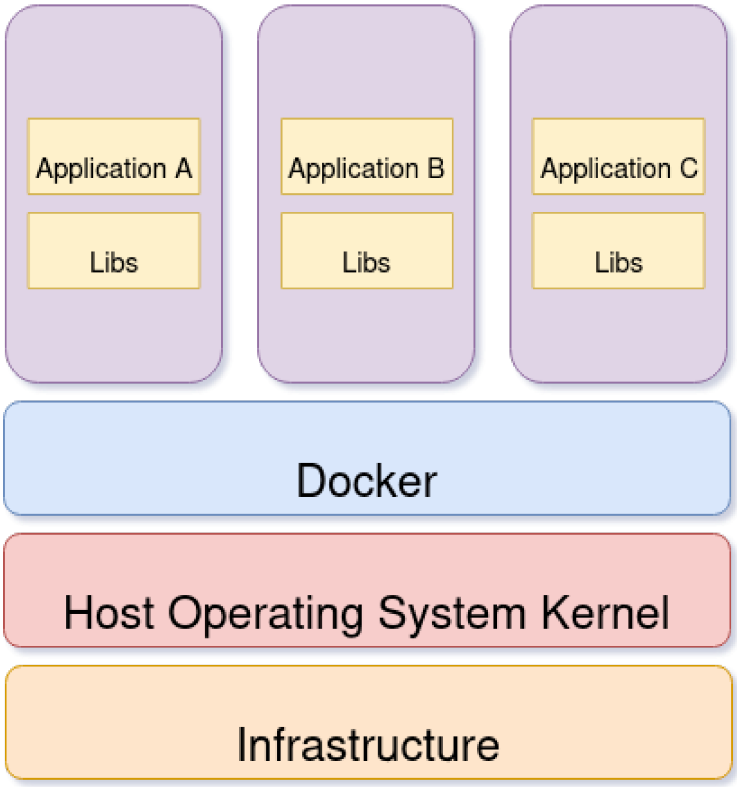


as the host operating system.

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 container runtime while maintaining isolation – just like a traditional virtual machine.



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 host 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:

```
uname -a
# Linux alpha-centauri 5.8.0-22-generic #23-Ubuntu SMP Fri Oct 9 00:34:40 UTC 2020
x86_64 x86_64 x86_64 GNU/Linux

docker run alpine uname -a
# Linux f08dbbe9199b 5.8.0-22-generic #23-Ubuntu SMP Fri Oct 9 00:34:40 UTC 2020
x86_64 Linux
```

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 essentially create another temporary writable layer on top of the previous read-only ones.

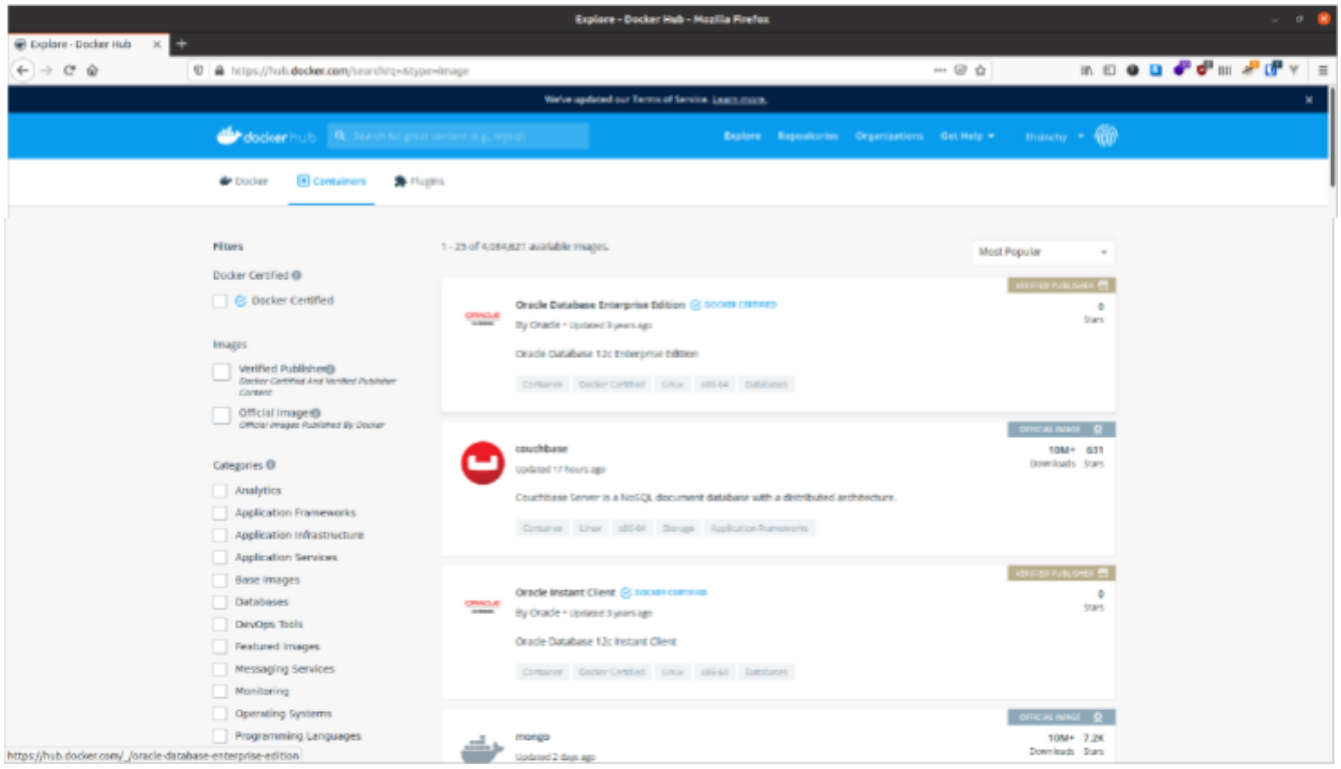
This concept will become a lot clearer in upcoming sections of this book. But for now, just keep in mind that images are multi-layered read-only files carrying your application in a desired state inside them.

What is a Docker Registry?

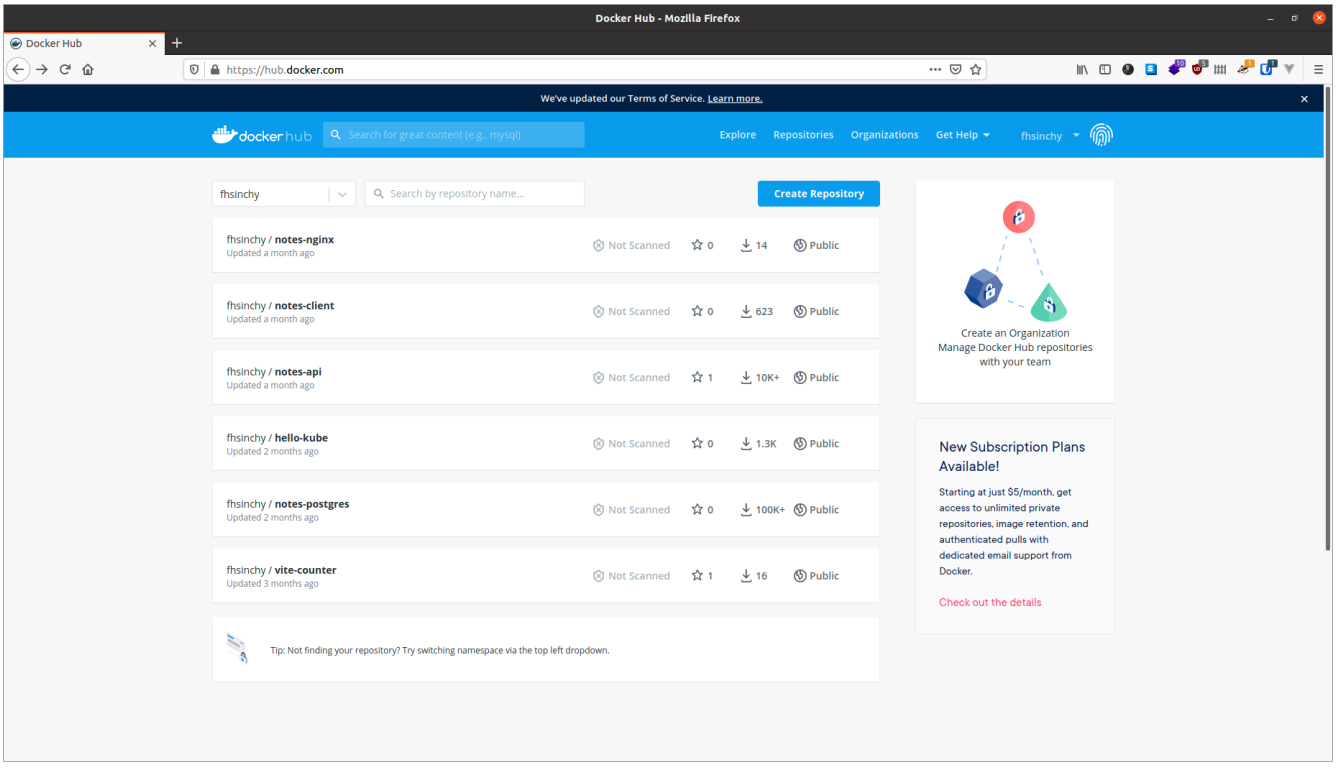
You've already learned about two very important pieces of the puzzle, *Containers* and *Images*. The final piece is the *Registry*.

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.



You can share any number of public images on Docker Hub for free. People around the world will be able to download them and use them freely. Images that I've uploaded are available on my profile (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 from the client. The daemon is capable of managing various Docker objects.
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.

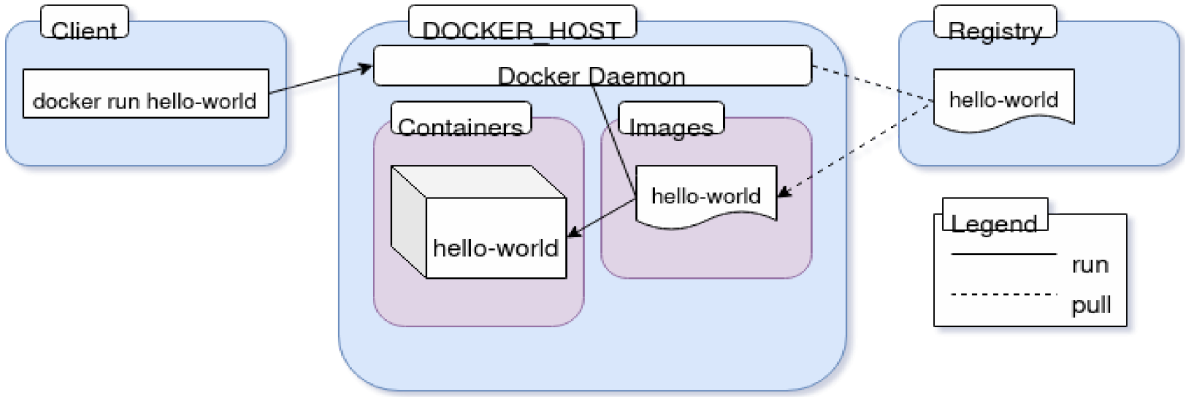
According to the official docs,

"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 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 `hello-world` image, indicated by the `latest: Pulling from library/hello-world` line in your terminal.

```
Unable to find image
'hello-world:latest' locally latest:
Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:d58e752213a51785838f9eed2b7a498ffa1cb3aa7f946dda11af39286c3db9a9
Status: Downloaded newer image for hello-world:latest
```

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 fetched, it'll stay in the local cache. So if you execute the command again, you won't see the following lines in the output:

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.

How to Run a Container

Previously you've used `docker run` to create and start a container 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.

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

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

-

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, will attempt to perform configuration
# /docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
# /docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
# 10-listen-on-ipv6-by-default.sh: Getting the checksum of /etc/nginx/conf.d/default.conf
# 10-listen-on-ipv6-by-default.sh: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
# /docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
# /docker-entrypoint.sh: Configuration complete; ready for start up
```

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.

To allow access from outside of a container, you must publish the appropriate port inside the container to a

port on your local network. The common syntax for the option is as 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.

How to Use Detached Mode

Another very popular option of the run command is the –detach 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-dock

# 9f21cb77705810797c4b847dbd330d9c732ffddba14fb435470567a7a3f46cdc
```

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.

How to List Containers

The container ls command can be used to list out containers that are currently running. To do so execute following command:

```
docker container ls

# CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
# 9f21cb777058        fhsinchy/hello-dock   "/docker-entrypoint.…"   5 seconds ago       Up
5 seconds       0.0.0.0:8080->80/tcp   gifted_sammet
```

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.

```
docker container ls --all

# CONTAINER ID   IMAGE          COMMAND             CREATED          STATUS                      PORTS                    NAMES
# 9f21cb777058   fhsinchy/hello-dock  "/docker-entrypoint...."  2 minutes ago    Up 2 minutes                0.0.0.0:8080->80/tcp  gifted_samme
# 6cf52771dde1   fhsinchy/hello-dock  "/docker-entrypoint...."  3 minutes ago    Exited (0) 3 minutes ago                  reverent_torvalds
# 128ec8ceab71   hello-world    "/hello"            4 minutes ago    Exited (0) 4 minutes ago                  exciting_chebyshev
```

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-dock-container fhsinchy/hello-dock
```

```
# b1db06e400c4c5e81a93a64d30acc1bf821bed63af36cab5cdb95d25e114f5fb
```

The 8080 port on local network is occupied by the `gifted_sammet` container (the container created in the previous sub-section). That's why you'll have to use a different port number, like 8888. Now to verify, run the command:

```
docker container ls



# CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES

# b1db06e400c4        fhsinchy/hello-dock   "/docker-entrypoint.…"   28 seconds ago      Up
26 seconds       0.0.0.0:8888->80/tcp   hello-dock-container

# 9f21cb777058        fhsinchy/hello-dock   "/docker-entrypoint.…"   4 minutes ago       Up
4 minutes        0.0.0.0:8080->80/tcp   gifted_sammet
```

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 contain command works for containers both in running state and stopped state.

How to Stop or Kill a Running Container

ctrl + c

Containers running in the foreground can be stopped by simply closing the terminal window or hitting **ctrl+c**. Containers running in the background, however, can not be stopped in the same 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.

In cases where you want to send a SIGKILL signal instead of a SIGTERM signal, you may use the container kill command 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:

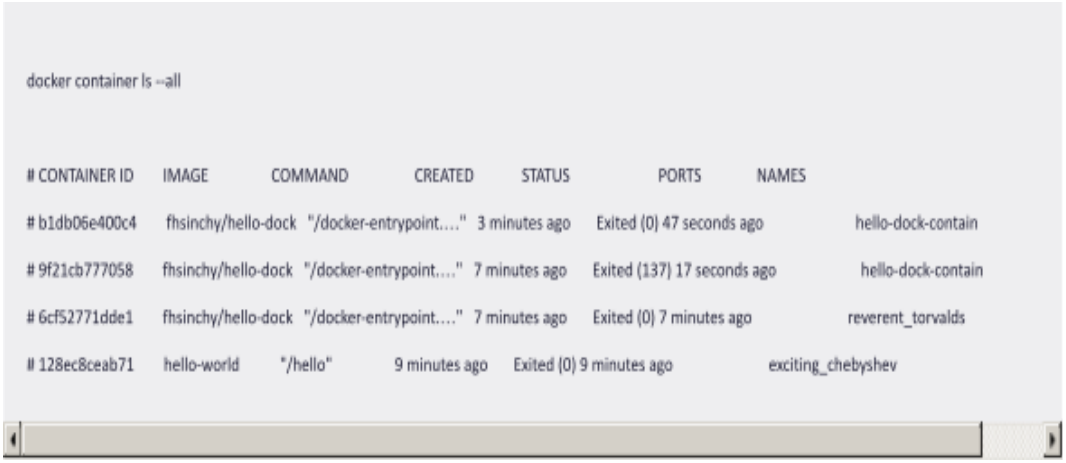
- Restarting a container that has been previously stopped or killed.
- Rebooting a running container.

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.

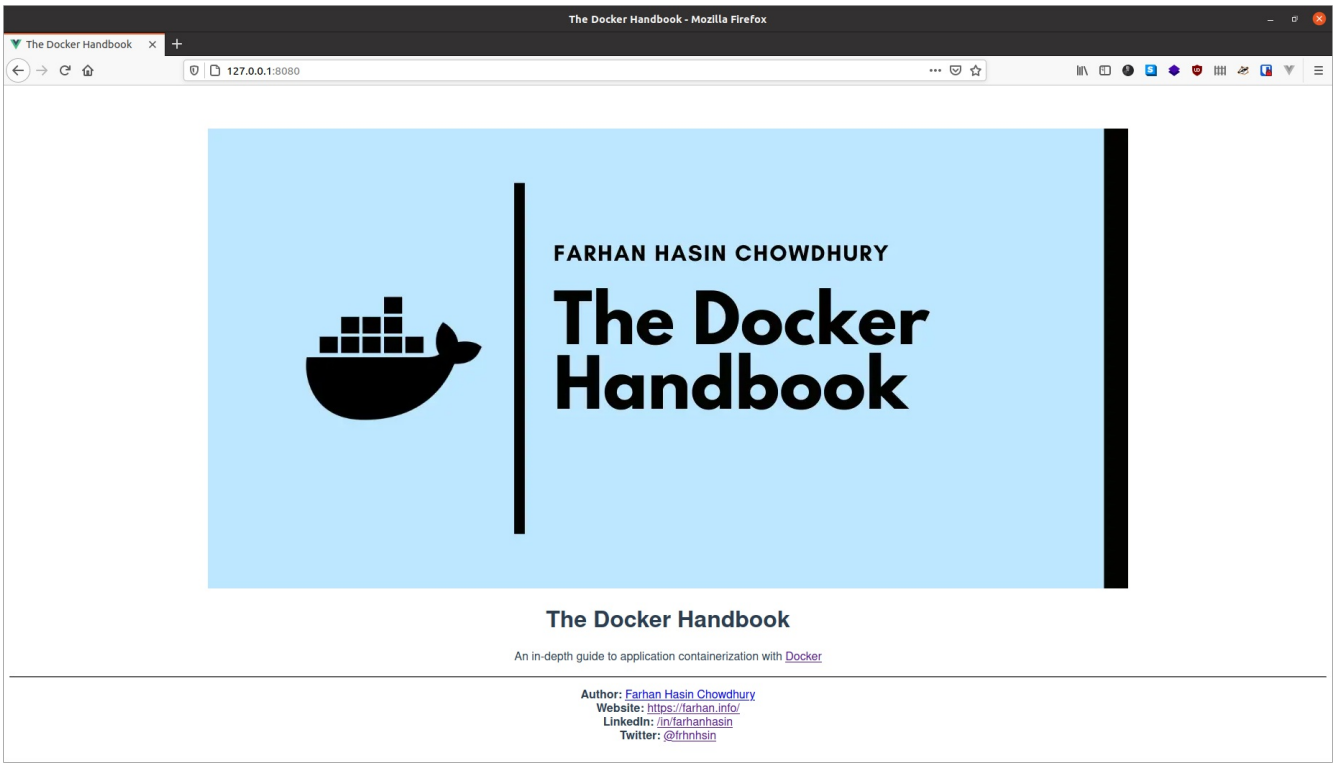
Now to restart the **hello-dock-container** container, you may execute the following command:

```
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 application just like before.



Now, in scenarios where you would like to reboot a running container you may use the container restart command. The 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
- **container start** a given image. 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 something like the following:

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

# 2e7ef5098bab92f4536eb9a372d9b99ed852a9a816c341127399f51a6d053856

docker container ls --all

# CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS               NAMES
# 2e7ef5098bab        fhsinchy/hello-dock   "/docker-entrypoint.…"   30 seconds ago
Created                             hello-dock
```

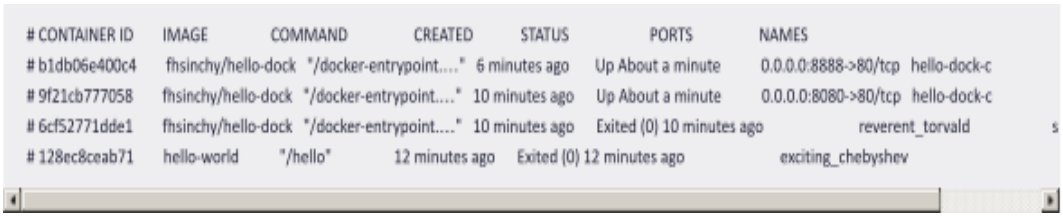
Evident by the output of the **container ls --all** command, a container with the name of **hello-dock** has been created using the **fhsinchy/hello-dock** image. The STATUS of the container is Created at the moment, and, given that it's not running, it won't be listed without the use of the --all option.

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

Although you can get away with the container run command for the majority of the scenarios, there will be some situations later on in the book that require you to use this container create command.

```
docker container start hello-dock
# hello-dock
docker container ls

# CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
PORTS               NAMES
# 2e7ef5098bab        fhsinchy/hello-dock   "/docker-entrypoint.…"   About a minute ago
Up 29 seconds       0.0.0.0:8080->80/tcp   hello-dock
```



```
# CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS                      PORTS                    NAMES
# b1db06e400c4   fhsinchy/hello-dock   "/docker-entrypoint...."   6 minutes ago    Up About a minute           0.0.0.0:8888->80/tcp   hello-dock-c
# 9f21cb777058   fhsinchy/hello-dock   "/docker-entrypoint...."   10 minutes ago   Up About a minute           0.0.0.0:8080->80/tcp   hello-dock-c
# 6cf52771dde1   fhsinchy/hello-dock   "/docker-entrypoint...."   10 minutes ago   Exited (0) 10 minutes ago                  reverent_torvald
# 128ec8ceab71   hello-world    "/hello"                 12 minutes ago   Exited (0) 12 minutes ago                  exciting_chebyshev
```

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
```

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:

```
docker container rm 6cf52771dde1

# 6cf52771dde1
```

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                  CREATED
# b1db06e400c4        fhsinchy/hello-dock   "/docker-entrypoint.…"   8 minutes ago
# 9f21cb777058        fhsinchy/hello-dock   "/docker-entrypoint.…"   12 minutes ago
```

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 hello-dock container with the --rm option, execute the following command:

```
docker container run --rm --detach --publish 8888:80 --name hello-dock-volatile
fhsinchy/hello-dock

# 0d74e14091dc6262732bee226d95702c21894678efb4043663f7911c53fb79f3
```

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

```
docker container ls

# CONTAINER ID   IMAGE              COMMAND              CREATED          STATUS
PORTS              NAMES
```

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     NAMES
```

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.

How to Run a Container in Interactive Mode

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.

```
docker container run --rm -it ubuntu

# root@dbb1f56b9563:/# cat /etc/os-release
# 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-policies/privacy-policy
"
# 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:

```
docker container run -it node

# Welcome to Node.js v15.0.0.
# Type ".help" for more information.
# > ['farhan', 'hasin', 'chowdhury'].map(name => name.toUpperCase())
# [ '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 00:34:40 UTC 2020
x86_64 Linux
```

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 almostvany 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.

The generic syntax for encoding a string using base64 is as follows:

```
# 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 echo -n my-secret | base64

# 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.

How to Work With Executable Images

In the previous section, I briefly mentioned executable images. 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**

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=rmbyext
```

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>
```

To test it out, open up your terminal inside an empty directory and create some files in it with different extensions. You can use the touch command to do so. Now, I have a directory on my computer 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
```

`pdf`
To delete all the 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

`rmbyext`
them just like the 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` directory inside the container, the files should be accessible to the container.

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 file system directory absolute path>:<read write access>
```

The third field is optional but you must pass the absolute path of your local directory and the absolute path of the directory inside 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.

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 a 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 creating your very own images.

In this section, you'll learn the fundamentals of creating images, 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

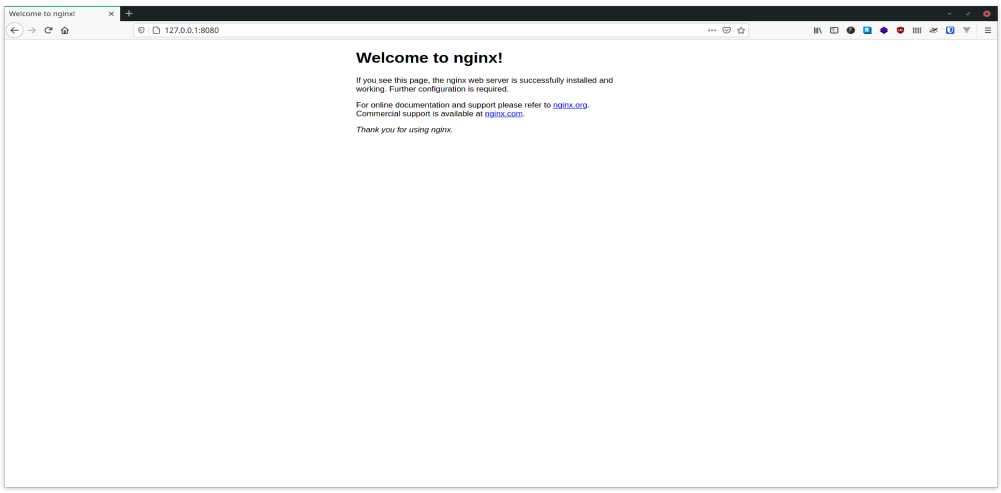
As I've already explained in the Hello World in Docker section, images are multi-layered self-contained files that act as the template for creating Docker containers. They are like a frozen, read-only copy of a container.

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 --publish 8080:80 nginx
#b379ecd5b6b9ae27c144e4fa12bdc5d06355436
66f75c14039eea8d5f38e3f56
docker container ls

# CONTAINER ID       IMAGE           COMMAND                  CREATED          STATUS          PORTS
NAMES
# b379ecd5b6b9       nginx           "/docker-entrypoint.…"   8 seconds ago    Up 8 seconds
0.0.0.0:8080->80/tcp   default-nginx
```

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` is a collection of instructions that,

once processed by the daemon, results in an image. Content for the `Dockerfile` is as follows:

```
FROM ubuntu:latest

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 written in shell form. These can also be written in exec form. You can consult the official reference for more information.

- 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 *Dockerfile* you just wrote, open up your terminal inside the custom-nginx the directory
and execute the following command:

```
docker image build .

# Sending build context to Docker daemon  3.584kB
# Step 1/4 : FROM ubuntu:latest
#  ---> d70eaf7277ea
# Step 2/4 : EXPOSE 80
#  ---> Running in 9eae86582ec7
# Removing intermediate container 9eae86582ec7
#  ---> 8235bd799a56
# Step 3/4 : RUN apt-get update &&     apt-get install nginx -y &&     apt-get clean && rm
-rf /var/lib/apt/lists/* #  ---> Running in a44725cbb3fa
### LONG INSTALLATION STUFF GOES HERE ###
# Removing intermediate container a44725cbb3fa
#  ---> 3066bd20292d
# 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.

To verify, visit http://127.0.0.1:8080 and you should see the default response page.

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 execute docker container run mysql:5.7 where mysql is the image repository and `5.7` is the tag.

In order to tag your custom NGINX image with 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 &&     apt-get clean && rm
-rf /var/lib/apt/lists/* #  ---> 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 image tag use the command to do that:

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

## or ##

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

How to List and Remove Docker Images

Just like the **container ls** command, you can use the image ls 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:

```
docker image rm custom-nginx:packaged

# Untagged: custom-nginx:packaged
# Deleted:
sha256:f8837621b99d3388a9e78d9ce49fbb773017f770eea80470fb85e0052beae242
# Deleted:
sha256:fdc6cdd8925ac25b9e0ed1c8539f96ad89ba1b21793d061e2349b62dd517dadf
# Deleted:
sha256:c20e4aa46615fe512a4133089a5cd66f9b7da76366c96548790d5bf865bd49c4
# Deleted:
sha256:6d6460a744475a357a2b631a4098aa1862d04510f3625feb316358536fcd8641
```

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:ba9558bdf2beda81b9acc652ce4931a85f0fc7f69dbc91b4efc4561ef7378aff
# deleted:
sha256:ad9cc3ff27f0d192f8fa5fadebf813537e02e6ad472f6536847c4de183c02c81
# deleted:
sha256:f1e9b82068d43c1bb04ff3e4f0085b9f8903a12b27196df7f1145aa9296c85e7
# deleted:
sha256:ec16024aa036172544908ec4e5f842627d04ef99ee9b8d9aaa26b9c2a4b52baa

# Total reclaimed space: 59.19MB
```

The 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-nginx:packaged` image can be visualized as follows:

```
docker image history custom-nginx:packaged

# IMAGE          CREATED          CREATED BY                                      SIZE
COMMENT
# 7f16387f7307       5 minutes ago       /bin/sh -c #(nop)  CMD ["nginx" "-g" "daemon…
0B
# 587c805fe8df       5 minutes ago       /bin/sh -c apt-get update &&     apt-get ins…
60MB
# 6fe4e51e35c1       6 minutes ago       /bin/sh -c #(nop)  EXPOSE 80                 0B
# d70eaf7277ea       17 hours ago        /bin/sh -c #(nop)  CMD ["/bin/bash"]         0B
# <missing>          17 hours ago        /bin/sh -c mkdir -p /run/systemd && echo 'do…   7B
# <missing>          17 hours ago        /bin/sh -c [ -z "$(apt-get indextargets)" ]     0B
# <missing>          17 hours ago        /bin/sh -c set -xe   && echo '#!/bin/sh' > /…   811B
# <missing>          17 hours ago        /bin/sh -c #(nop) ADD file:435d9776fdd3a1834…
72.9MB
```

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.
- Finally the upper most layer 7f16387f7307 was created by /bin/sh -c #(nop) CMD ["nginx", "-g", "daemon off;"] 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 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 `rmbyext` for the application resides inside the sub-directory with the same name.
Before you start working on the 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.
- The `rmbyext` script should be set as the entry-point so the image can take extension names as arguments.

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=rmbyext &&     apk del git

ENTRYPOINT [ "rmbyext" ]
```

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

In this FROM 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:

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

```
docker image build --tag rmbyext .

# Sending build context to Docker daemon  2.048kB
# Step 1/4 : FROM python:3-alpine
# 3-alpine: Pulling from library/python
# 801bfaa63ef2: Already exists
# 8723b2b92bec: Already exists
# 4e07029ccd64: Already exists
# 594990504179: Already exists
# 140d7fec7322: Already exists
# Digest: sha256:7492c1f615e3651629bd6c61777e9660caa3819cf3561a47d1d526dfeee02cf6
# Status: Downloaded newer image for python:3-alpine
#  ---> d4d4f50f871a
# Step 2/4 : WORKDIR /zone
#  ---> Running in 454374612a91
# Removing intermediate container 454374612a91
#  ---> 7f7e49bc98d2
# Step 3/4 : RUN apk add --no-cache git &&     pip install
git+https://github.com/fhsinchy/rmbyext.git#egg=rmbyext &&     apk del git #  ---> Running in
27e2e96dc95a
### LONG INSTALLATION STUFF GOES HERE ###
# Removing intermediate container 27e2e96dc95a
#  ---> 3c7389432e36
# Step 4/4 : ENTRYPOINT [ "rmbyext" ]
#  ---> Running in f239bbea1ca6
# Removing intermediate container f239bbea1ca6
#  ---> 1746b0cedbc7
#
Successf
ully built
1746b0c
edbc7 #
Successf
ully
tagged
rmbyext
:latest

docker image ls

# REPOSITORY        TAG       IMAGE ID      CREATED         SIZE
# rmbyext           latest    1746b0cedbc7  4 minutes ago   50.9MB
```

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:

```
docker login

# Login with your Docker ID to push and pull images from Docker Hub. If you don't have a
Docker ID, head over to https://hub.docker.com to creat # Username: fhsinchy #
Password:
```

```
# WARNING! Your password will be stored unencrypted in
/home/fhsinchy/.docker/config.json.
# Configure a credential helper to remove this warning. See
# https://docs.docker.com/engine/reference/commandline/login/#credentials-store
```

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 aprevious 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>

fhsinchy

```
$ docker image build --tag fhsinchy/custom-nginx:latest --file Dockerfile.built .

# Step 1/9 : FROM ubuntu:latest
#  ---> d70eaf7277ea
# Step 2/9 : RUN apt-get update && apt-get install build-essential libpcre3 libpcre3-dev zlib1g
#  ---> cbe1ced3da11
### LONG INSTALLATION STUFF GOES HERE ###
# Step 3/9 : ARG FILENAME="nginx-1.19.2"
#  ---> Running in 33b62a0e9ffb
# Removing intermediate container 33b62a0e9ffb
#  ---> fafc0aceb9c8
# Step 4/9 : ARG EXTENSION="tar.gz"
#  ---> Running in 5c32eeb1bb11
# Removing intermediate container 5c32eeb1bb11
#  ---> 36efdf6efacc
```

```
# Step 5/9 : ADD https://nginx.org/download/${FILENAME}.${EXTENSION} .
#Downloading[==================================================
```

```
>
]
1
.
0
4
9
M
B
/
1
.
0
4
9
M
B
#
-
-
-
>
d
b
a
2
5
2
f
8
d
6
0
9
#
S
t
e
p
6
/
9
:
R
U
N
t
a
r
-
x
v
f
$
{
F
I
L
E
N
A
M
E
}
.
$
{
E
X
T
```

```
E
N
S
I
O
N
}
&
&
r
m
$
{
F
I
L
E
N
A
M
E
}
.
$
{
E
X
T
E
N
S
I
O
N
}
#
-
-
-
>
R
u
n
n
i
n
g
i
n
2
f
5
b
0
9
1
b
2
1
2
5
### LONG
EXTRACTIO
N STUFF
GOES HERE
###
# Removing
intermediat
e container
```

```
2f5b091b2125
 ---> 2c9a325d74f1
Step 7/9 : RUN cd ${FILENAME} && ./configure --sbin-path=/usr/bin/nginx --conf-path=/etc/nginx/nginx.conf --error-log-path=/
 ---> Running in 11cc82dd5186
### LONG CONFIGURATION AND BUILD STUFF GOES HERE ###
Removing intermediate container 11cc82dd5186
 ---> 6c122e485ec8
Step 8/9 : RUN rm -rf /${FILENAME}}
 ---> Running in 04102366960b
Removing intermediate container 04102366960b
 ---> 6bfa35420a73
Step 9/9 : CMD ["nginx", "-g", "daemon off;"]
 ---> Running in 63ee44b571bb
Removing intermediate container 63ee44b571bb
```

```
# Login Succeeded
```

```
# --->
4ce79556db
1b
#
Successfully
built
4ce79556db
1b
#
Successfully
tagged
fhsinchy/cus
tom-nginx:l
atest
```

syntax. My username is so the command will look like this:

v

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:ffe93440256c9edb2ed67bf3bba3c204fec3a46a36ac53358899ce1a9eee497a size:
1788
```

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.

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.
- Accessing the database server using its IP address and default 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.Networks}} {{.IPAddress}}
{{end}}' notes-api-db-server
#  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 destroyed and recreated, the IP address may change. Keeping track of these
changing IP addresses can be pretty hectic.

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 attached to the network of the host system.
- **none** - This driver disables networking for containers 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 container run --rm --detach --name hello-dock --publish 8080:80
fhsinchy/hello-dock
# a37f723dad3ae793ce40f97eb6bb236761baa92d72a2c27c24fc7fda0756657d

docker network inspect --format='{{range .Containers}}{{.Name}}{{end}}' bridge
# 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 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 name.
- **User-defined bridges provide better isolation:**

```
docker network create skynet
```

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.

Now that you've learned quite a lot about a user-defined network, it's time to create one for yourself. A network can be created using the `network create` command. The generic syntax for the command is as follows:

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

skyne
t

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

```
# 7bd5f351aa892ac6ec15fed8619fc3bbb95a7dcdd58980c28304627c8f7eb070

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 container to a network. The generic syntax for the command is as follows:

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

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

```
docker network connect skynet hello-dock

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

#  hello-dock

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

#  hello-dock
```

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

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

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

hello-dock
To run another container attached to the same network, you can execute the following command:

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

`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.

How to Detach Containers from a Network in Docker

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.

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

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

`hello-dock` container from the `skynet`

To detach the network, you can execute the following command:

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

`network connect` command, the `network disconnect`
Just like the 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>
```

`skynet`

To remove the network from your system, you can execute the following command:

```
docker network rm skynet
```

`network prune`

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