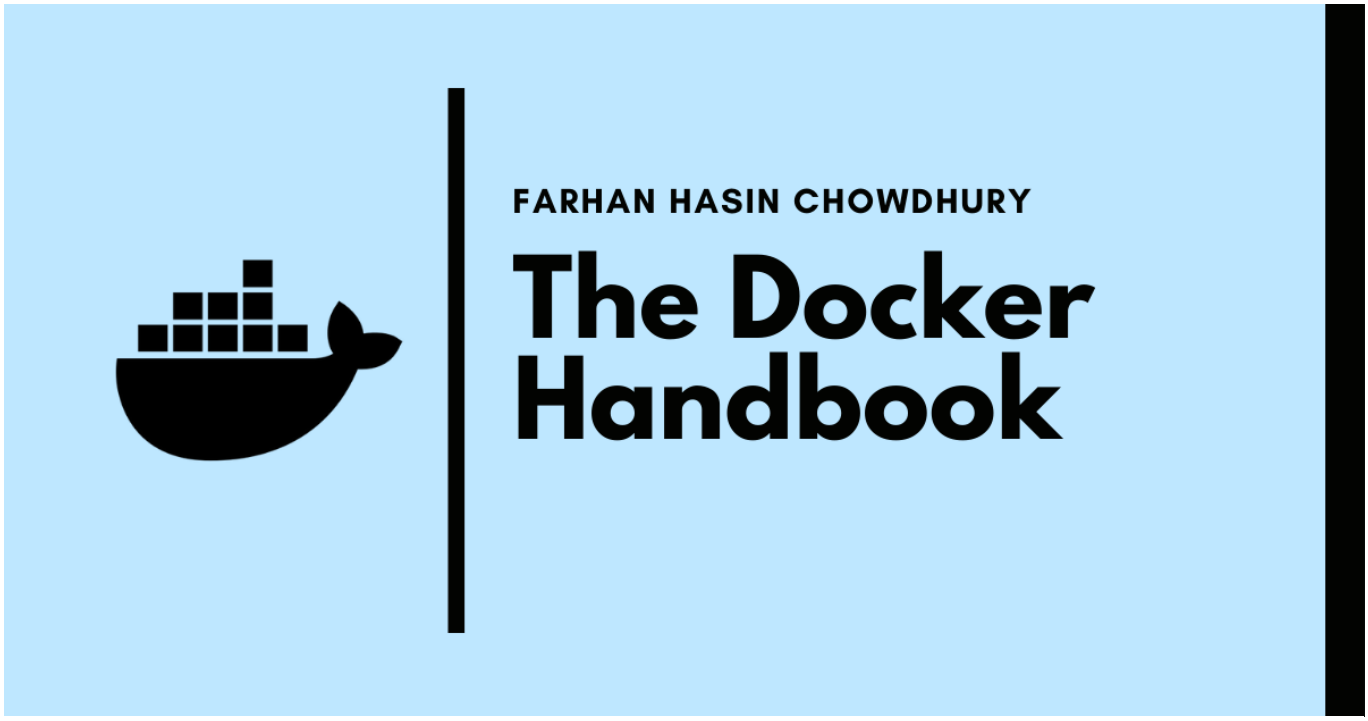
13 JULY 2020 / #DOCKER

The Docker Handbook

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

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

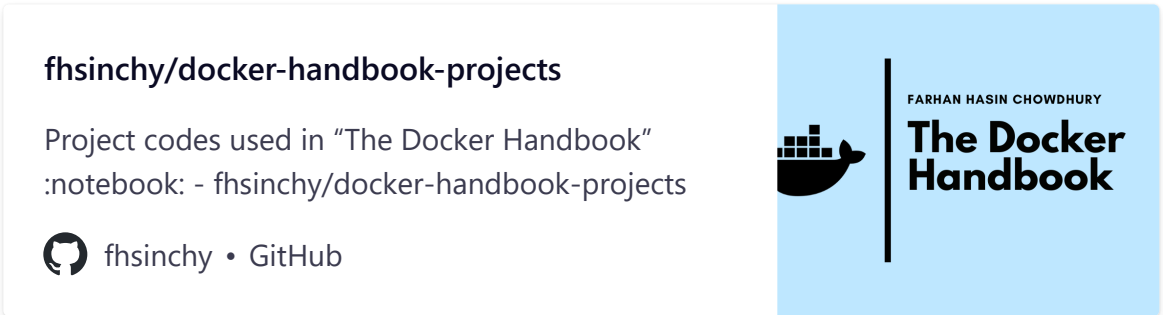
- Containerize (almost) any application
- Upload custom Docker Images in Docker Hub
- Work with multiple containers using Docker Compose

Prerequisites

- Familiarity with the Linux Terminal
- Familiarity with JavaScript (some of the later projects use JavaScript)

Project Code

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



Spare a ☆ to keep me motivated

Donate

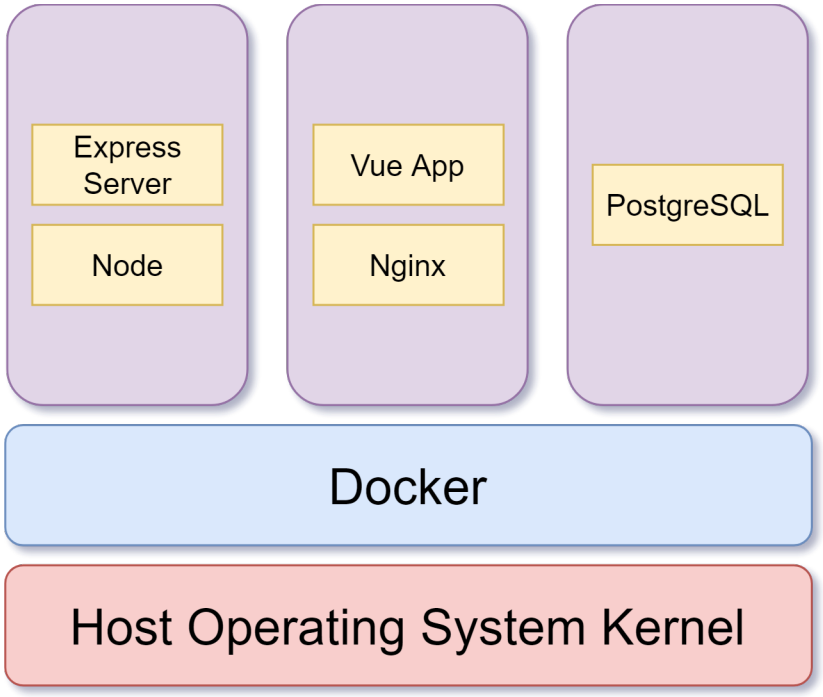
Table of Contents

Introduction to Containerization and Docker

Containerization is the process of encapsulating software code along with all of its dependencies inside a single package so that it can be

Docker is an open source containerization platform. It provides the ability to run applications in an isolated environment known as a *container*.

Containers are like very lightweight virtual machines that can run directly on our host operating system's kernel without the need of a hypervisor. As a result we can run multiple containers simultaneously.



Simultaneously Running Containers

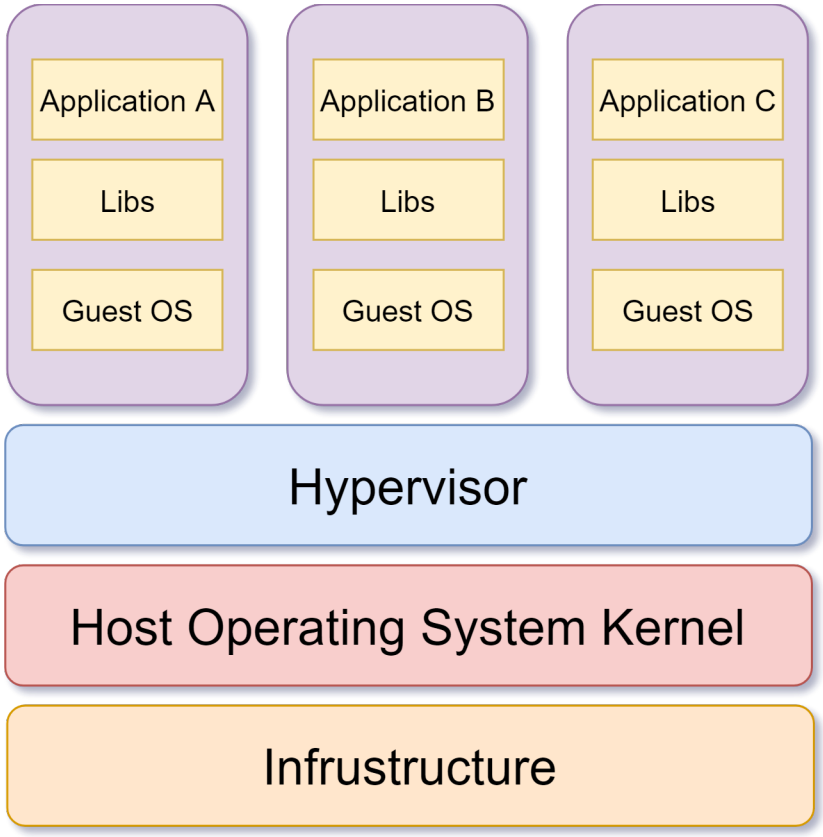
Each container contains an application along with all of its dependencies and is isolated from the other ones. Developers can exchange these containers as *image(s)* through a *registry* and can also deploy directly on servers.

Virtual Machines vs Containers

A virtual machine is the emulated equivalent of a physical computer

system.

A program known as a hypervisor creates and runs virtual machines. The physical computer running a hypervisor is called the host system, while the virtual machines are called guest systems.



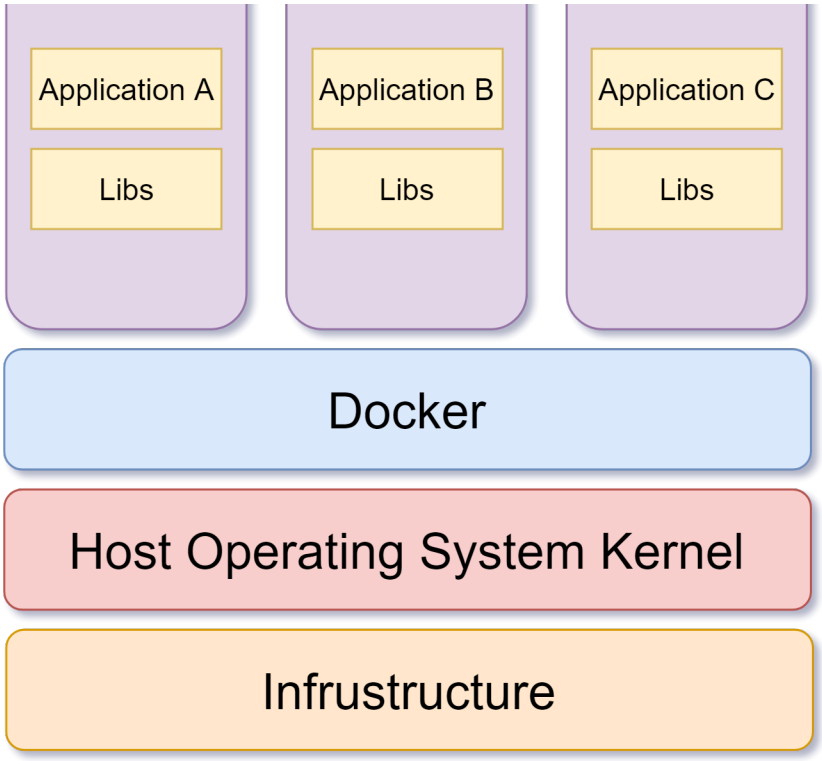
Virtual Machines

The hypervisor treats resources — like the CPU, memory, and storage — as a pool that can be easily reallocated between the existing guest virtual machines.

Hypervisors are of two types:

- Type 1 Hypervisor (VMware vSphere, KVM, Microsoft Hyper-V).
- Type 2 Hypervisor (Oracle VM VirtualBox, VMware Workstation Pro/VMware Fusion).

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.

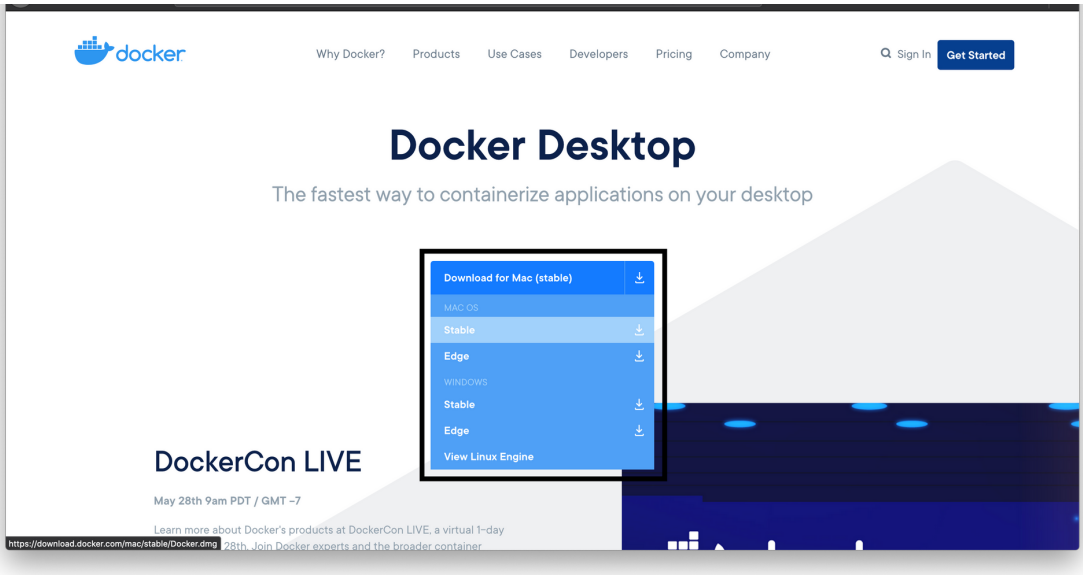


Containers

Containers sit on top of the physical machine and its operating system. Each container shares the host operating system kernel and, usually, the binaries and libraries, as well.

Installing Docker

Navigate to the download page for Docker Desktop and choose your operating system from the drop-down:

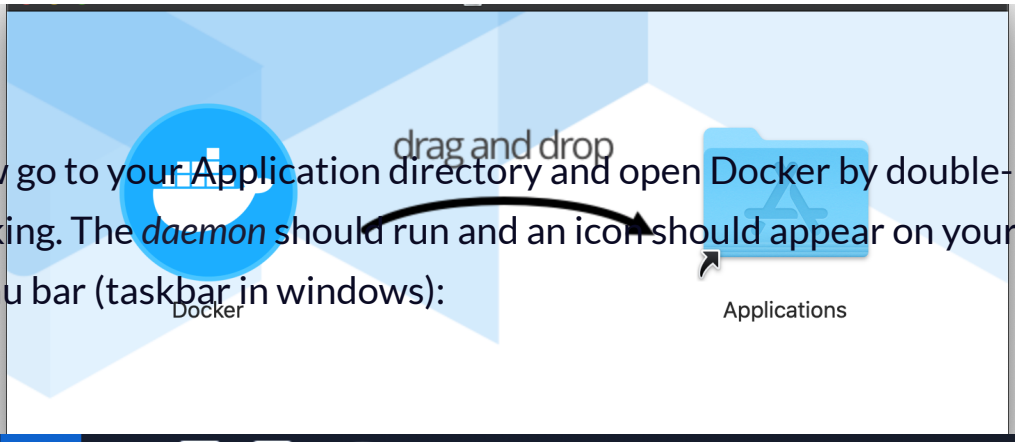


Select Your Operating System

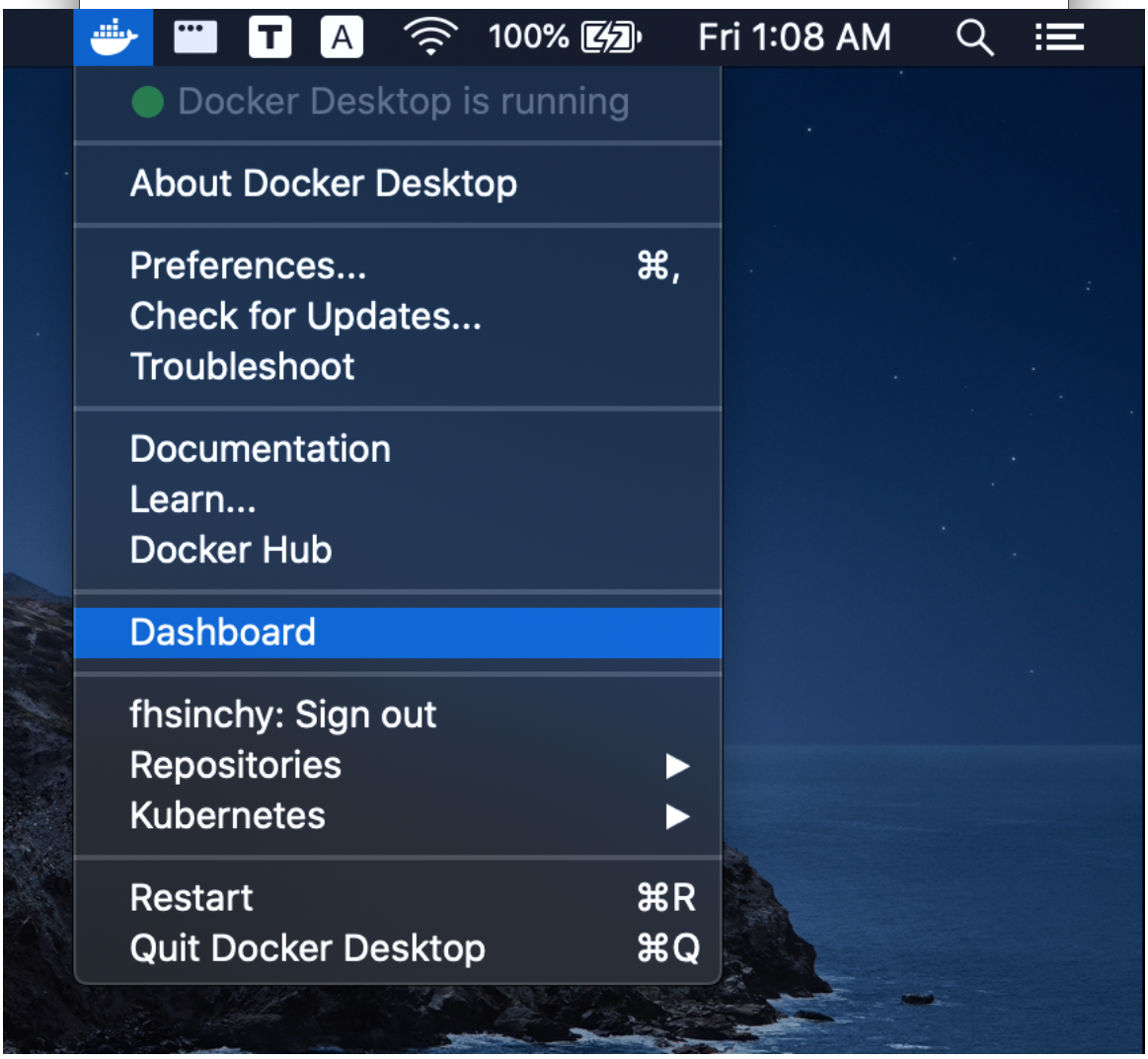
I'll be showing the installation process for the Mac version but I believe installation for other operating systems should be just as straightforward.

The Mac installation process has two steps:

1. Mounting the downloaded *Docker.dmg* file.
2. Dragging and dropping *Docker* into your *Application* directory.

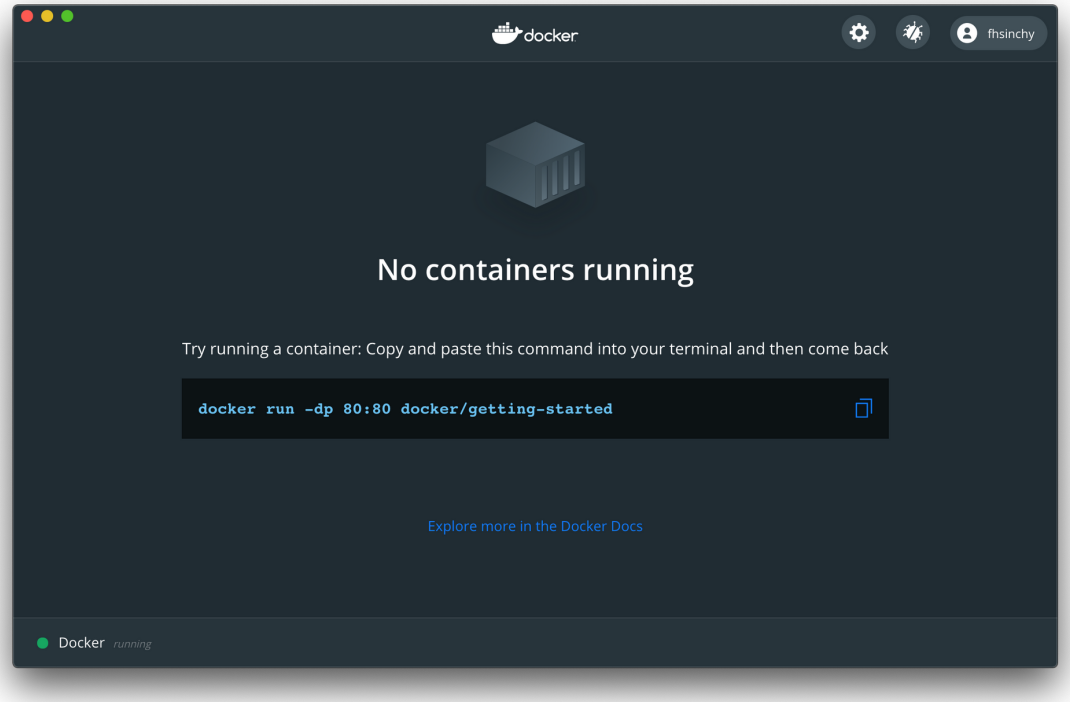


Now go to your Application directory and open Docker by double-clicking. The *daemon* should run and an icon should appear on your menu bar (taskbar in windows):



Docker Icon

You can use this icon to access the *Docker Dashboard*:



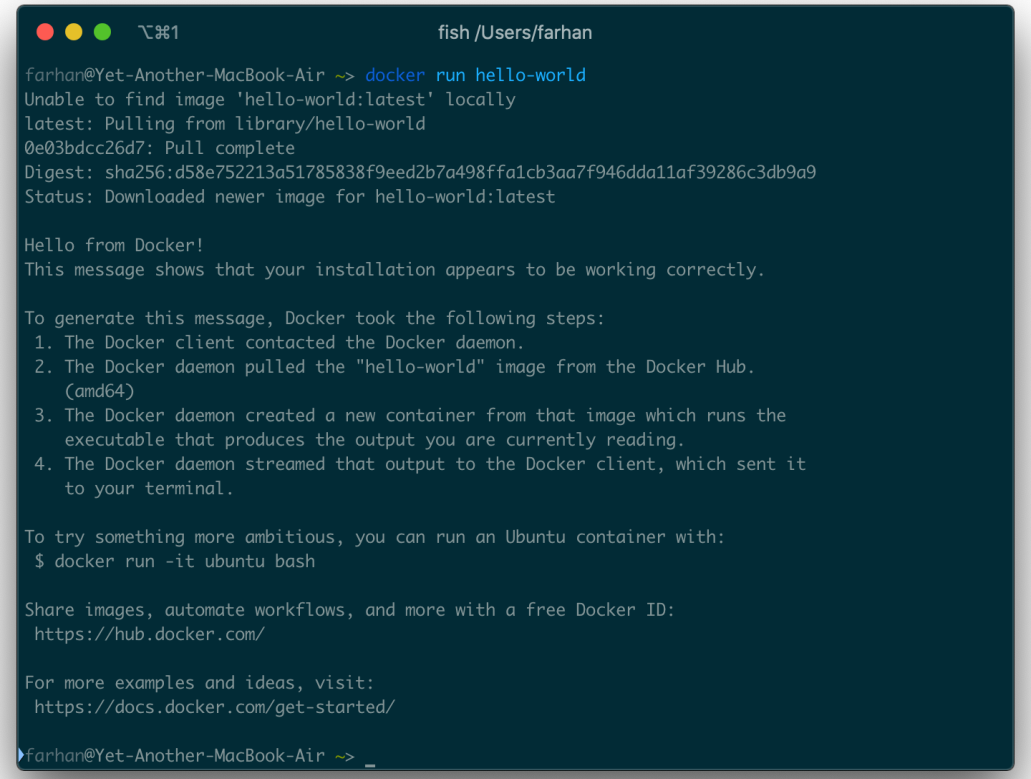
Docker Dashboard

It may look a bit boring at the moment, but once you've run a few containers, this will become much more interesting.

Now that we have Docker ready to go on our machines, it's time for us to run our first container. Open up terminal (command prompt in windows) and run following command:

```
docker run hello-world
```

If everything goes fine you should see some output like the following:

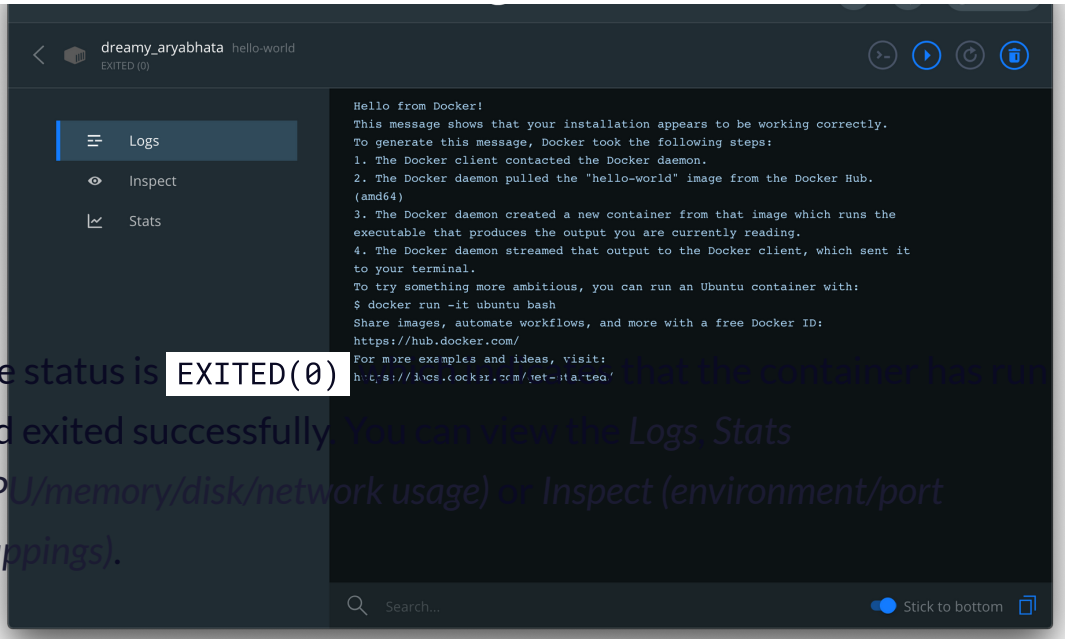


output from `docker run hello-world` command

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

Almost every image contains a default command. In case of the hello-world image, the default command is to execute the *hello* binary compiled from the previously mentioned C code.

If you open up the dashboard again, you should find the hello-world container there:



The status is `EXITED(0)` that the container has run and exited successfully. You can view the *Logs*, *Stats (CPU/memory/disk/network usage)* or *Inspect (environment/port mappings)*.

To understand what just happened, you need to get familiar with the Docker Architecture, Images and Containers, and Registries.

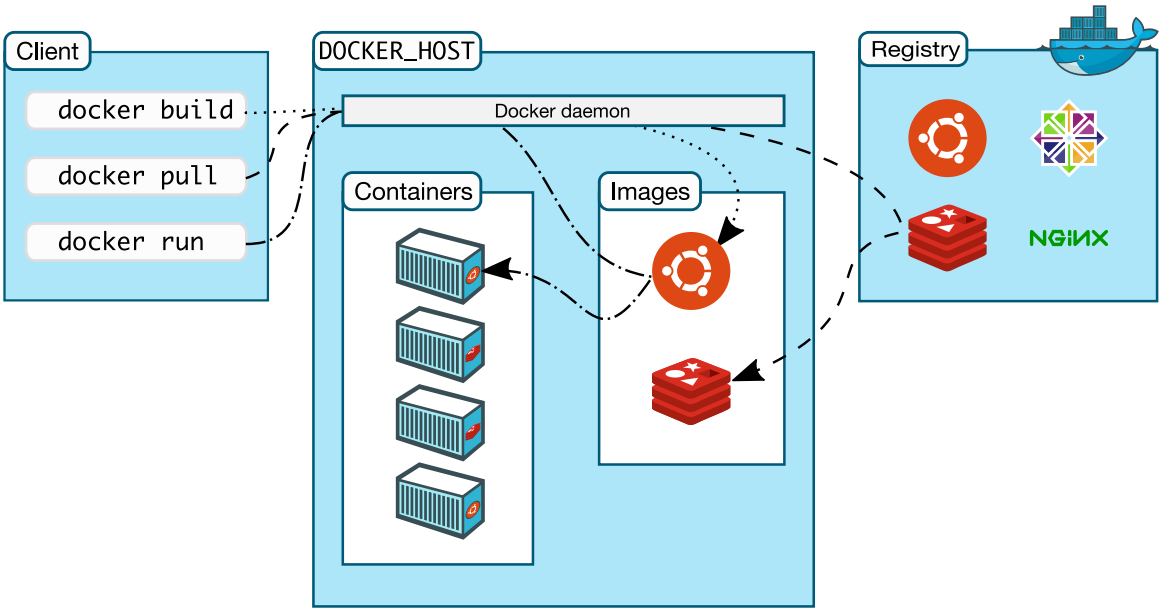
Container Logs

Docker Architecture

Docker uses a client-server architecture. The engine consists of three major components:

1. **Docker Daemon:** The daemon is a long running application that keeps on going in the background, listening to the commands issued by the client. It can manage Docker objects such as images, containers, networks, and volumes.
2. **Docker Client:** The client is a command-line interface program accessible by `docker` command. This client tells the server what to do. When we execute a command like `docker run hello-world`, the client tells the the daemon to carry out the task.
3. **REST API:** Communication between the daemon and the client happens using a REST API over UNIX sockets or network interfaces.

There is a nice graphical representation of the architecture on Docker's official documentation:



https://docs.docker.com/get-started/overview/#docker-architecture

Don't worry if it looks confusing at the moment. Everything will become much clearer in the upcoming sub-sections.

Images and Containers

Images are multi-layered self-contained files with necessary instructions to create containers. Images can be exchanged through registries. We can use any image built by others or can also modify them by adding new instructions.

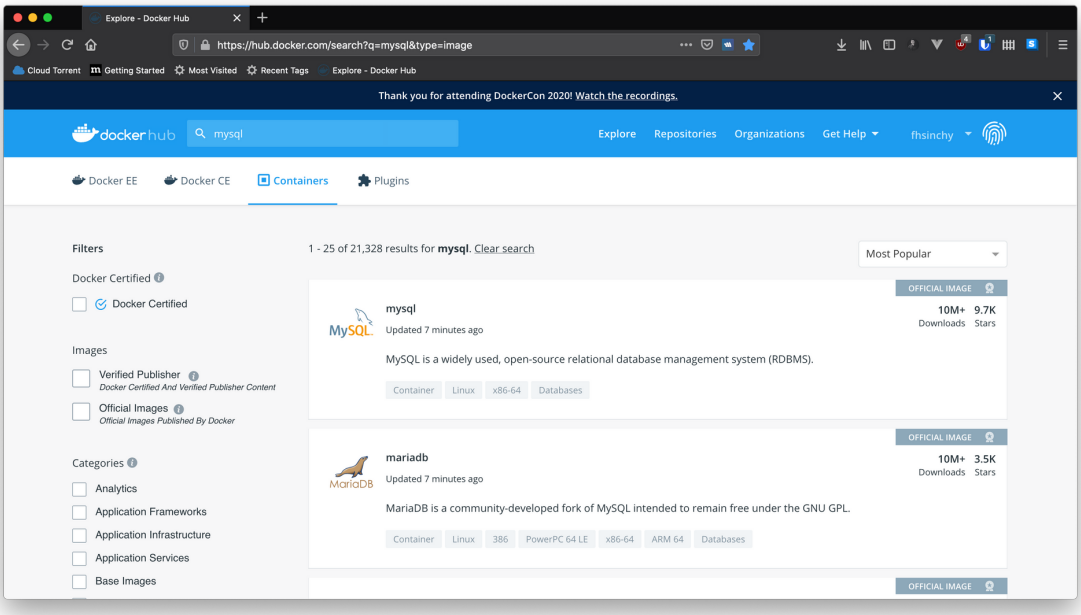
Images can be created from scratch as well. The base layer of an image is read-only. When we edit a Dockerfile and rebuild it, only the modified part is rebuilt in the top layer.

Containers are runnable instances of images. When we pull an image

suitable for running the program included in the image. This isolated environment is a container. If we compare images with classes from OOP then containers are the objects.

Registries

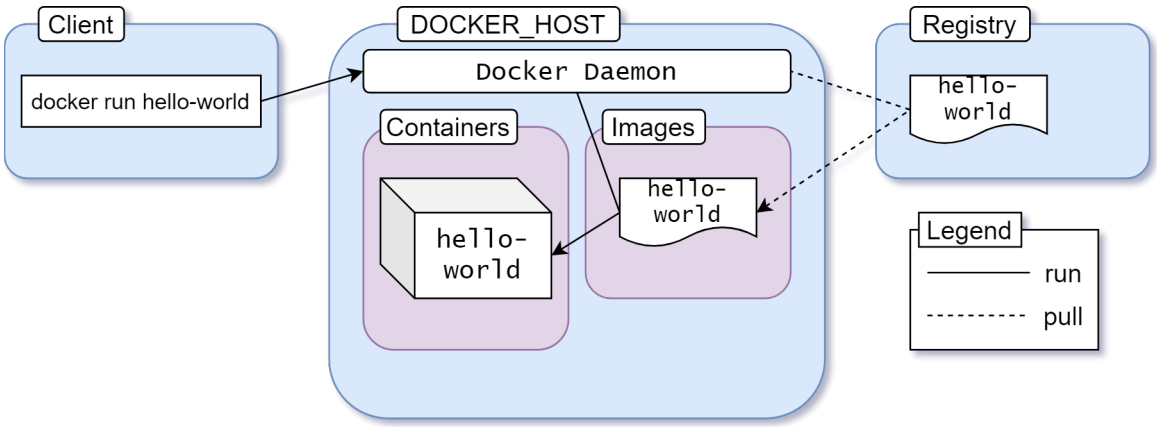
Registries are storage for Docker images. Docker Hub is the default public registry for storing images. Whenever we execute commands like `docker run` or `docker pull` the daemon usually fetches images from the hub. Anyone can upload images to the hub using `docker push` command. You can go to the hub and search for images like any other website.



Docker Hub

If you create an account, you'll be able to upload custom images as well. Images that I've uploaded are available for everyone at https://hub.docker.com/u/fhsinchy page.

Now that you're familiar with the architecture, images, containers, and registries, you're ready to understand what happened when we executed the `docker run hello-world` command. A graphical representation of the process is as follows:



docker run hello-world

The entire process happens in five steps:

1. We execute the `docker run hello-world` command.
2. Docker client tells the daemon that we want to run a container using the hello-world image.
3. Docker daemon pulls the latest version of the image from the registry.
4. Creates a container from the image.
5. Runs the newly created container.

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,

won't see the following lines in the output:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
0e03bdcc26d7: Pull complete
Digest: sha256:d58e752213a51785838f9eed2b7a498ffa1cb3aa7f946dda11af39286c
Status: Downloaded newer image for hello-world:latest
```

If there is a newer version of the image available, 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 more detail in a later section.

Manipulating Containers

In the previous section, we've had a brief encounter with the Docker client. It is the command-line interface program that takes our commands to the Docker daemon. In this section, you'll learn about more advanced ways of manipulating containers in Docker.

Running Containers

In the previous section, we've used `docker run` to create and run a container using the hello-world image. The generic syntax for this command is:

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

Here `image name` can be any image from Docker Hub or our local

run and not just **run**, the reason behind that is the `docker run` command actually does the job of two separate docker commands. They are:

1. `docker create <image name>` - creates a container from given image and returns the container id.
2. `docker start <container id>` - starts a container by given id of a already created command.

To create a container from the hello-world image execute the following command:

```
docker create hello-world
```

The command should output a long string like `cb2d384726da40545d5a203bdb25db1a8c6e6722e5ae03a573d717cd93342f61` – this is the container id. This id can be used to start the built container.

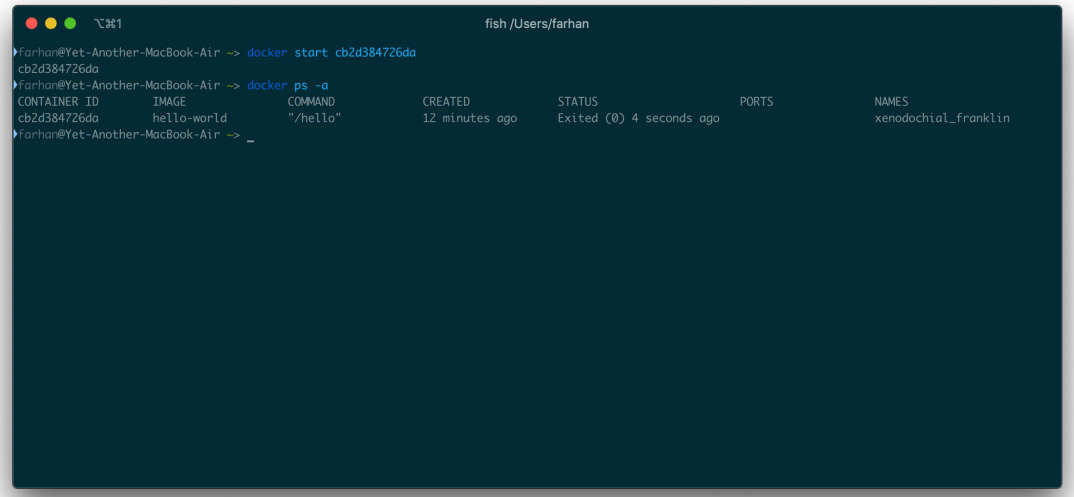
The first 12 characters of the container id are enough for identifying the container. Instead of using the whole string, using `cb2d384726da` should be fine.

To start this container execute the following command:

```
docker start cb2d384726da
```

You should get the container id back as output and nothing else. You

dashboard, you'll see that the container has run and exited successfully.



Output from `docker start cb2d384726da` and `docker ps -a` commands

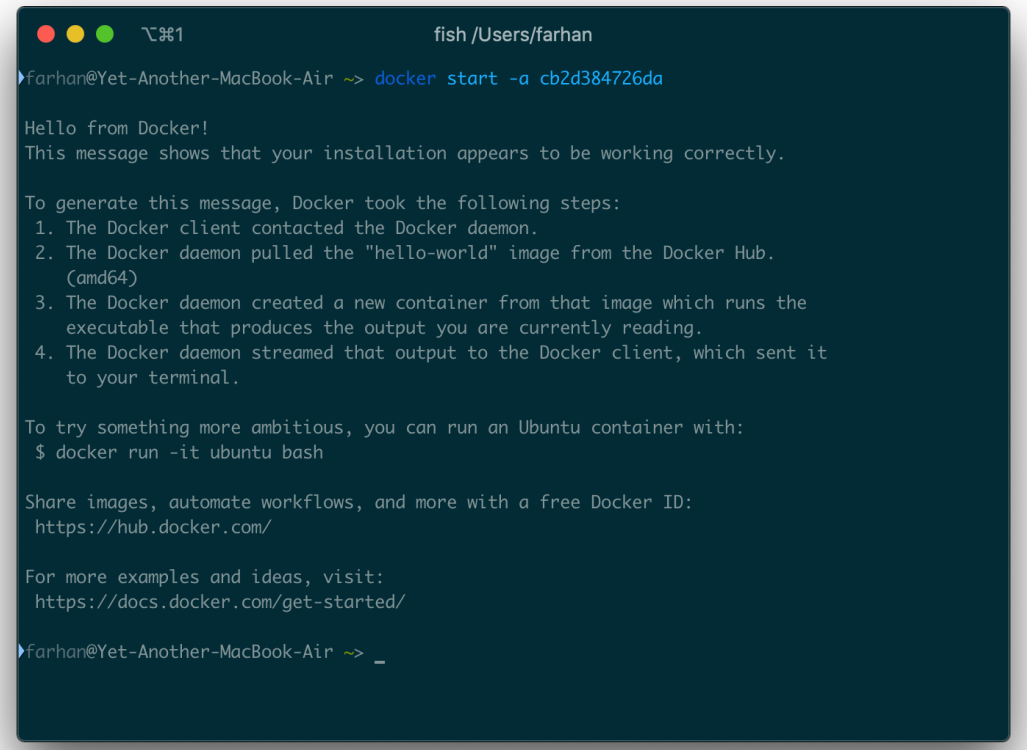
What happened here is we didn't attach our terminal to the output stream of the container. UNIX and LINUX commands usually open three I/O streams when run, namely `STDIN`, `STDOUT`, and `STDERR`.

If you want to learn more, there is an amazing article out there on the topic.

To attach your terminal to the output stream of the container you have to use the `-a` or `--attach` option:

```
docker start -a cb2d384726da
```

If everything goes right, then you should see the following output:

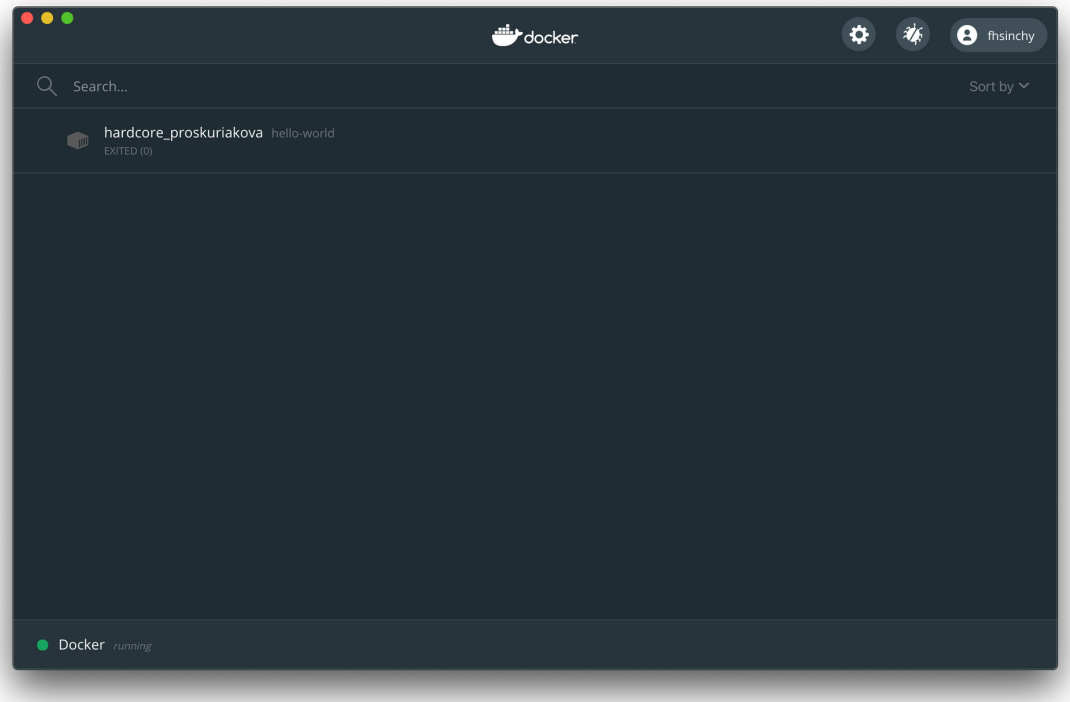


Output from `docker start -a cb2d384726da` command

already running. Using `run` command will create a new container every time.

Listing Containers

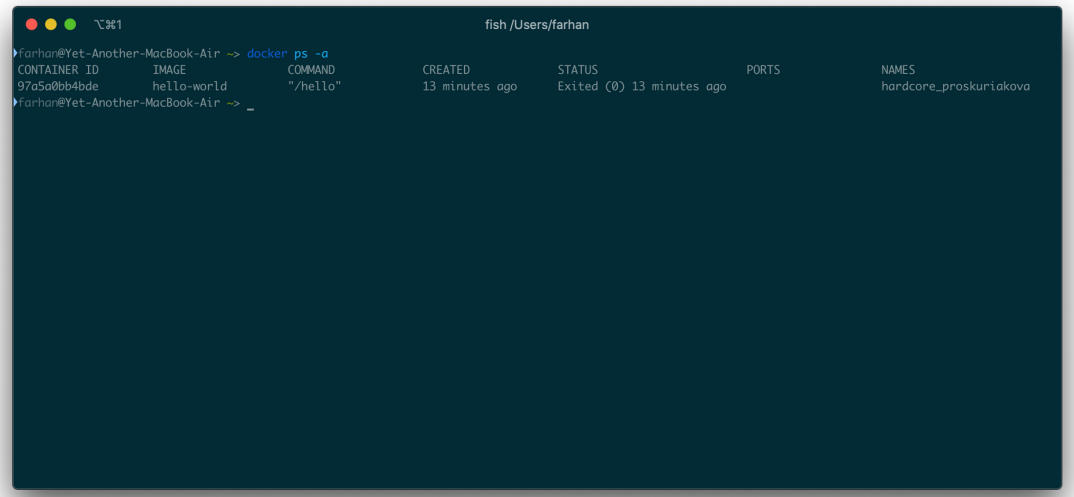
You may remember from the previous section, that the dashboard can be used for inspecting containers with ease.



Container List

It's a pretty useful tool for inspecting individual containers, but is too much for viewing a plain list of the containers. That's why there is a simpler way to do that. Execute the following command in your terminal:

And you should see a list of all the containers on your terminal.



Output from `docker ps -a` command

The `-a` or `--all` option indicates that we want to see not only the running containers but also the stopped ones. Executing `ps` without the `-a` option will list out the running containers only.

Restarting Containers

We've already used the `start` command to run a container. There is another command for starting containers called `restart`. Though the commands seem to serve the same purpose on the surface, they have a slight difference.

The `start` command starts containers that are not running. The `restart` command, however, kills a running container and starts that again. If we use `restart` with a stopped container then it'll function

Cleaning Up Dangling Containers

Containers that have exited already, remain in the system. These dangling or unnecessary containers take up space and can even create issues at later times.

There are a few ways of cleaning up containers. If we want to remove a container specifically, we can use the `rm` command. Generic syntax for this command is as follows:

```
docker rm <container id>
```

To remove a container with id `e210d4695c51`, execute following command:

```
docker rm e210d4695c51
```

And you should get the id of the removed container as output. If we want to clean up all Docker objects (images, containers, networks, build cache) we can use the following command:

```
docker system prune
```

Docker will ask for confirmation. We can use the `-f` or `--force` option to skip this confirmation step. The command will show the

amount of reclaimed space at the end of its successful execution.

Running Containers in Interactive Mode

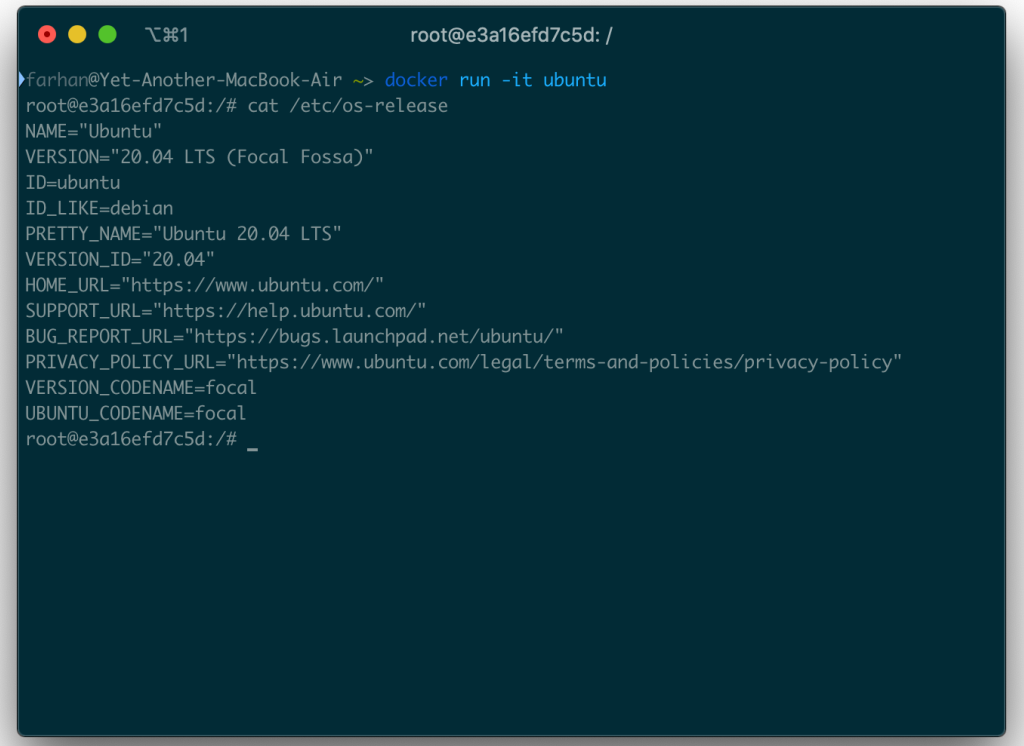
So far we've only run containers built from the hello-world image. The default command for hello-world image is to execute the single hello.c program that comes with the image.

All images are not that simple. Images can encapsulate an entire operating system inside them. Linux distributions such as Ubuntu, Fedora, Debian all have official Docker images available in the hub.

We can run Ubuntu inside a container using the official ubuntu image. If we try to run an Ubuntu container by executing `docker run ubuntu` command, we'll see nothing happens. But if we execute the command with `-it` option as follows:

```
docker run -it ubuntu
```

We should land directly on bash inside the Ubuntu container. In this bash window, we'll be able to do tasks, that we usually do in a regular Ubuntu terminal. I have printed out the OS details by executing the standard `cat /etc/os-release` command:



Output from `docker run -it ubuntu` command

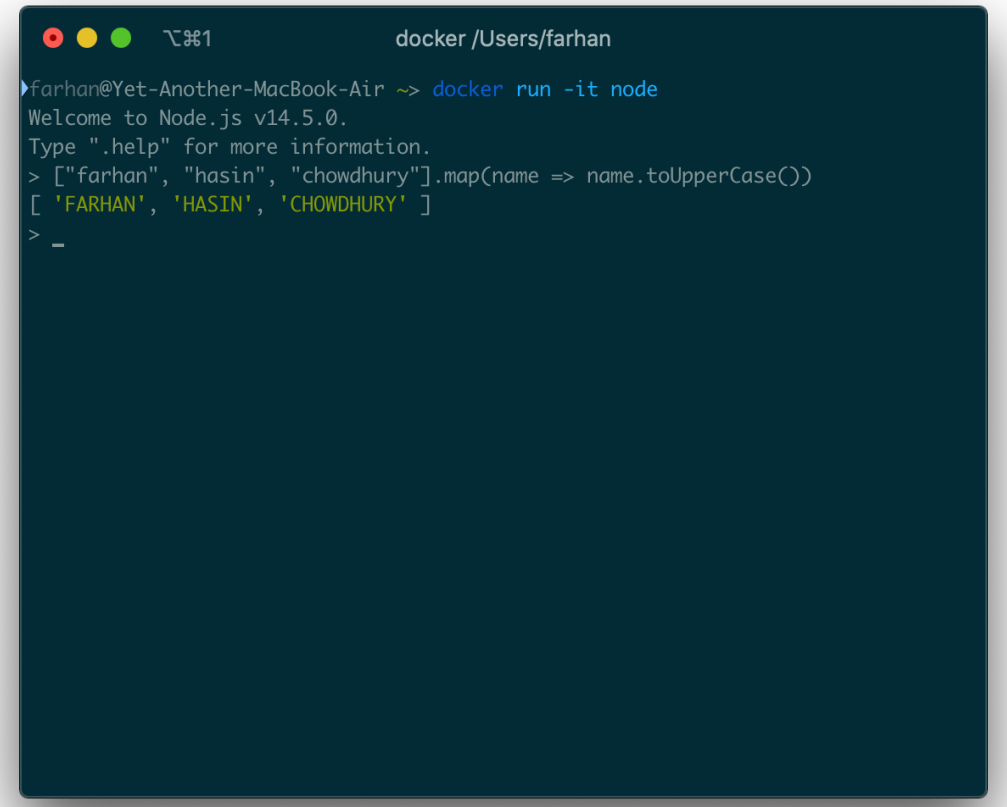
The reason behind the necessity of this `-it` option is that the Ubuntu image is configured to start bash upon startup. Bash is an interactive program – that means if we do not type in any commands, bash won't do anything.

To interact with a program that is inside a container, we have to let the container know explicitly that we want an interactive session.

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

- The `-i` option connects us to the input stream of the container, so that we can send inputs to bash.
- The `-t` option makes sure that we get some good formatting and a native terminal like experience.

We need to use the `-it` option whenever we want to run a container in interactive mode. Executing `docker run -it node` or `docker run -it python` should land us directly on the node or python REPL program.



We can not run any random container in interactive mode. To be eligible for running in interactive mode, the container has to be configured to start an interactive program on startup. Shells, REPLs, CLIs, and so on are examples of some interactive programs.

Creating Containers Using Executable Images

Up until now I've been saying that Docker images have a default command that they execute automatically. That's not true for every image. Some images are configured with an entry-point (`ENTRYPOINT`) instead of a command (`CMD`).

An entry-point allows us to configure a container that will run as an executable. Like any other regular executable, we can pass arguments to these containers. The generic syntax for passing arguments to an executable container is as follows:

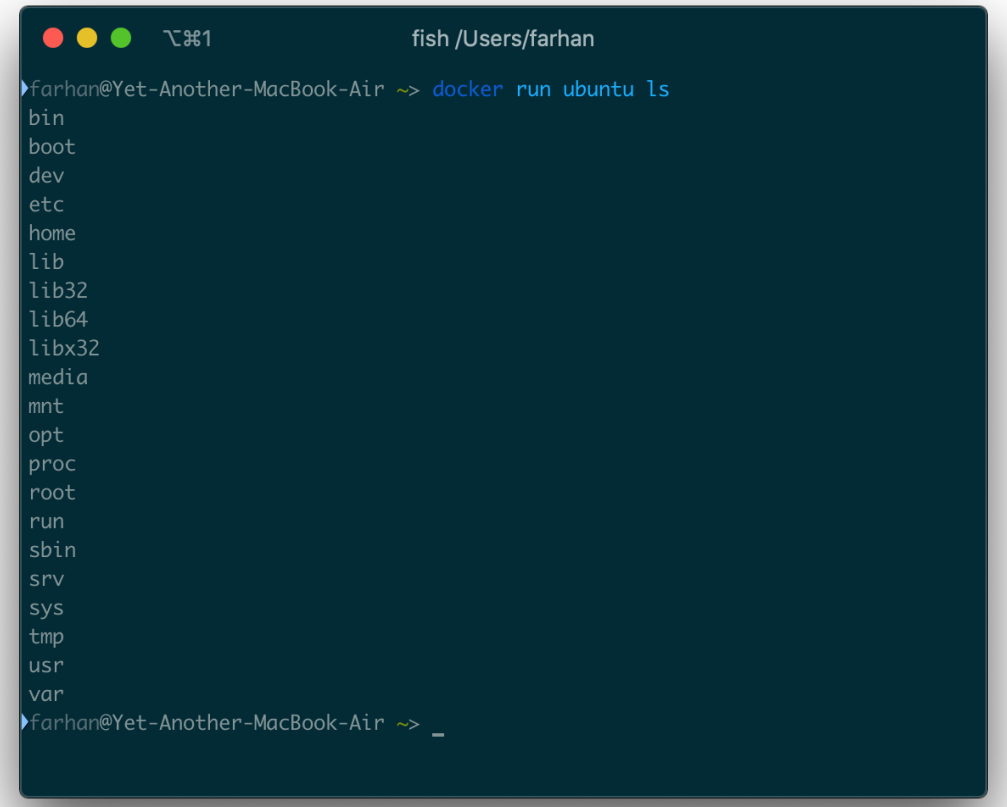
```
docker run <image name> <arguments>
```

The Ubuntu image is an executable, and the entry-point for the image is bash. Arguments passed to an executable container will be passed directly to the entry-point program. That means any argument that we pass to the the Ubuntu image will be passed directly to bash.

To see a list of all directories inside the Ubuntu container, you can pass the `ls` command as an argument.

```
docker run ubuntu ls
```

You should get a list of directories like the following:



Output from `docker run ubuntu ls` command

Notice that we're not using the `-it` option, because we don't want to interact with bash, we just want the output. We can pass any valid bash command as arguments. Like passing the `pwd` command as an argument will return the present working directory.

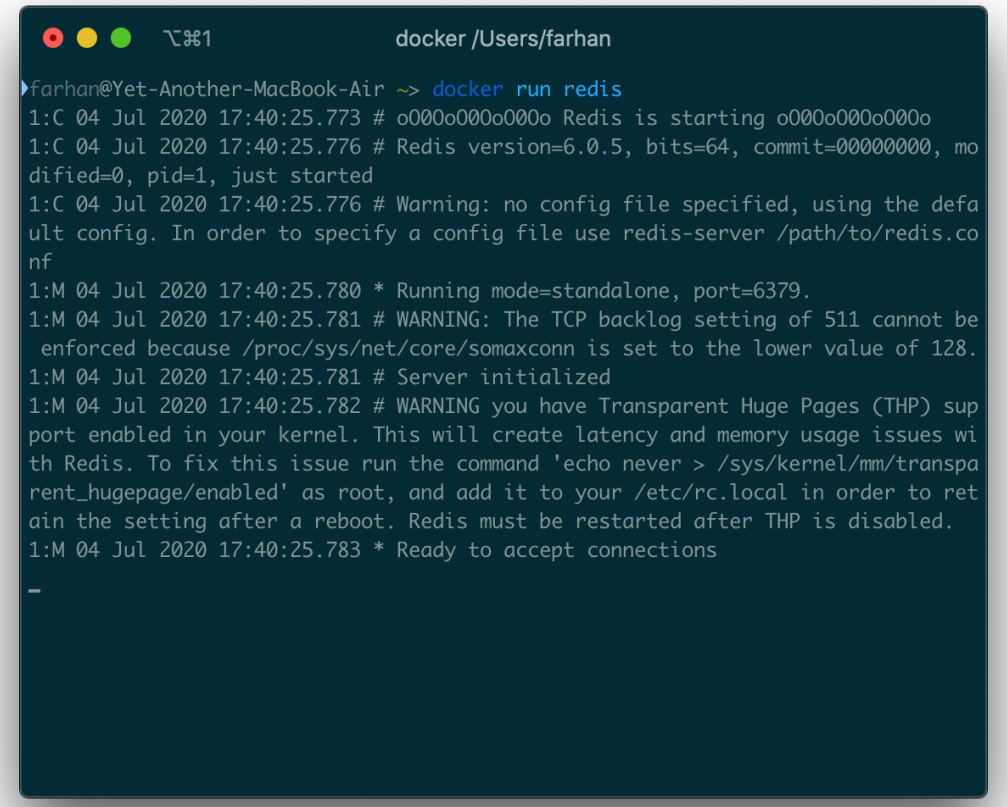
The list of valid arguments usually depends on the entry-point program itself. If the container uses the shell as entry-point, any valid shell command can be passed as arguments. If the container uses some other program as the entry-point then the arguments valid for that particular program can be passed to the container.

Running Containers in Detached Mode

Assume that you want to run a Redis server on your computer. Redis is a very fast in-memory database system, often used as cache in various applications. We can run a Redis server using the official redis image. To do that by execute the following command:

```
docker run redis
```

It may take a few moments to fetch the image from the hub and then you should see a wall of text appear on your terminal.



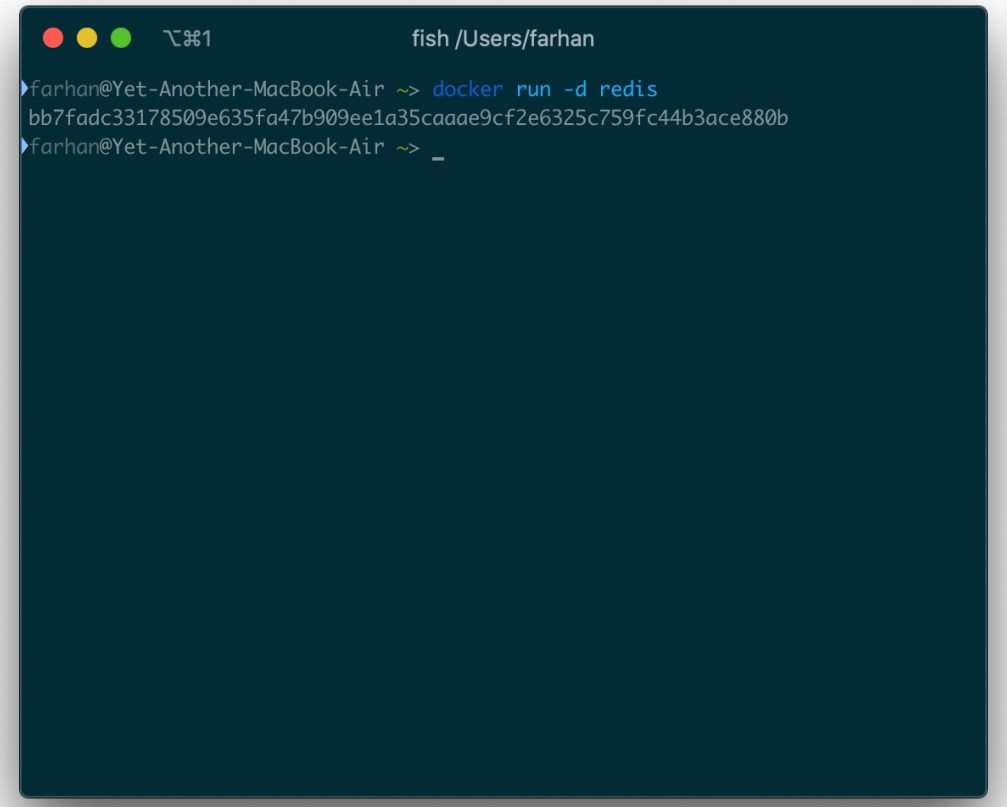
Output from `docker run redis` command

As you can see, the Redis server is running and is ready to accept connections. To keep the server running, you have to keep this terminal window open (which is a hassle in my opinion).

You can run these kind of containers in detached mode. Containers running in detach mode run in the background like a service. To detach a container, we can use the `-d` or `--detach` option. To run the container in detached mode, execute the following command:

```
docker run -d redis
```

You should get the container id as output.



Output from `docker run -d redis` command

The Redis server is now running in the background. You can inspect it using the dashboard or by using the `ps` command.

Executing Commands Inside a Running Container

Now that you have a Redis server running in the background, assume that you want to perform some operations using the `redis-cli` tool. You can not just go ahead and execute `docker run redis redis-cli`. The container is already running.

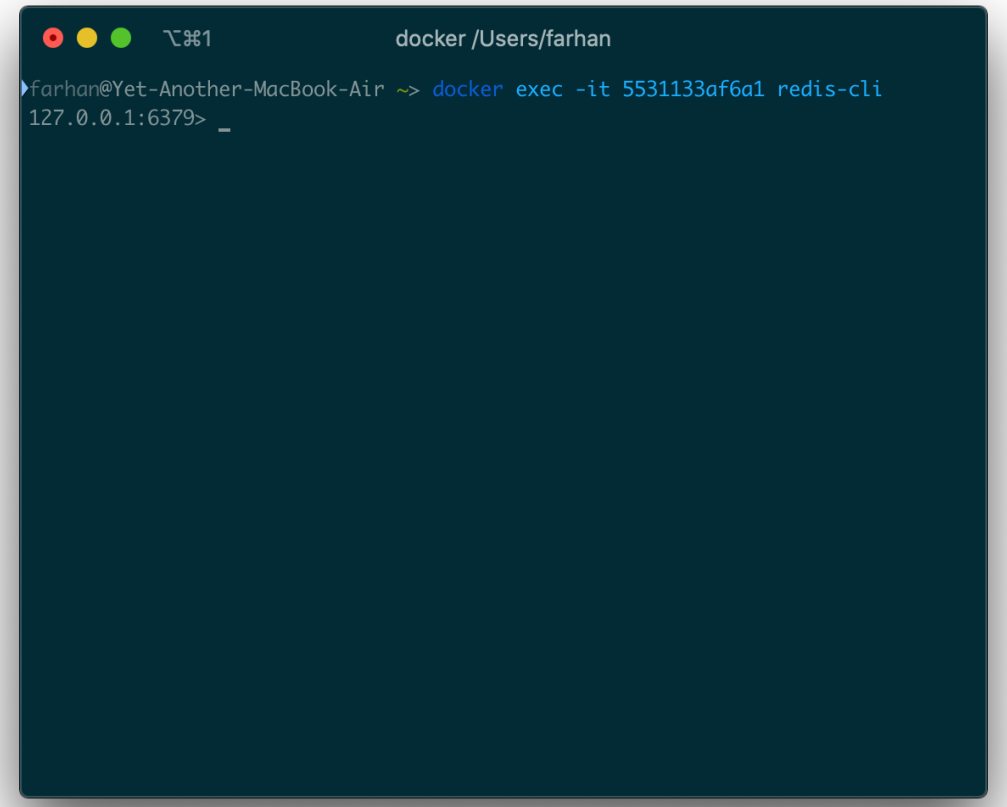
For situations like this, there is a command for executing other commands inside a running container called `exec`, and the generic syntax for this command is as follows:

```
docker exec <container id> <command>
```

If the id for the Redis container is `5531133af6a1` then the command should be as follows:

```
docker exec -it 5531133af6a1 redis-cli
```

And you should land right into the `redis-cli` program:



Output from `docker exec -it 5531133af6a1 redis-cli` command

Notice we're using the `-it` option as this is going to be an interactive session. Now you can run any valid Redis command in this window and the data will be persisted in the server.

You can exit simply by pressing `ctrl + c` key combination or closing the terminal window. Keep in mind however, the server will keep running in the background even if you exit out of the CLI program.

Starting Shell Inside a Running Container

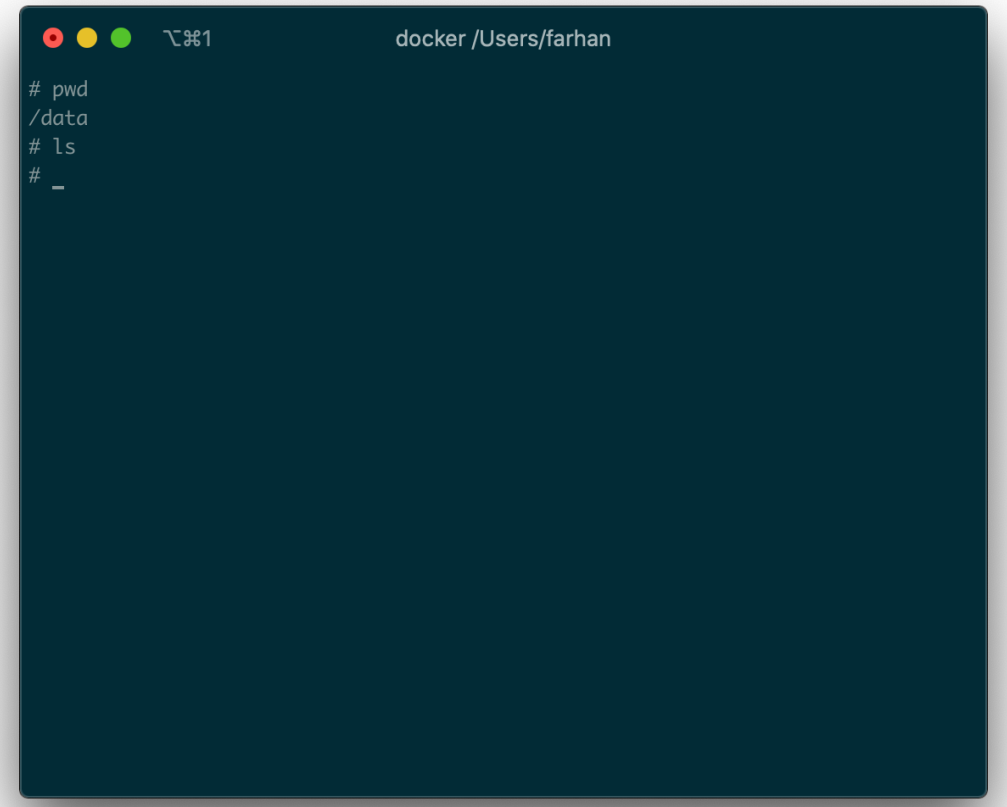
Assume that you want to use the shell inside a running container for some reason. You can do that by just using the `exec` command with `sh` being the executable like the following command:

```
docker exec -it <container id> sh
```

If the id of the redis container is `5531133af6a1` the, execute the following command to start a shell inside the container:

```
docker run exec -it 5531133af6a1 sh
```

You should land directly on a shell inside the container.

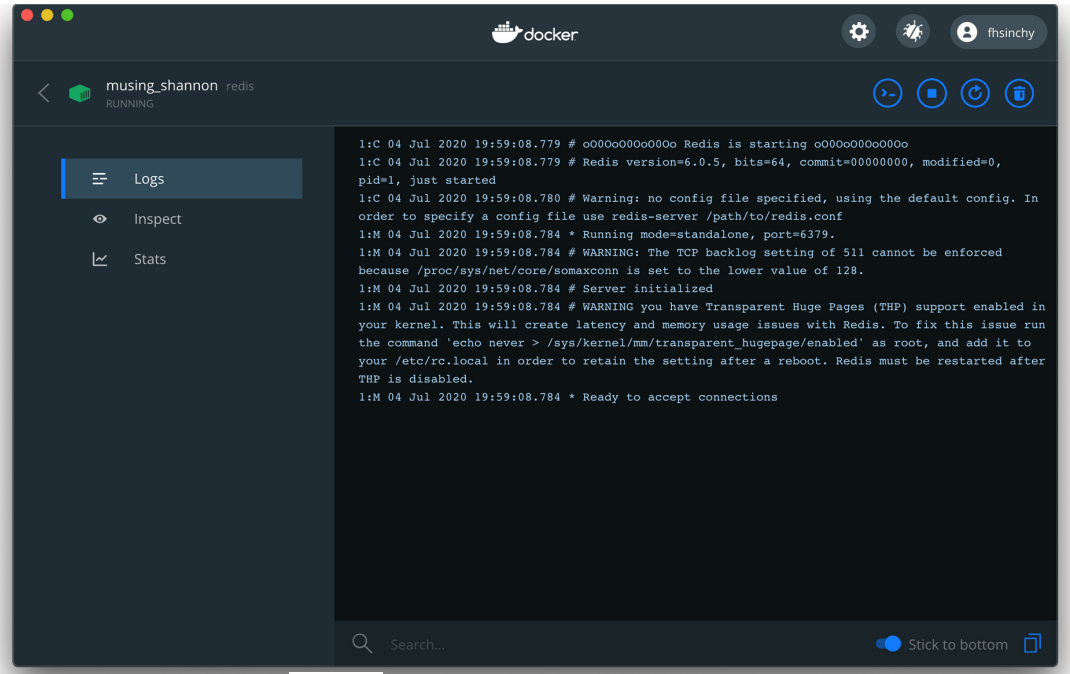


Output from `docker run exec -it 5531133af6a1 sh` command

You can execute any valid shell command here.

Accessing Logs From a Running Container

If we want to view logs from a container, the dashboard can be really helpful.



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

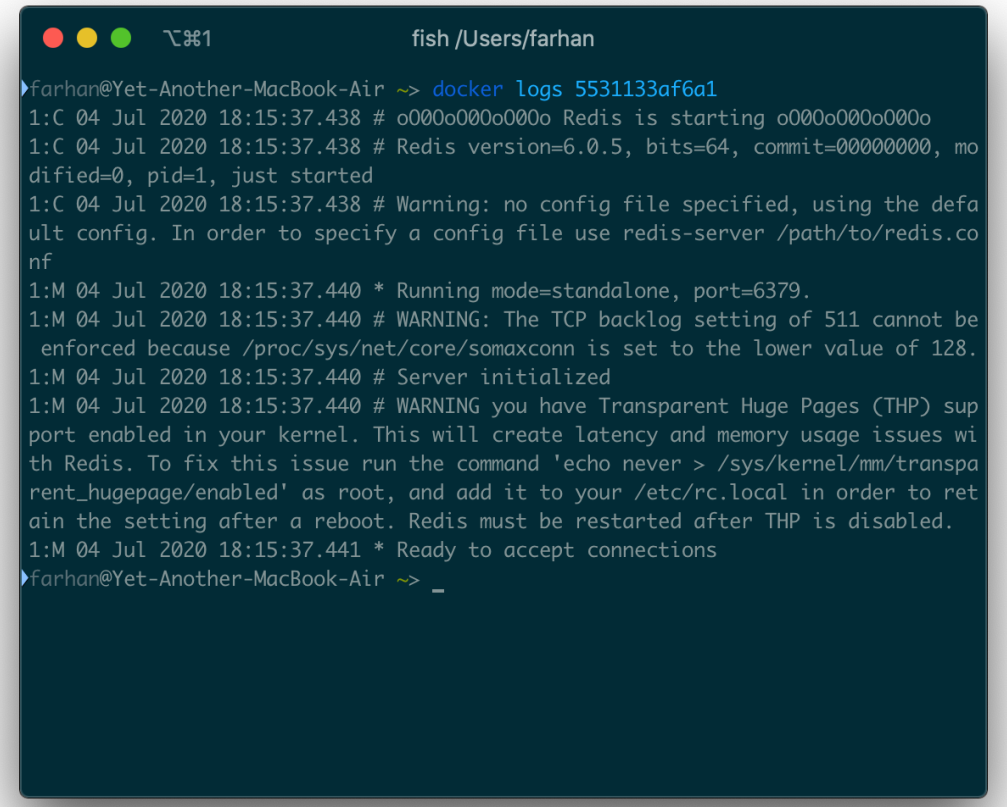
Logs in the Docker Dashboard

```
docker logs <container id>
```

If the id for Redis container is `5531133af6a1` then execute following command to access the logs from the container:

```
docker logs 5531133af6a1
```

You should see a wall of text appear on your terminal window.



Output from `docker logs 5531133af6a1` command

This is just a portion from the log output. You can kind of hook into the output stream of the container 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` key combination or closing the window. The container will keep running even if you exit out of the log window.

Stopping or Killing a Running Container

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

There are two commands for stopping a running container:

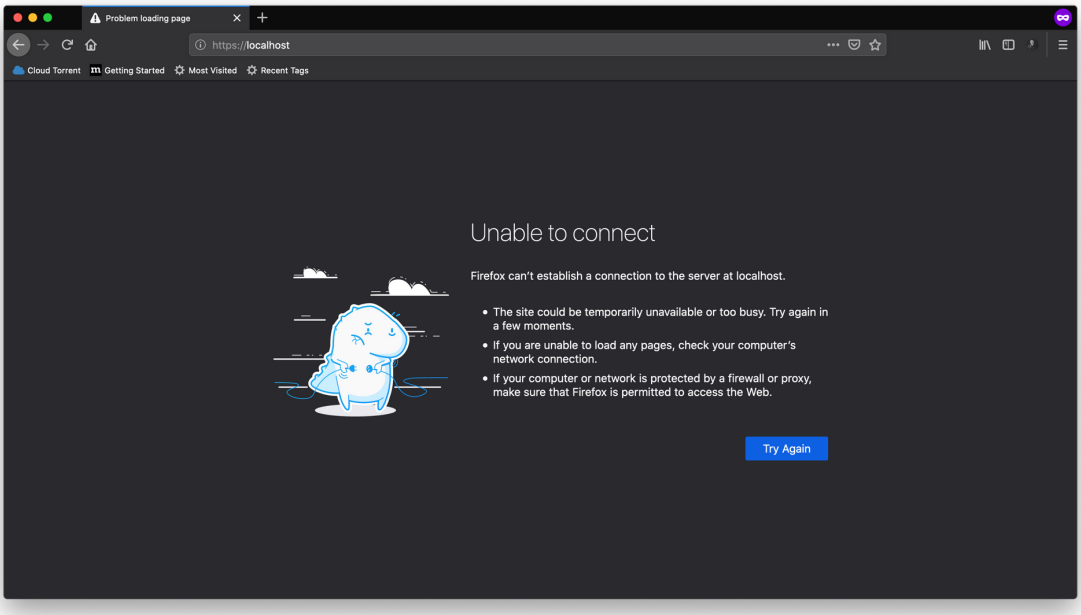
- `docker stop <container id>` - attempts to stop the container gracefully by sending a `SIGTERM` signal to the container. If the container doesn't stop within a grace period, a `SIGKILL` signal is sent.
- `docker kill <container id>` - stops the container immediately by sending a `SIGKILL` signal. A `SIGKILL` signal can not be ignored by a recipient.

To stop a container with id `bb7fadc33178` execute `docker stop bb7fadc33178` command. Using `docker kill bb7fadc33178` will terminate the container immediately without giving a chance to clean up.

Mapping Ports

Assume that you want to run an instance of the popular Nginx web server. You can do that by using the official nginx image. Execute the following command to run a container:

Nginx is meant to be kept running, so you may as well use the `-d` or `--detach` option. By default Nginx runs on port 80. But if you try to access `http://localhost:80` you should see something like the following:



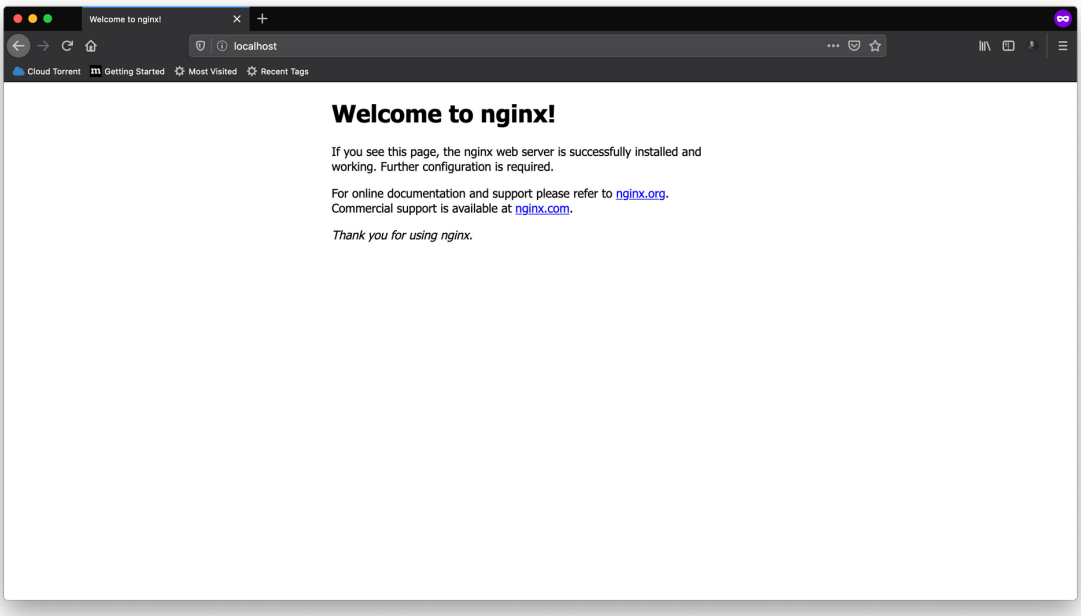
http://localhost:80

That's because Nginx is running on port 80 inside the container. Containers are isolated environments and your host system knows nothing about what's going on inside a container.

To access a port that is inside a container, you need to map that port to a port on the host system. You can do that by using the `-p` or `--port` option with the `docker run` command. Generic syntax for this option is as follows:

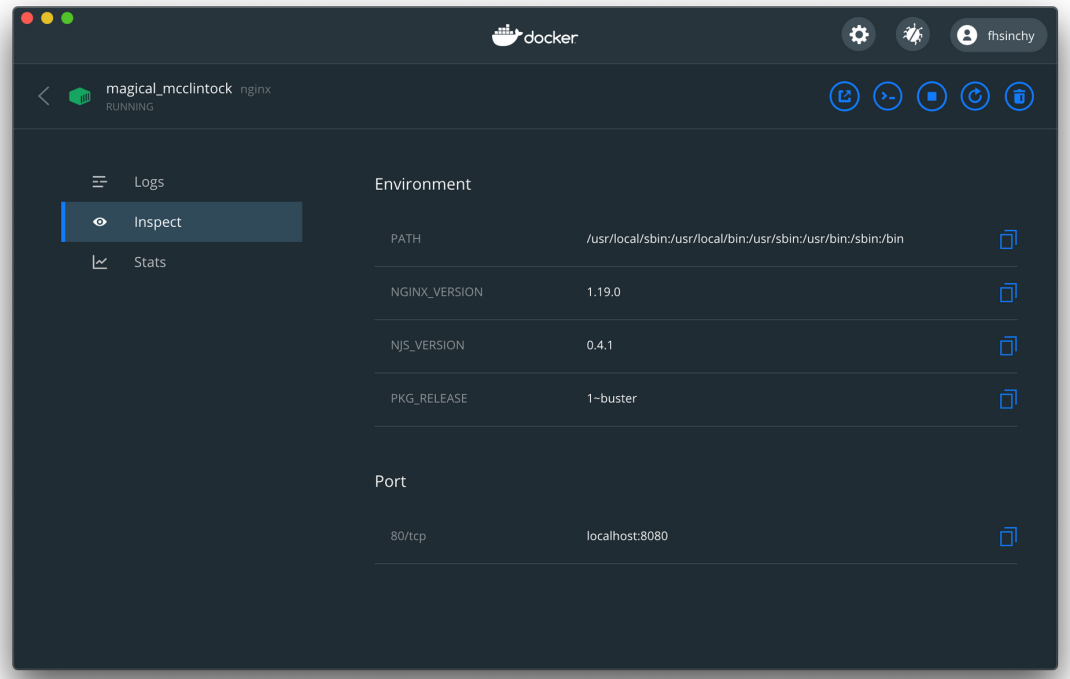
```
docker run -p <host port:container port> nginx
```

Executing `docker run -p 80:80 nginx` will map port 80 on the host machine to port 80 of the container. Now try accessing `http://localhost:80` address:



http://localhost:80

If you execute `docker run -p 8080:80 nginx` instead of `80:80` the Nginx server will be available on port 8080 of the host machine. If you forget the port number after a while you can use the dashboard to have a look at it:



Port mapping in the Docker Dashboard

The *Inspect* tab contains information regarding the port mappings. As you can see, I've mapped port 80 from the container to port 8080 of the host system.

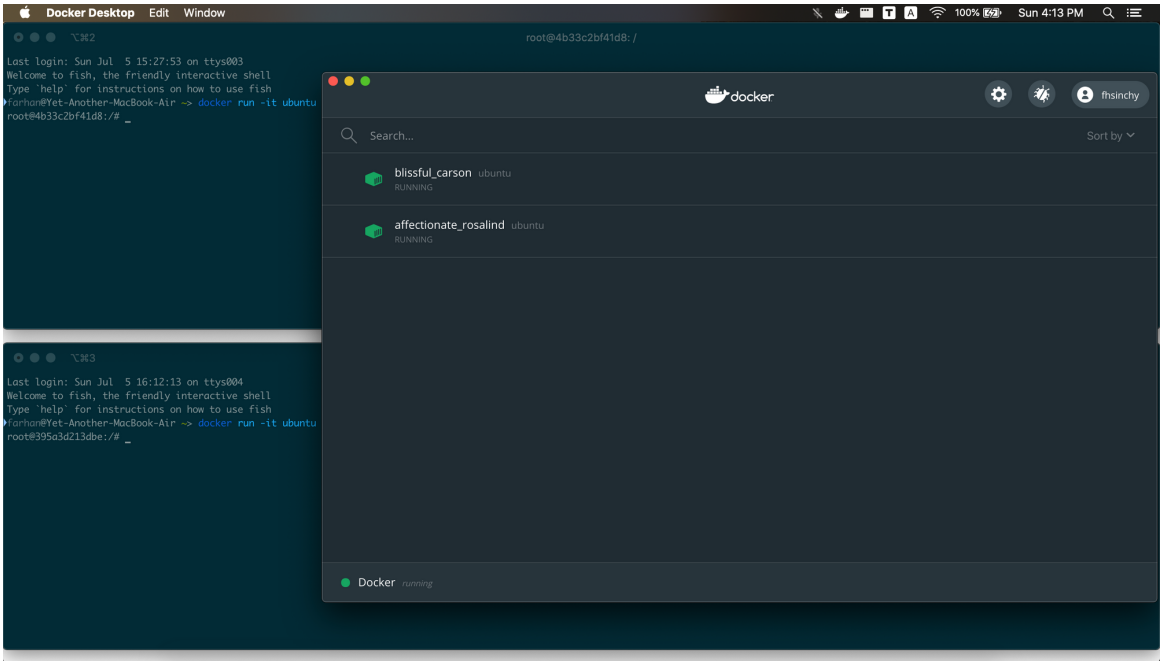
Demonstration of Container Isolation

From the moment that I introduced you to the concept of a container, I've been saying that containers are isolated environments. When I say isolated, I not only mean from the host system but also from other containers.

In this section, we'll do a little experiment to understand this isolation stuff. Open up two terminal windows and execute and run two Ubuntu container instances using the following command:

```
docker run -it ubuntu
```

If you open up the dashboard you should see two Ubuntu containers running:



Running two Ubuntu containers

Now on the upper window, execute following command:

```
mkdir hello-world
```

The `mkdir` command creates a new directory. Now to see the list of directories in both containers execute the `ls` command inside both of them:

As you can see, the `hello-world` directory exists inside the container open on the upper terminal window and not on the lower one. It goes to prove that the containers although created from the same image are isolated from each other.

Output from the `ls` command inside both containers

This is something important to understand. Assume a scenario where you've been working inside a container for a while. Then you `stop` the container and on the next day you execute `docker run -it ubuntu` once again. You'll see all your works have been lost.

I hope you remember from a previous sub-section, the `run` command creates and starts a new container every time. So remember to start previously created containers using the `start` command and not the `run` command.

Creating Custom Images

Now that you have a solid understanding of the many ways you can manipulate a container using the Docker client, it's time to learn how to make custom images.

In this section, you'll learn many important concepts regarding

building images, creating containers from them, and sharing them with others.

> I suggest that you install Visual Studio Code with the official Docker Extension before going into the subsequent sub-sections.

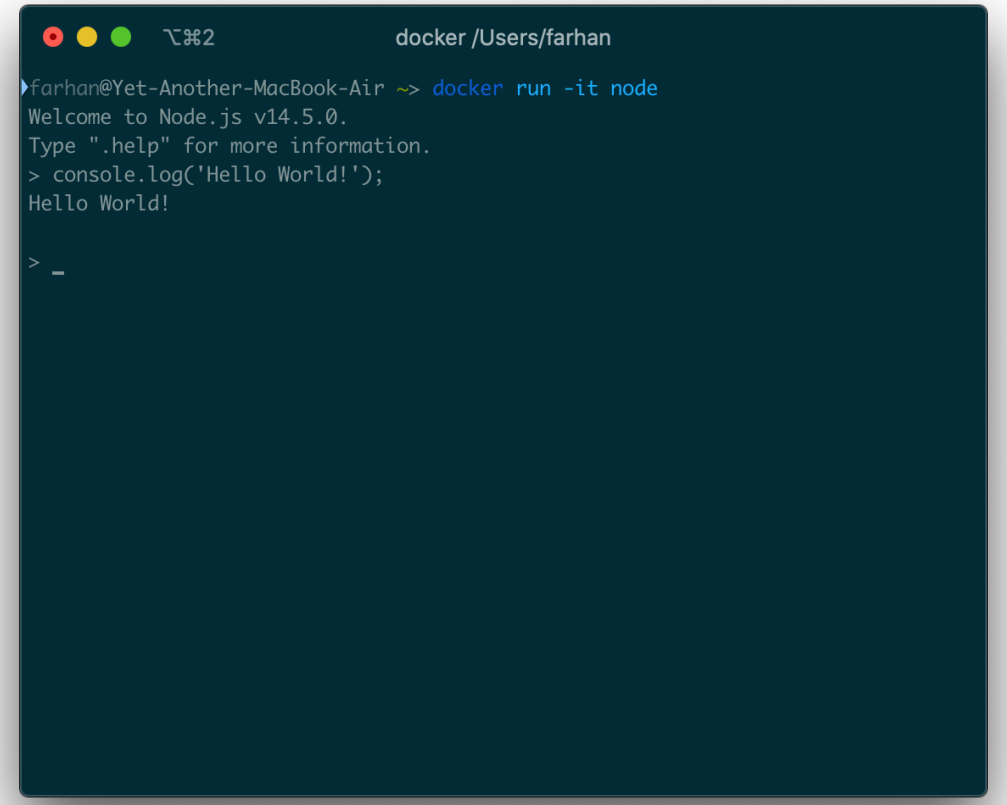
Image Creation Basics

In this sub-section we'll focus on the structure of a Dockerfile and the common instructions. A Dockerfile is a text document, containing a set of instructions for the Docker daemon to follow and build an image.

To understand the basics of building images we'll create a very simple custom Node image. Before we begin, I would like to show you how the official node image works. Execute the following command to run a container:

```
docker run -it node
```

The Node image is configured to start the Node REPL on startup. The REPL is an interactive program hence the usage of `-it` flag.



Output from the `docker run -it node` command

You can execute any valid JavaScript code here. We'll create a custom node image that functions just like that.

To start, create a new directory anywhere in your computer and create a new file named `Dockerfile` inside there. Open up the project folder inside a code editor and put following code in the `Dockerfile` :

```
FROM ubuntu

RUN apt-get update
RUN apt-get install nodejs -y

CMD [ "node" ]
```

I hope you remember from a previous sub-section that images have multiple layers. Each line in a `Dockerfile` is an instruction and each instruction creates a new layer.

Let me break down the `Dockerfile` line by line for you:

```
FROM ubuntu
```

Every valid `Dockerfile` must start with a `FROM` instruction. This instruction starts a new build stage and sets the base image. By setting `ubuntu` as the base image, we say that we want all the functionalities from the Ubuntu image to be available inside our image.

Now that we have the Ubuntu functionalities available in our image, we can use the Ubuntu package manager (`apt-get`) to install Node.

```
RUN apt-get update
RUN apt-get install nodejs -y
```

The `RUN` instruction will execute any commands in a new layer on top

instructions, we can refer to Node, because we've installed that in this step.

```
CMD [ "node" ]
```

The purpose of a `CMD` instruction is to provide defaults for an executing container. These defaults can include an executable, or you can omit the executable, in which case you must specify an `ENTRYPOINT` instruction. There can be only one `CMD` instruction in a Dockerfile. Also, single quotes are not valid.

Now to build an image from this `Dockerfile`, we'll use the `build` command. The generic syntax for the command is as follows:

```
docker build <build context>
```

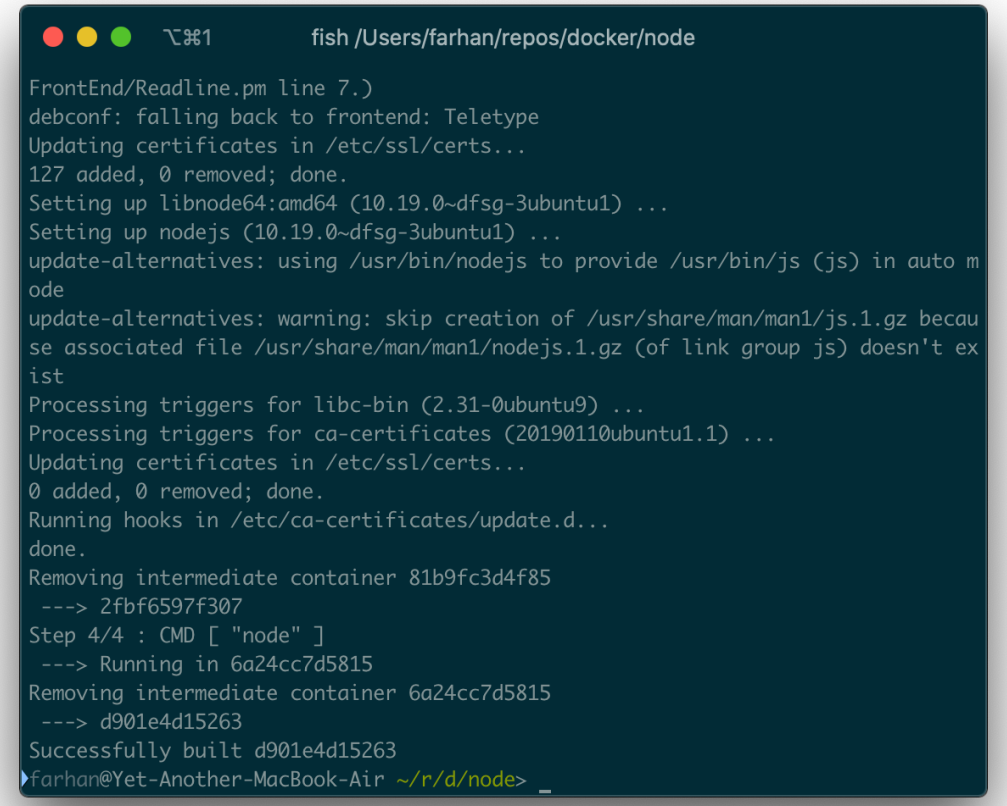
The `build` command requires a Dockerfile and the build's context. The context is the set of files and directories located in the specified location. Docker will look for a `Dockerfile` in the context and use that to build the image.

Open up a terminal window inside that directory and execute the following command:

```
docker build .
```

We're passing `.` as the build context which means the current directory. If you put the `Dockerfile` inside another directory like `/src/Dockerfile`, then the context will be `./src`.

The build process may take some time to finish. Once done, you should see a wall of text in your terminal:

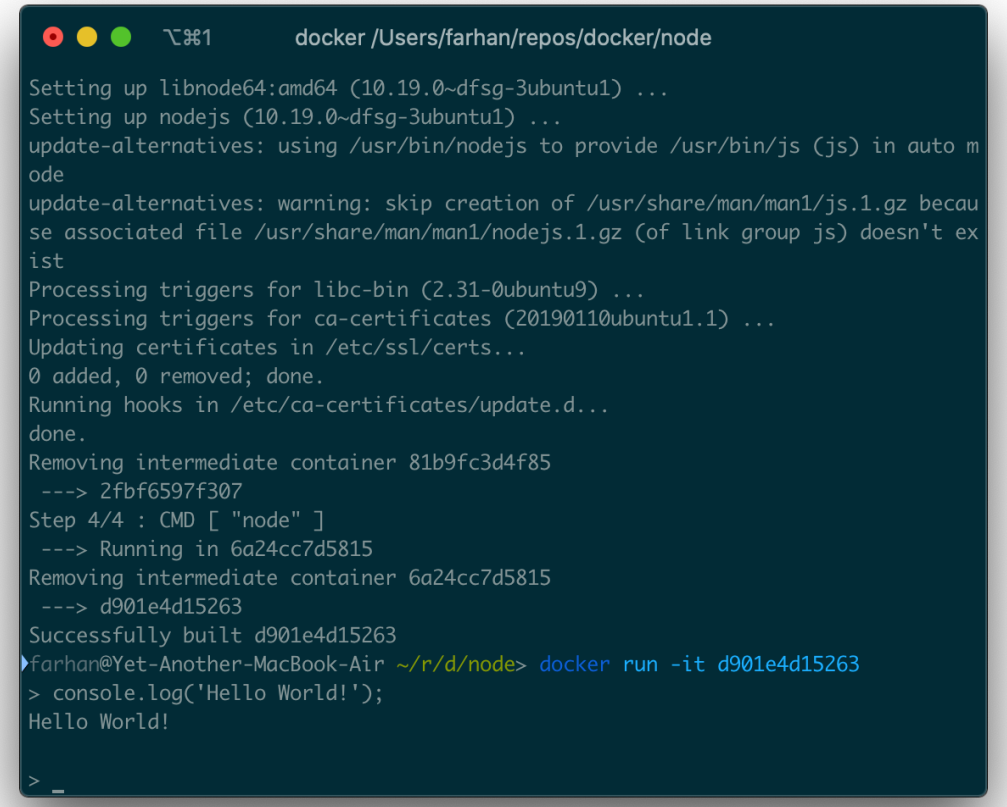


Output from `docker build .` command

If everything goes fine, you should see something like `Successfully built d901e4d15263` at the end. This random string is the image id and not container id. You can execute the `run` command with this image id to create and start a new container.

```
docker run -it d901e4d15263
```

Remember, the Node REPL is an interactive program, so the `-it` option is necessary. Once you've run the command you should land on the Node REPL:



Node REPL inside our custom image

You can execute any valid JavaScript code here, just like the official Node image.

Creating an Executable Image

I hope you remember the concept of an executable image from a previous sub-section. Images that can take additional arguments just like a regular executable. In this sub-section, you'll learn how to make one.

We'll create a custom bash image and will pass arguments like we did

with the Ubuntu image in a previous sub-section. Start by creating a `Dockerfile` inside an empty directory and put following code in that:

```
FROM alpine

RUN apk add --update bash

ENTRYPOINT [ "bash" ]
```

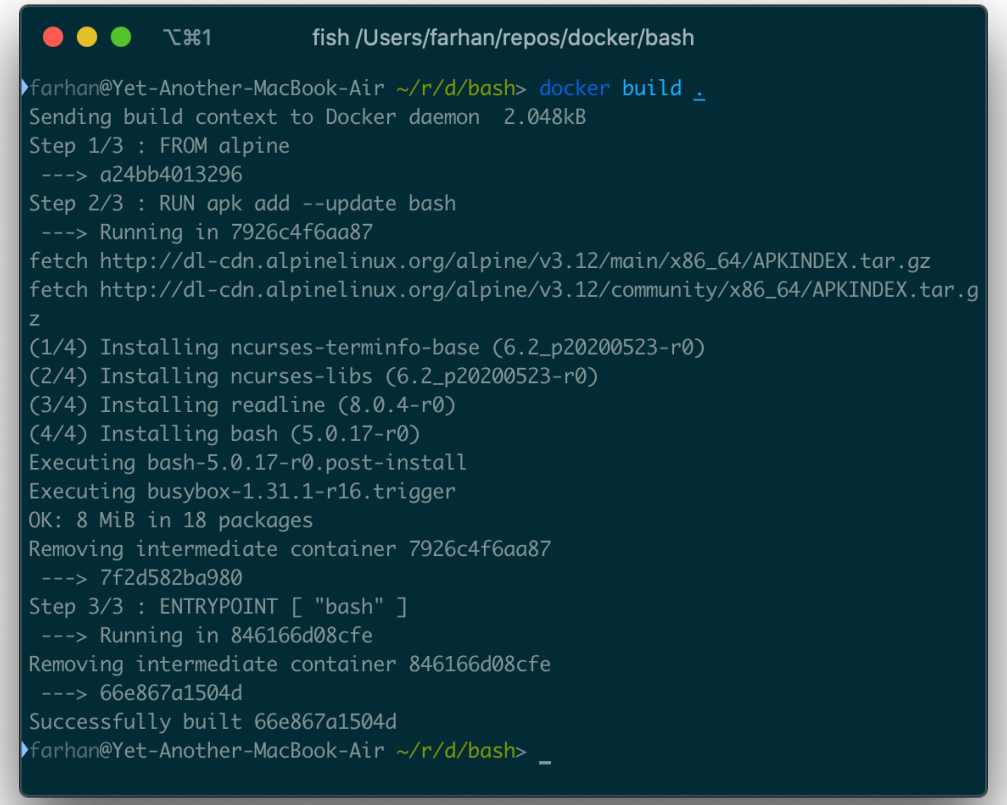
We're using the alpine image as the base. Alpine Linux is a security-oriented, lightweight Linux distribution.

Alpine doesn't come with bash by default. So on the second line we install bash using Alpine package manager, `apk`. `apk` for Alpine is what `apt-get` is for Ubuntu. The last instruction sets `bash` as the entry-point for this image. As you can see, the `ENTRYPOINT` instruction is identical to the `CMD` instruction.

To build the image execute following command:

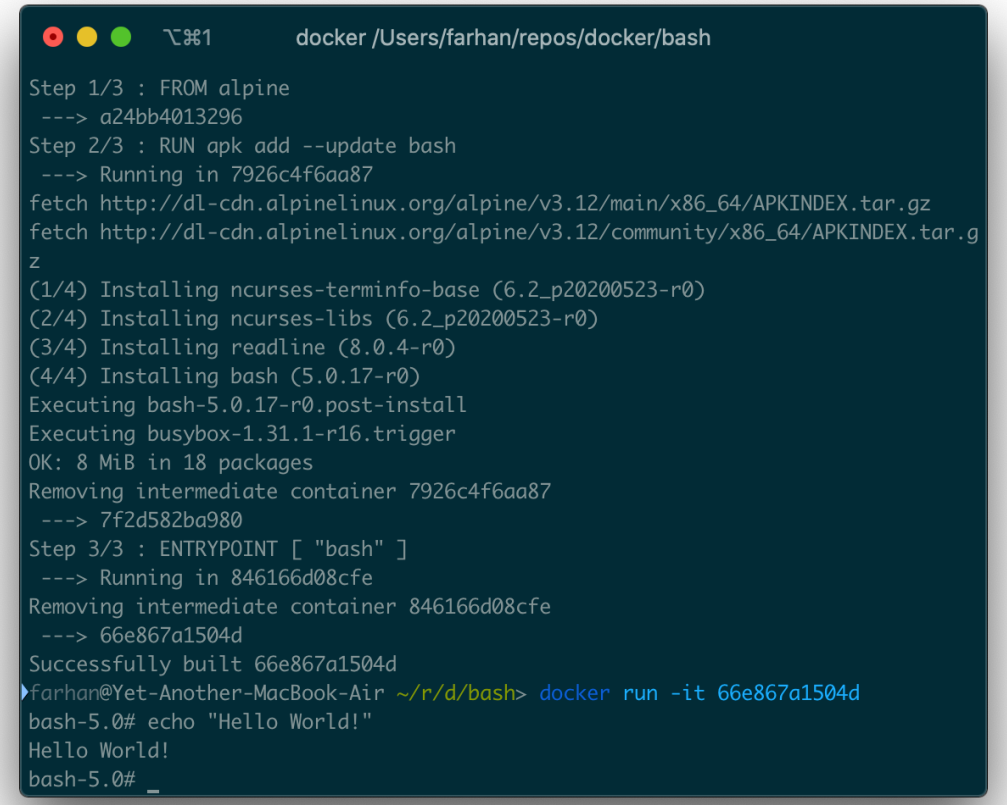
```
docker build .
```

The build process may take some time. Once done, you should get the newly created image id:



Output from `docker build .` command

You can run a container from the resultant image with the `run` command. This image has an interactive entry-point, so make sure you use the `-it` option.

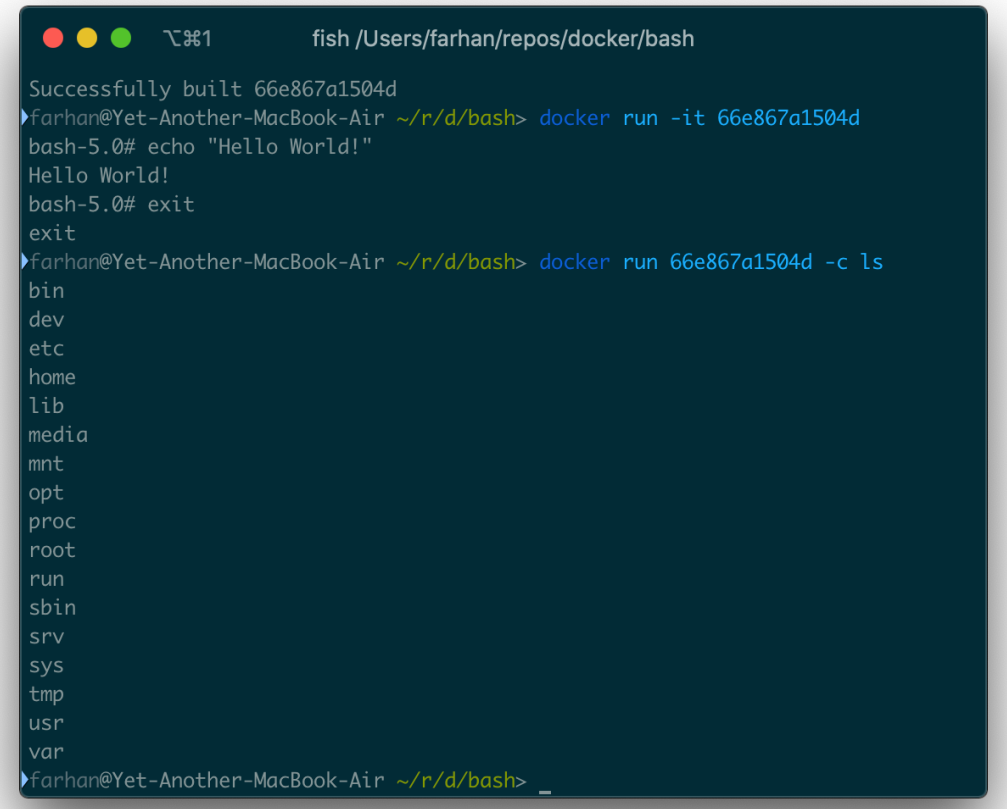


Bash running inside our custom image

Now you can pass any argument to this container just like you did with the Ubuntu container. To see a list of all files and directories, you can execute following command:

```
docker run 66e867a1504d -c ls
```

The `-c ls` option will be passed directly to bash and should return a list of directories inside the container:



Output from `docker run 66e867a1504d -c ls` command

The `-c` option has nothing to do with Docker client. It's a bash command line option. It reads commands from subsequent strings.

Containerizing an Express Application

So far we've only created images that contain no additional files. In this sub-section you'll learn how to containerize a project with source files in it.

If you've cloned the project code repository, then go inside the `express-api` directory. This is a REST API that runs on port 3000 and returns a simple JSON payload when hit.

To run this application you need to go through following steps:

1. Install necessary dependencies by executing `npm install`.
2. Start the application by executing `npm run start`.

instructions, you need to go through the following steps:

1. Use a base image that allows you to run Node applications.
2. Copy the `package.json` file and install the dependencies by executing `npm run install`.
3. Copy all necessary project files.
4. Start the application by executing `npm run start`.

Now, create a new `Dockerfile` inside the project directory and put following content in it:

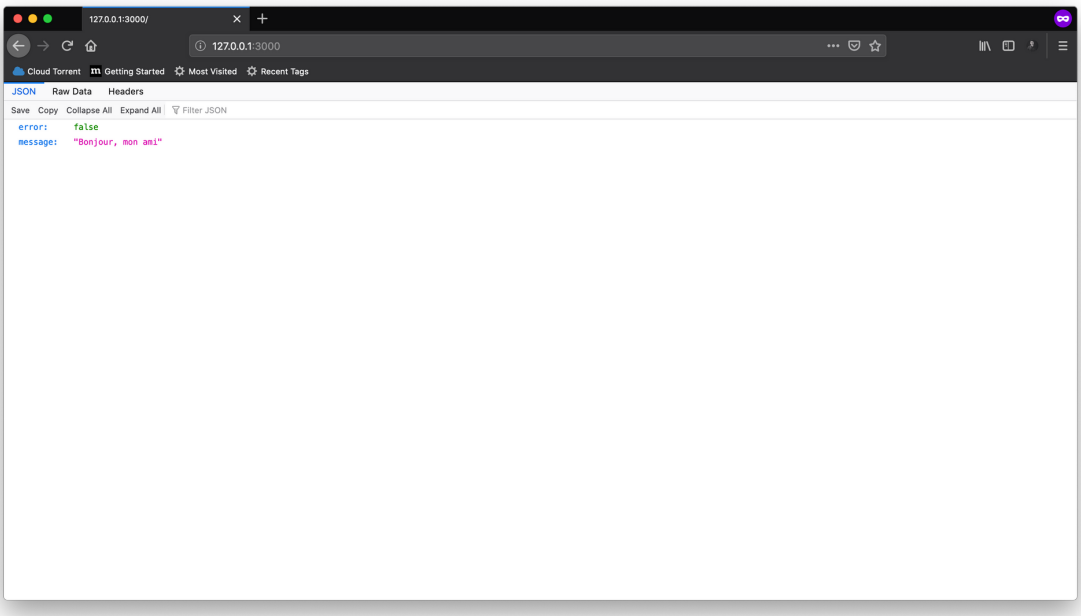
```
FROM node

WORKDIR /usr/app

COPY ./package.json ./
RUN npm install

COPY . .

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

We're using Node as our base image. The `WORKDIR` instruction sets the working directory for any `RUN`, `CMD`, `ENTRYPOINT`, `COPY` and `ADD` instructions that follow it in the `Dockerfile`. It's kind of like `cd` 'ing into the directory.

The `COPY` instruction will copy the `./package.json` to the working directory. As we have set the working directory on the previous line, `.` will refer to `/usr/app` inside the container. Once the `package.json` has been copied, we then install all the necessary dependencies using the `RUN` instruction.

In the `CMD` instruction, we set `npm` as the executable and pass `run` and `start` as arguments. The instruction will be interpreted as `npm run start` inside the container.

Now build the image with `docker build .` and use the resultant image id to run a new container. The application runs on port 3000 inside the container, so don't forget to map that.



Response from `express-api` application

Once you've successfully run the container, visit `http://127.0.0.1:3000` and you should see a simple JSON response. Replace the 3000 if you've used some other port from the host system.

Working with Volumes

In this sub-section I'll be presenting a very common scenario. Assume that you're working on a fancy front-end application with React or

`ite-counter` directory. This is a simple Vue application initialized with `npm init vite-app` command.

To run this application in development mode, we need to go through the following steps:

1. Install necessary dependencies by executing `npm install`.
2. Start the application in development mode by executing `npm run dev`.

To replicate the above mentioned process using Dockerfile instructions, we need to go through the following steps:

1. Use a base image that allows you to run Node applications.
2. Copy the `package.json` file and install the dependencies by executing `npm run install`.
3. Copy all necessary project files.
4. Start the application in development mode by executing `npm run dev`.

In there create a new `Dockerfile.dev` and put following content in it:

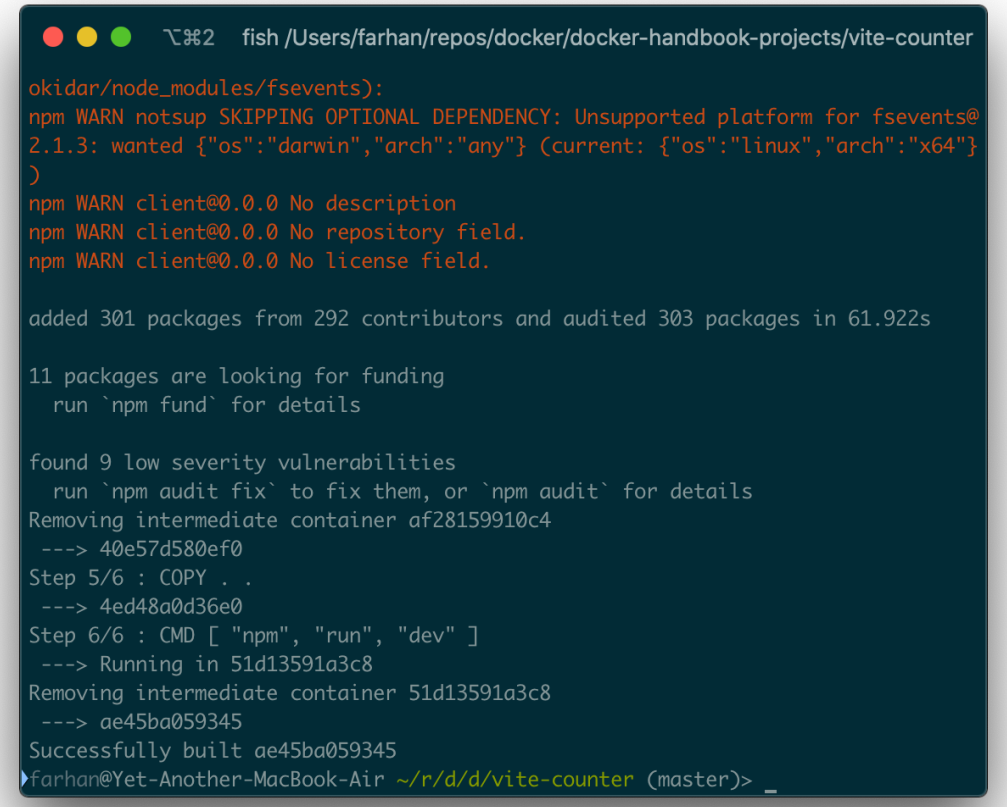
```
FROM node

WORKDIR /usr/app

COPY ./package.json ./
RUN npm install

COPY . .

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

Nothing fancy here. We're copying the `package.json` file, installing the dependencies, copying the project files and starting the development server by executing `npm run dev`.

Build the image by executing following command:

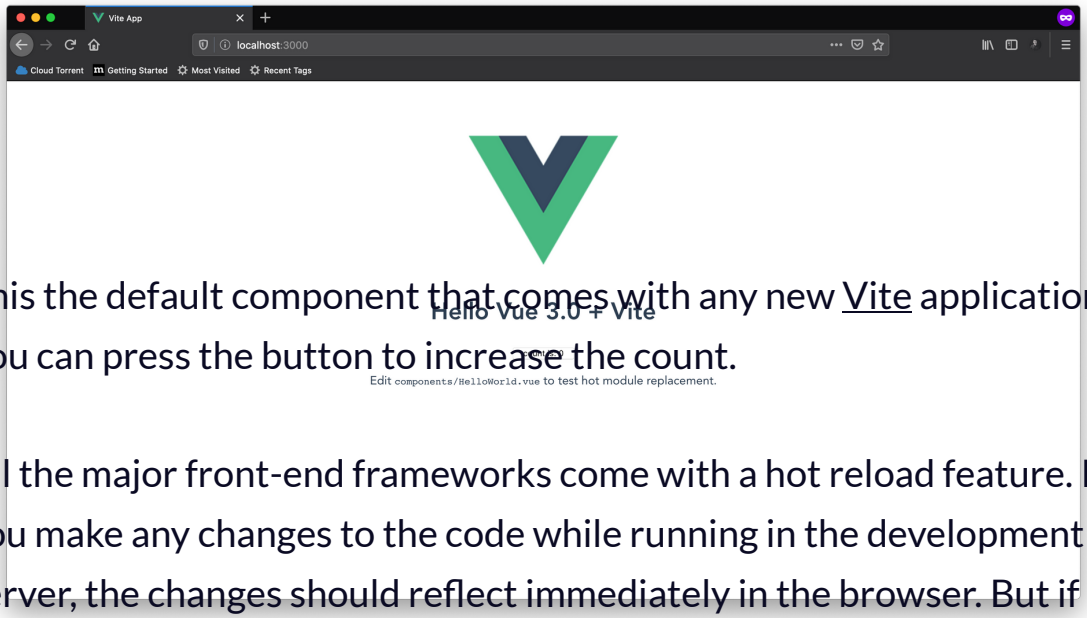
```
docker build -f Dockerfile.dev .
```

Docker is programmed to look for a `Dockerfile` within the build's context. But we've named our file `Dockerfile.dev`, thus we have to use the `-f` or `--file` option and let Docker know the filename. The `.` at the end indicates the context, just like before.



Output from `docker build -f Dockerfile.dev .` command.

The development server runs on port 3000 inside the container, so make sure you map the port while creating and starting a container. I can access the application by visiting `http://127.0.0.1:3000` on my system.



The `vite-counter` app running without volumes

This the default component that comes with any new Vite application. You can press the button to increase the count.

All the major front-end frameworks come with a hot reload feature. If you make any changes to the code while running in the development server, the changes should reflect immediately in the browser. But if you go ahead and make any changes to the code in this project, you'll see no changes in the browser.

Well, the reason is pretty straightforward. When you're making changes in the code, you are changing the code in your host system, not the copy inside the container.

There is a solution to this problem. Instead of making a copy of the source code inside the container what we can do is, we can just let the container access the files from our host directly.

To do that, Docker has an option called `-v` or `--volume` for the `run` command. Generic syntax for the `volume` option is as follows:

```
docker run -v <absolute path to host directory>:<absolute path to contair
```

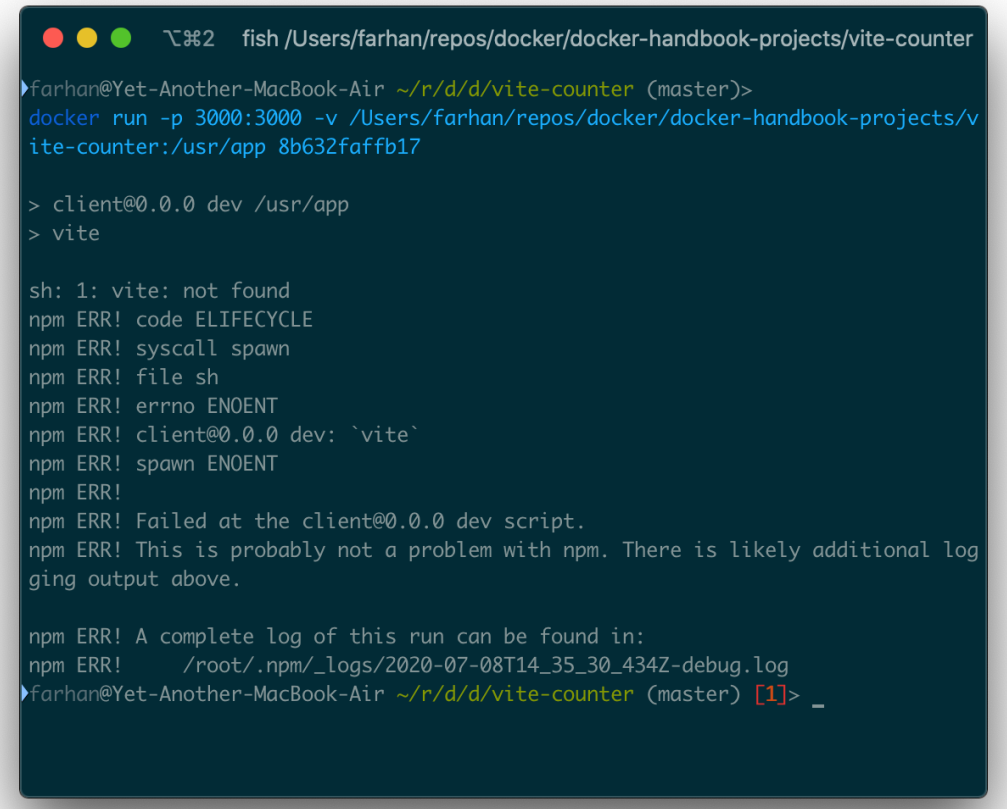
You can use the `pwd` shell command to get the absolute path of the current directory. My host directory path is `/Users/farhan/repos/doc`

working directory path is `/usr/app` and the image id is `8b632faffb17`. So my command will be as follows:

```
docker run -p 3000:3000 -v /Users/farhan/repos/docker/docker-handbook-prc
```

If you execute the above command, you'll be presented with an error saying `sh: 1: vite: not found`, which means that the dependencies are not present inside the container.

If you do not get such an error, that means you've installed the dependencies in your host system. Delete the `node_modules` folder in your local system and try again.



`sh: 1: vite: not found` error output

But if you look into the `Dockerfile.dev` , at the fourth line, we've clearly written the `RUN npm install` instruction.

Let me explain why this is happening. When using volumes, the container accesses the source code directly from our host system, and as you know, we haven't installed any dependencies in the host system.

Installing the dependencies can solve the problem but isn't ideal at all. Because some dependencies get compiled from source every time you install them. And if you're using Windows or Mac as your host operating system, then the binaries built for your host operating system will not work inside a container running Linux.

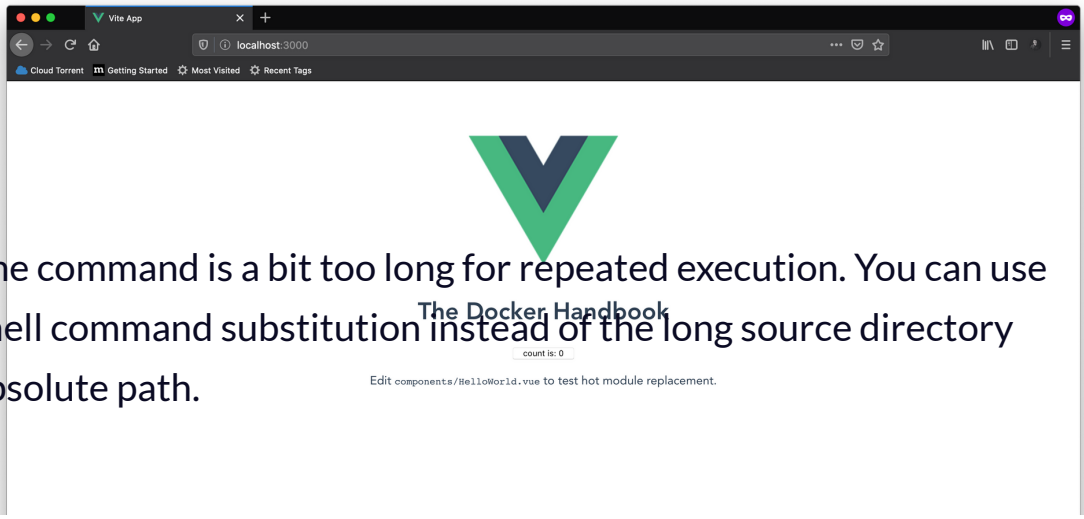
To solve this problem, you have to know about the two types of volumes Docker has

- **Named Volumes:** These volumes have a specific source from outside the container, for example `-v ($PWD):/usr/app`.
- **Anonymous Volumes:** These volumes have no specific source, for example `-v /usr/app/node_modules`. When the container is deleted, anonymous volumes remain until you clean them up manually.

To prevent the `node_modules` directory from getting overwritten, we'll have to put it inside an anonymous volume. To do that, modify the previous command as follows:

```
docker run -p 3000:3000 -v /usr/app/node_modules -v /Users/farhan/repos/c
```

The only change we've made is the addition of a new anonymous volume. Now run the command and you'll see the application running. You can even change anything and see the change immediately in the browser. I've changed the default header a bit.



The command is a bit too long for repeated execution. You can use shell command substitution instead of the long source directory absolute path.

```
docker run -p 3000:3000 -v /usr/app/node_modules -v $(pwd):/usr/app 8b632
```

The `vite-counter` app running with volumes

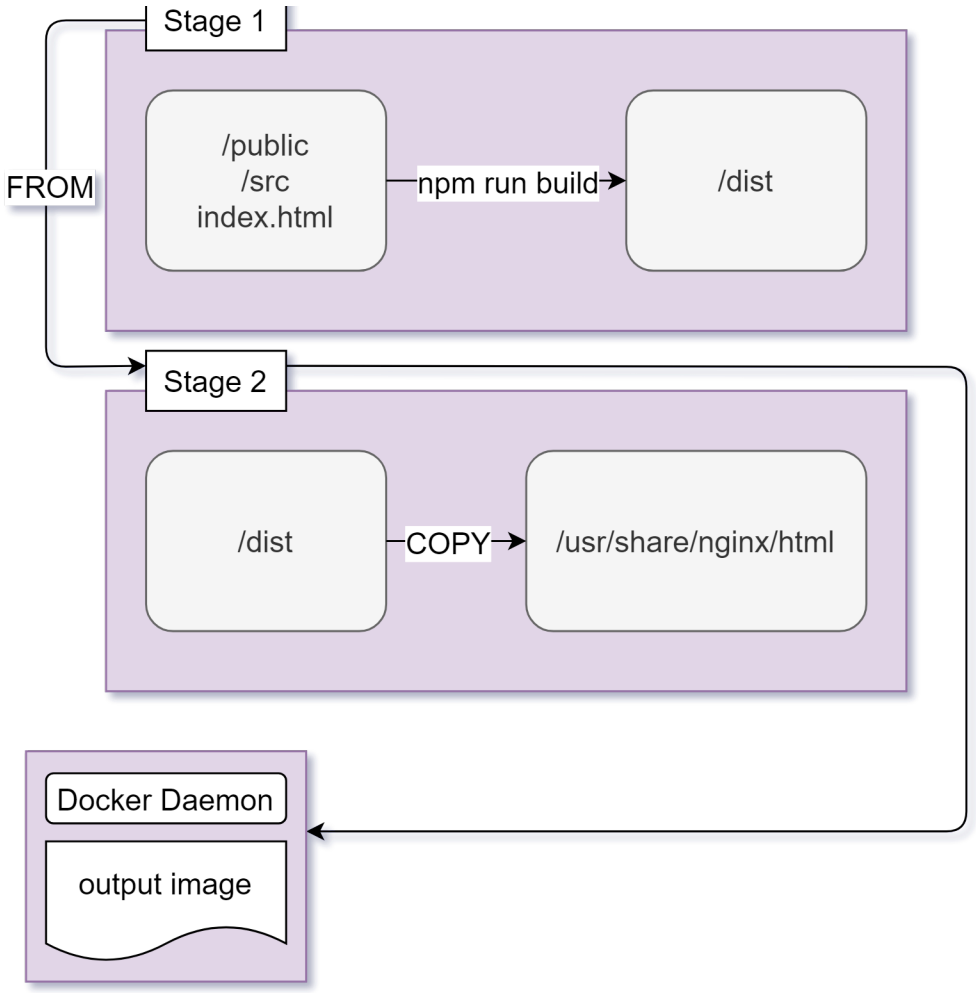
The `$(pwd)` bit will be replace with the absolute path to the present directory you're in. So make sure you've opened your terminal window inside the project folder.

Multi-staged Builds

Introduced in Docker v17.05, multi-staged build is an amazing feature. In this sub-section, you'll again work with the `vite-counter` application.

In the previous sub-section, you created `Dockerfile.dev` file, which is clearly for running the development server. Creating a production build of a Vue or React application is a perfect example of a multi-stage build process.

First let me show you how the production build will work in the following diagram:



A multi-staged build process

As you can see from the diagram, the build process has two steps or stages. They are as follows:

1. Executing `npm run build` will compile our application into a bunch of JavaScript, CSS and an `index.html` file. The production build will be available inside the `/dist` directory on the project root. Unlike the development version though, the production build doesn't come with a fancy server.
2. We'll have to use Nginx for serving the production files. We'll copy the files built in stage 1 to the default document root of Nginx and make them available.

Now if we want to see the steps like we did with our previous two projects, it should go like as follows:

1. Use a base image (node) that allows us to run Node applications.
2. Copy the `package.json` file and install the dependencies by executing `npm run install`.
3. Copy all necessary project files.
4. Make the production build by executing `npm run build`.
5. Use another base image (nginx) that allows us to run serve the production files.
6. Copy the production files from the `/dist` directory to the default document root (`/usr/share/nginx/html`).

Let's get to work now. Create a new `Dockerfile` inside the `vite-counter` project directory. Content for the `Dockerfile` is as follows:

```
FROM node as builder

WORKDIR /usr/app

COPY ./package.json ./
RUN npm install

COPY . .

RUN npm run build

FROM nginx

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

The first thing that you might have noticed is the multiple `FROM` instructions. Multi-staged build process allows the usage of multiple `FROM` instructions. The first `FROM` instruction sets `node` as the base image, installs dependencies, copies all project files and executes `npm run build`. We're calling the first stage `builder`.

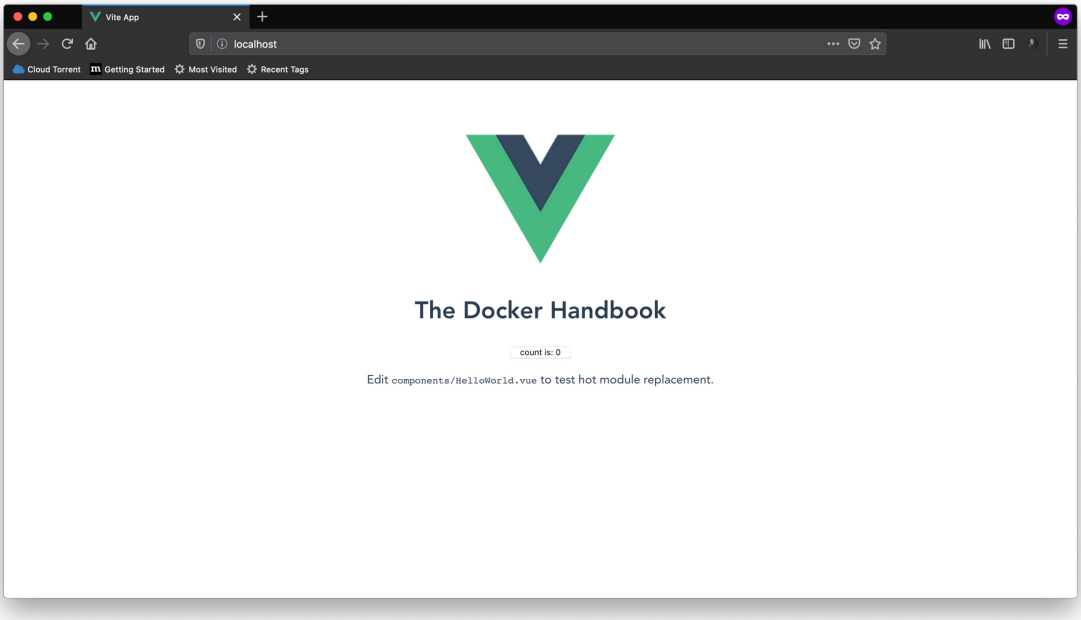
Then on the second stage we're using `nginx` as the base image. Copying all files from `/usr/app/dist` directory built during stage one to `usr/share/nginx/html` directory in the second stage. The `--from` option in the `COPY` instruction allows us to copy files between stages.

To build the image execute following command:

```
docker build .
```

We're using a file named `Dockerfile` this time so declaring the filename explicitly is unnecessary. Once the build process is finished, use the image id to run a new container. Nginx runs on port 80 by default so don't forget to map that

Once you've successfully started the container visit `http://127.0.0.1:80` and you should see the counter application running. Replace the 80 if you've used some other port from the host system.



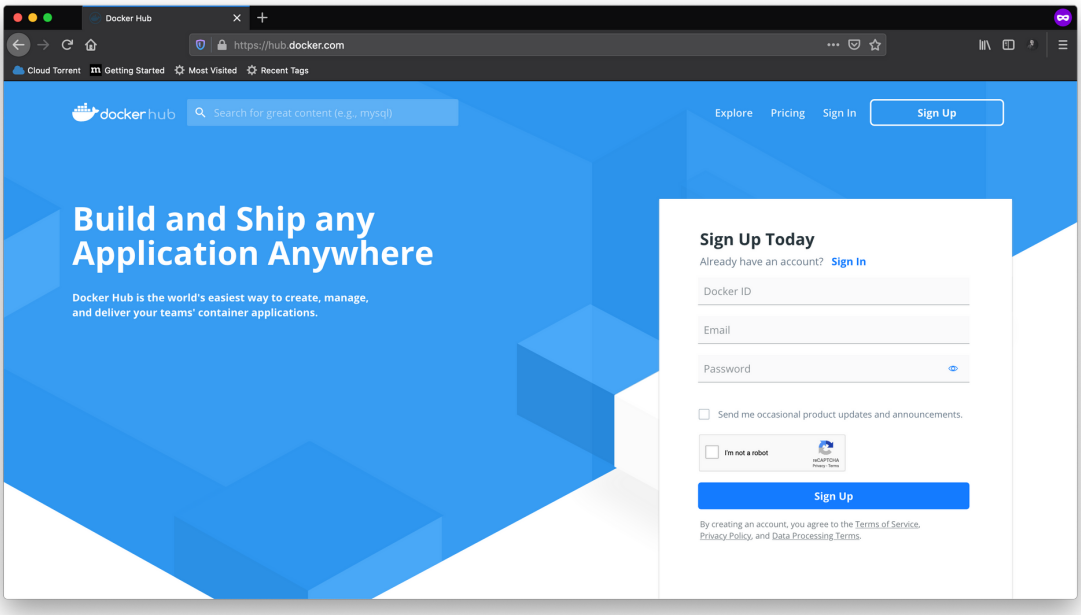
Production build of the `docker-counter` application running

The output image from this multi-staged build process is an Nginx based image containing just the built files and no extra data. It's optimized and lightweight in size as a result.

Uploading Built Images to Docker Hub

You've already built quite a lot of images. In this sub-section, you'll learn about tagging and uploading images to Docker Hub. Go ahead and sign up for a free account Docker Hub.

Donate



Sign up at Docker Hub

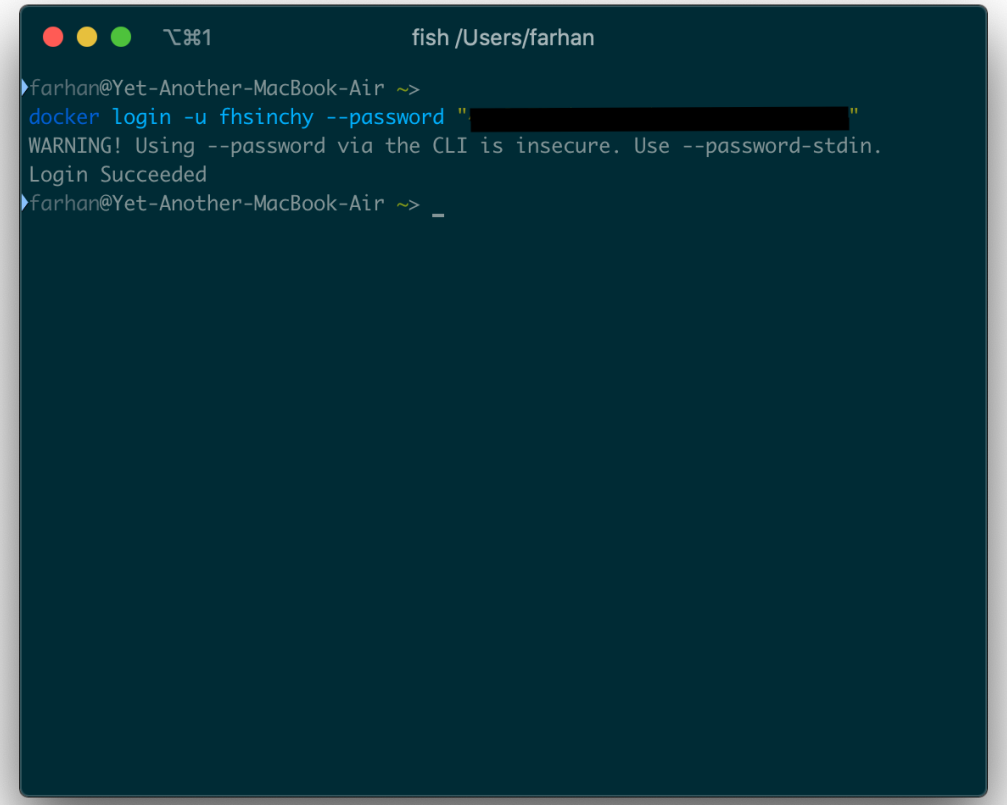
Once you've created the account, you log in using the Docker menu.

Sign in using the Docker menu

Or you can log in using a command from the terminal. Generic syntax for the command is as follows:

```
docker login -u <your docker id>  --password <your docker password>
```

If the login succeeds, you should see something like `Login Succeeded` on your terminal.



Sign in using the `docker login` command

Now you're ready to upload images. In order to upload images, you first have to tag them. If you've cloned the project code repository, open up a terminal window inside the `vite-counter` project folder.

command. The generic syntax for this option is as follows:

```
docker build -t <tag> <context of the build>
```

The general convention of tags is as follows:

```
<your docker id>/<image name>:<image version>
```

My Docker id is `fhsinchy`, so if I want to name the image `vite-counter` then the command should be as follows:

```
docker build -t fhsinchy/vite-controller:1.0 .
```

If you do not define the version after the colon, `latest` will be used automatically. If everything goes right, you should see something like `Successfully tagged fhsinchy/vite-controller:1.0` in your terminal. I am not defining the version in my case.

To upload this image to the hub you can use the `push` command. Generic syntax for the command is as follows:

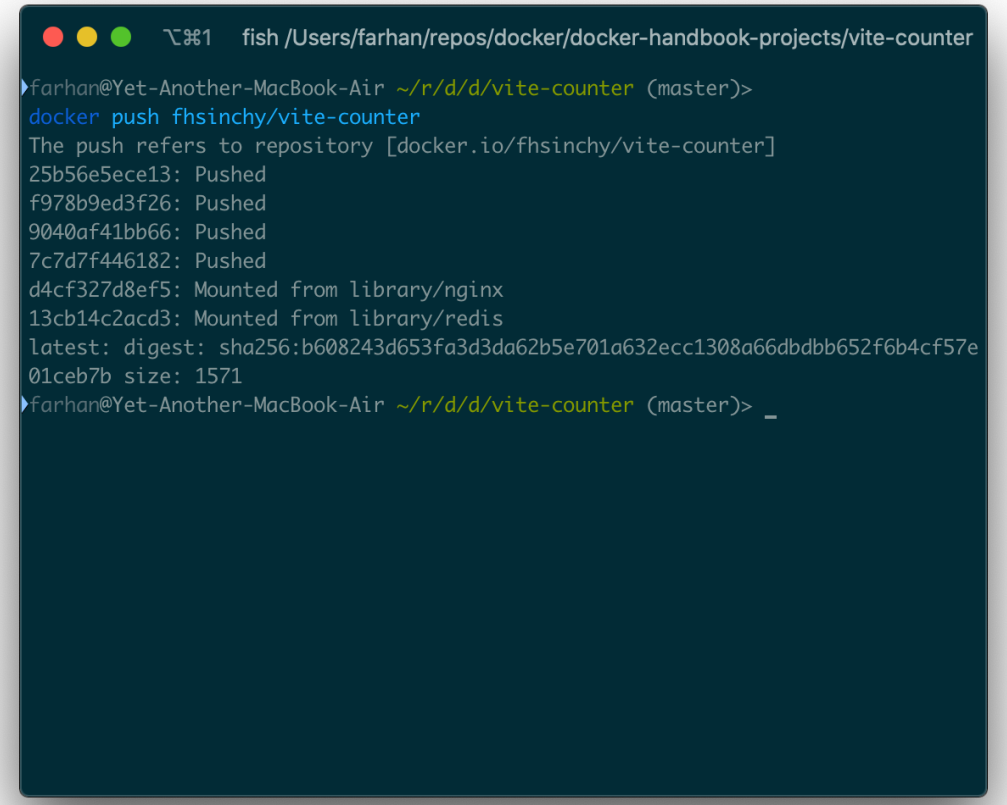
```
docker push <your docker id>/<image tag with version>
```

To upload the `fhsinchy/vite-counter` image the command should be

as follows:

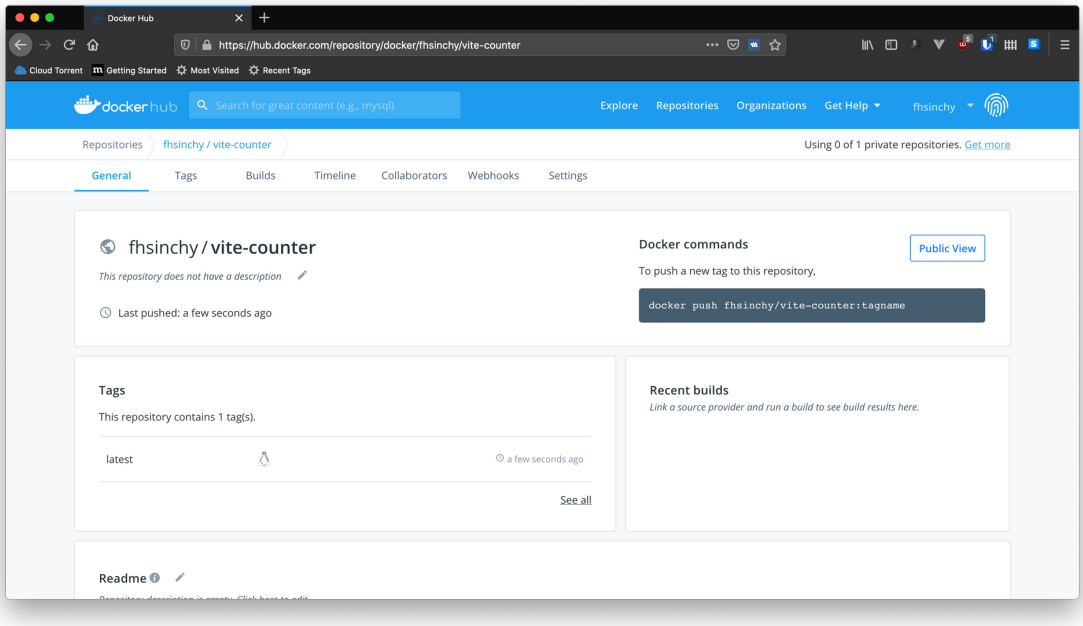
```
docker push fhsinchy/vite-counter
```

You should see some text like the following after the push is complete:



The image has been successfully pushed to Docker Hub

Anyone can view the image on the hub now.



Viewing the image on Docker Hub

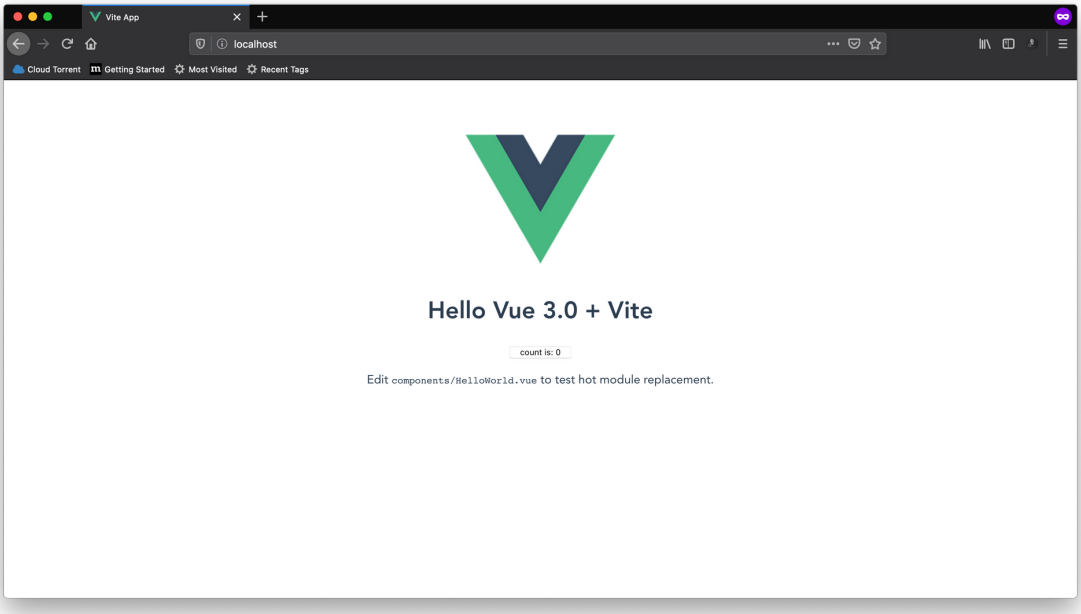
Generic syntax for running a container from this image is as follows:

```
docker run <your docker id>/<image tag with version>
```

To run the `vite-counter` application using this uploaded image, you can execute the following command:

```
docker run -p 80:80 fhsinchy/vite-counter
```

And you should see the `vite-counter` application running just like before.



The `vite-counter` application running using the uploaded image

You can containerize any application and distribute them through Docker Hub or any other registry, making them much easier to run or deploy.

Working with Multi-container Applications using Docker

Compose

So far we've only worked with applications that are comprised of only one container. Now assume an application with multiple containers. Maybe an API that requires a database service to work properly, or maybe a full-stack application where you have to work with an back-end API and a front-end application together.

In this section, you'll learn about working with such applications using a tool called Docker Compose.

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to 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.

Compose Basics

If you've cloned the project code repository, then go inside the `notes-api` directory. This is a simple CRUD API where you can create, read, update, and delete notes. The application uses PostgreSQL as its database system.

The project already comes with a `Dockerfile.dev` file. Content of the file is as follows:

```
FROM node:lts

WORKDIR /usr/app
```

```
COPY ./package.json .
RUN npm install

COPY . .

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

Just like the ones we've written in the previous section. We're copying the `package.json` file, installing the dependencies, copying the project files and starting the development server by executing `npm run dev`.

Using Compose is basically a three-step process:

1. Define your app's environment with a `Dockerfile` so it can be reproduced anywhere.
2. Define the services that make up your app in `docker-compose.yml` so they can be run together in an isolated environment.
3. Run `docker-compose up` and Compose starts and runs your entire app.

Services are basically containers with some additional stuff. Before we start writing your first YML file together, let's list out the services needed to run this application. There are only two:

1. `api` - an Express application container run using the `Dockerfile.dev` file in the project root.
2. `db` - a PostgreSQL instance, run using the official postgres image.

define your first service together. You can use `.yml` or `.yaml` extension. Both work just fine. We'll write the code first and then I'll break down the code line-by-line. Code for the `db` service is as follows:

```
version: "3.8"

services:
    db:
        image: postgres:12
        volumes:
            - ./docker-entrypoint-initdb.d:/docker-entrypoint-initdb.d
        environment:
            POSTGRES_PASSWORD: 63eaQB9wtLqmNBpg
            POSTGRES_DB: notesdb
```

Every valid `docker-compose.yml` 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 YML 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` is a service inside the services block.

The `db` block defines a new service in the application and holds necessary information to start the container. Every service requires either a pre-built image or a Dockerfile to run a container. For the `db` service we're using the official PostgreSQL image.

The `docker-entrypoint-initdb.d` directory in the project root

for keeping initialization scripts. There isn't a way to copy directories inside a `docker-compose.yml` file, that's why we have to use a volume.

The environment block holds environment variables. List of the valid environment variables can be found on the postgres image page on Docker Hub. The `POSTGRES_PASSWORD` variable sets the default password for the server and `POSTGRES_DB` creates a new database with the given name.

Now let's add the `api` service. Append following code to the file. Be very careful to match the indentation with the `db` service:

```
##
## make sure to align the indentation properly
##

api:
    build:
        context: .
        dockerfile: Dockerfile.dev
    volumes:
        - /usr/app/node_modules
        - ./:/usr/app
    ports:
        - 3000:3000
    environment:
        DB_CONNECTION: pg
        DB_HOST: db ## same as the database service name
        DB_PORT: 5432
        DB_USER: postgres
        DB_DATABASE: notesdb
        DB_PASSWORD: 63eaQB9wtLqmNBpg
```

We don't have a pre-built image for the `api` service, but we have a `Dockerfile.dev` file. The `build` block defines the build's `context` and

the `filename` of the Dockerfile to use. If the file is named just `Docker file` then the `filename` is unnecessary.
Mapping of the volumes is identical to what you've seen in the previous section. One anonymous volume for the `node_modules` directory and one named volume for the project root.

Port mapping also works in the same way as the previous section. The syntax is `<host system port>:<container port>`. We're mapping the port 3000 from the container to port 3000 of the host system.

In the `environment` block, we're defining the information necessary to setup the database connection. The application uses Knex.js as an ORM which requires these information to connect to the database. `DB_PORT: 5432` and `DB_USER: postgres` is default for any PostgreSQL server. `DB_DATABASE: notesdb` and `DB_PASSWORD: 63eaQB9wtLqmNBpg` needs to match the values from the `db` service. `DB_CONNECTION: pg` indicates to the ORM that we're using PostgreSQL.

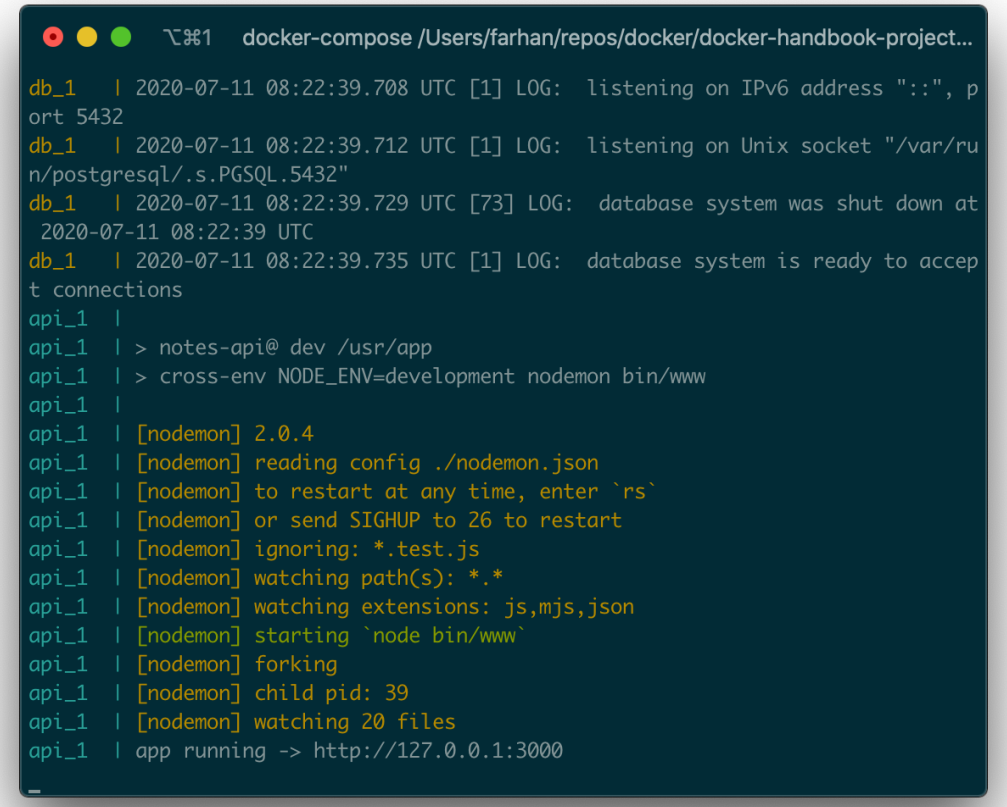
Any service defined in the `docker-compose.yml` file can be used as a host by using the service name. So the `api` service can actually connect to the `db` service by treating that as a host instead of a value like `127.0.0.1`. That's why we're setting the value of `DB_HOST` to `db`.

Now that the `docker-compose.yml` file is complete it's time for us to start the application. The Compose application can be accessed using a CLI tool called `dokcer-compose`. `docker-compose` CLI for Compose is what `docker` CLI is for Docker. To start the services, execute the following command:

```
docker compose up
```

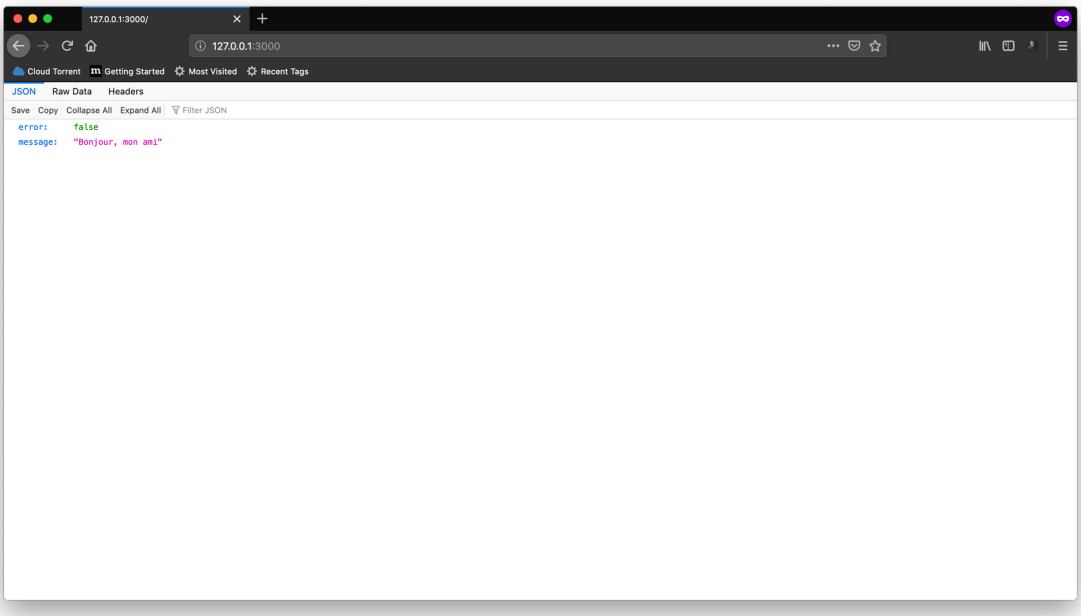
Executing the command will go through the `docker-compose.yml` file, create containers for each of the services and start them. Go ahead and execute the command. The startup process may take some time depending on the number of services.

Once done, you should see the logs coming in from all the services in your terminal window:



Output from `docker-compose up` command

The application should be running on `http://127.0.0.1:3000` address and upon visiting, you should a JSON response as follows:



http://127.0.0.1:3000

The API has full CRUD functionalities implemented. If you want to know about the end-points go look at the `/tests/e2e/api/routes/notes.test.js` file.

they don't exist. If you want to force a rebuild of the images, you can use the `--build` option with the `up` command. You can stop the services by closing the terminal window or by hitting the `ctrl + c` key combination.

Running Services in Detached Mode

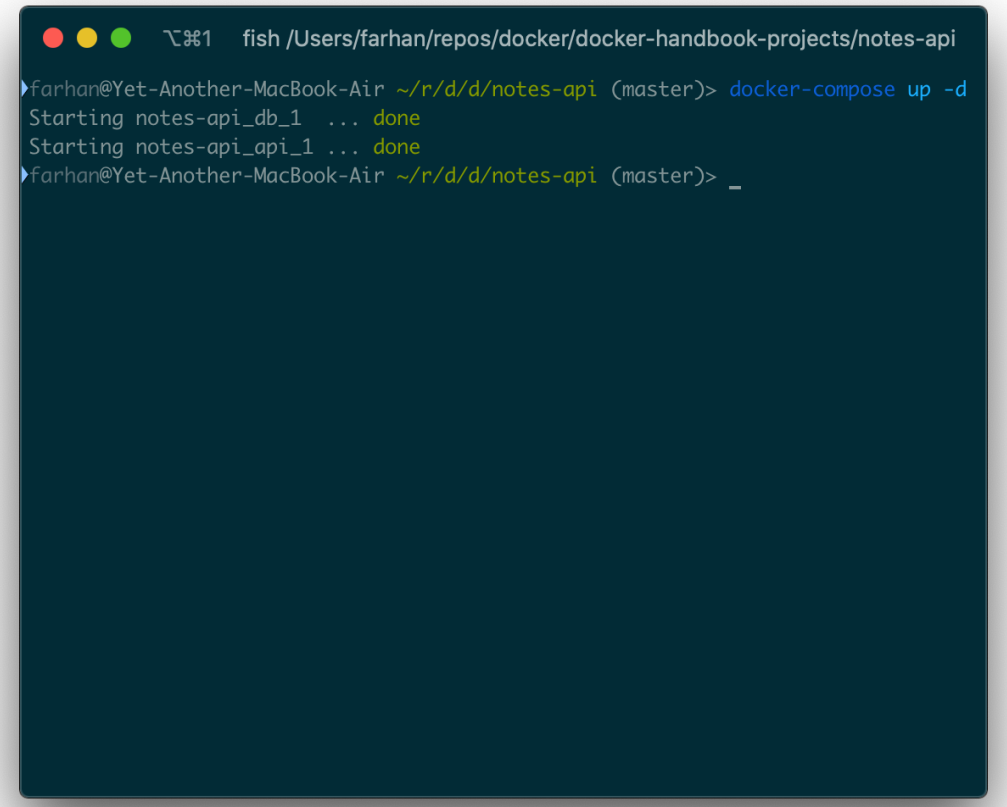
As I've already mentioned, services are containers and like any other container, services can be run in the background. To run services in detached mode you can use the the `-d` or `--detach` option with the `up` command.

To start the current application in detached mode, execute the following command:

```
docker-compose up -d
```

This time you shouldn't see the long wall of text that you saw in the previous sub-section.

Donate



Output from `docker-compose up -d` command

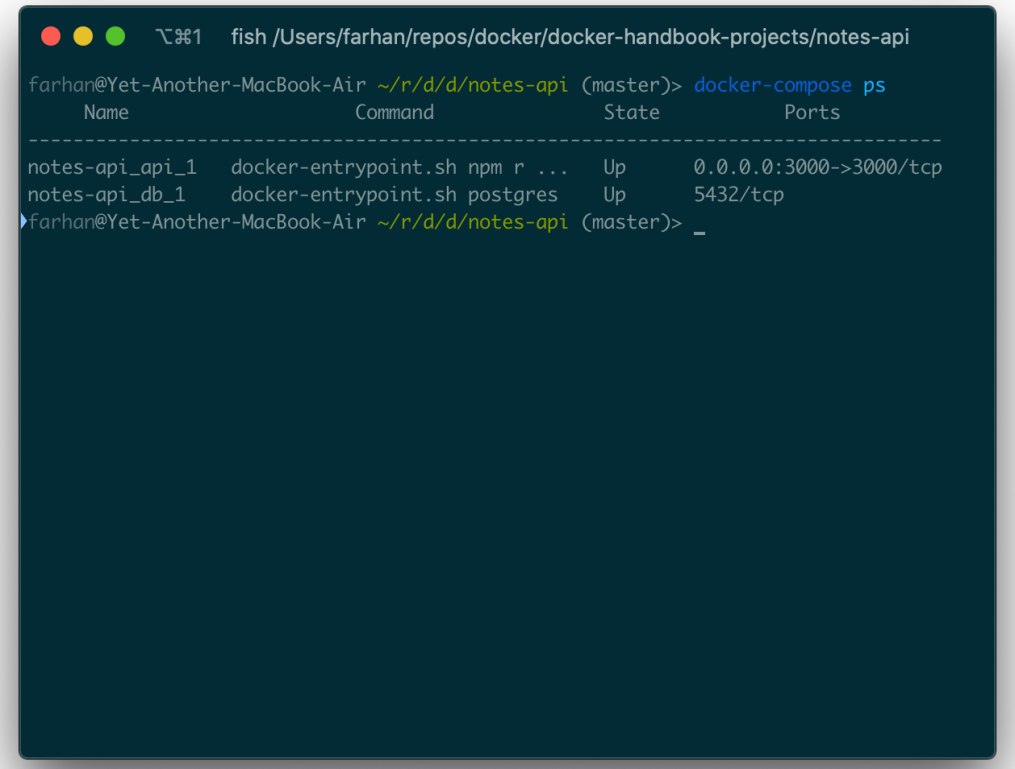
You should still be able to access the API at the `http://127.0.0.1:3000` address.

Listing Services

Just like the `docker ps` command, Compose has a `ps` command of its own. The main difference is the `docker-compose ps` command only lists containers part of a certain application. To list all the containers running as part of the `notes-api` application, run the following command in the project root:

```
docker-compose ps
```

Running the command inside the project directory is important. Otherwise it won't execute. Output from the command should be as follows:



Output from `docker-compose ps` command

The `ps` command for Compose shows services in any state by default. Usage of an option like `-a` or `--all` is unnecessary.

Executing Commands Inside a Running Service

Assume that our `notes-api` application is running and you want to access the `psql` CLI application inside the `db` service. There is a command called `exec` to do that. Generic syntax for the command is as follows:

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

Service names can be found in the `docker-compose.yml` file. The generic syntax for starting the `psql` CLI application is as follows:

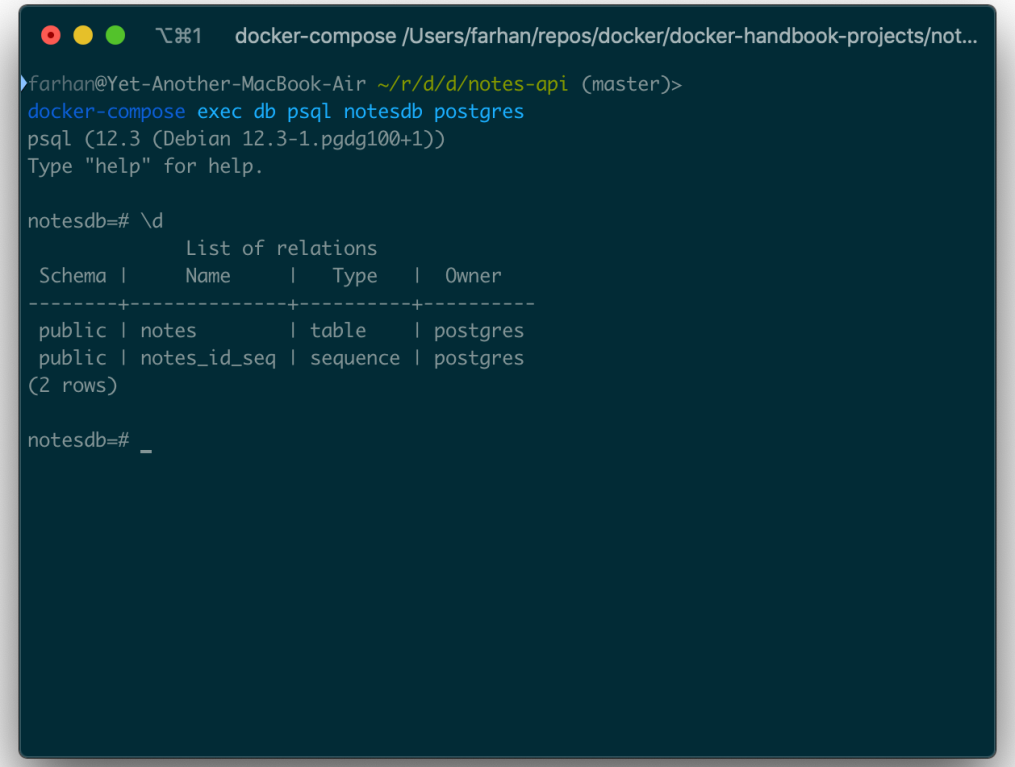
```
psql <database> <username>
```

Now to start the `psql` application inside the `db` service where the

command should be executed:

```
docker-compose exec db psql notesdb postgres
```

You should directly land on the `psql` application:



Output from `docker-compose exec db psql notesdb postgres` command

You can run any valid postgres command here. To exit out of the program write `\q` and hit enter.

Starting Shell Inside a Running Service

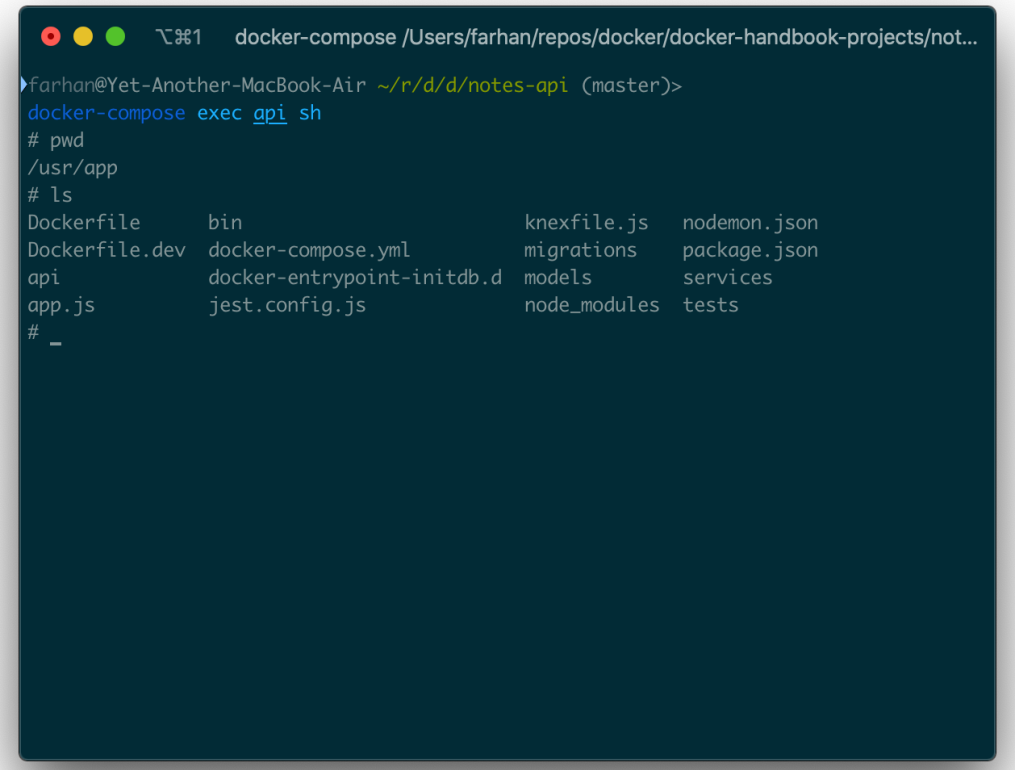
You can also start a shell inside a running container using the `exec` command. Generic syntax of the command should be as follows:

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

You can use `bash` in place of `sh` if the container comes with that. To start a shell inside the `api` service, the command should be as follows:

```
docker-compose exec api sh
```

This should land you directly on the shell inside the `api` service.

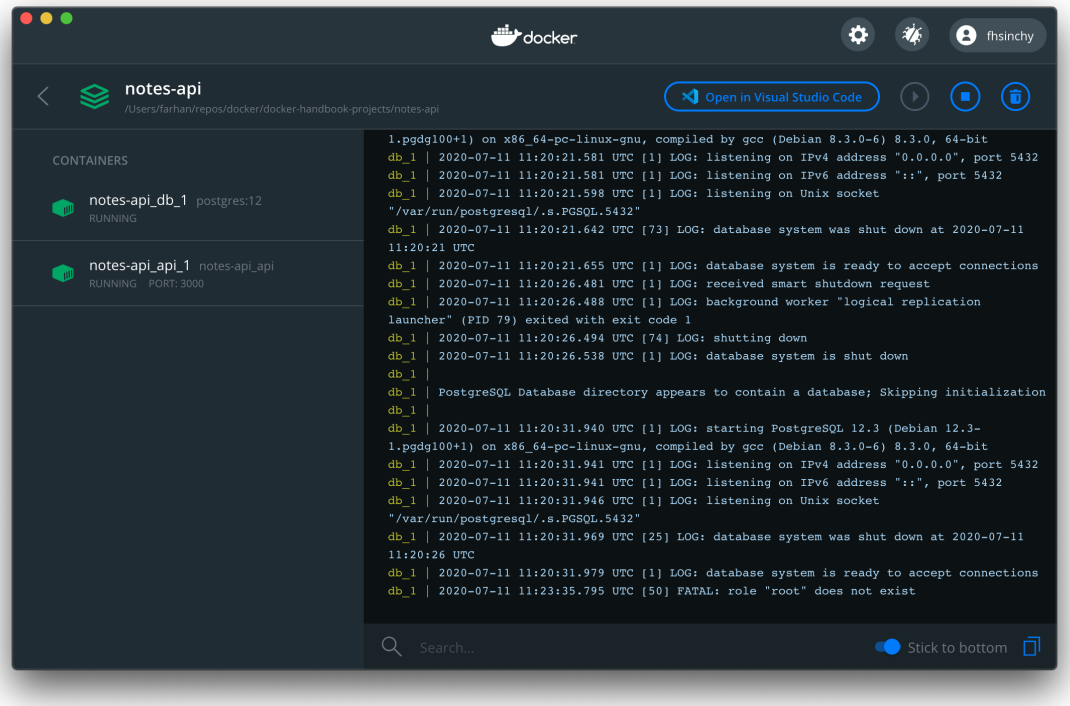


Output from the `docker-compose exec api sh` command

In there, you can execute any valid shell command. You can exit by executing the `exit` command.

Accessing Logs From a Running Service

If you want to view logs from a container, the dashboard can be really helpful.



Logs in the Docker Dashboard

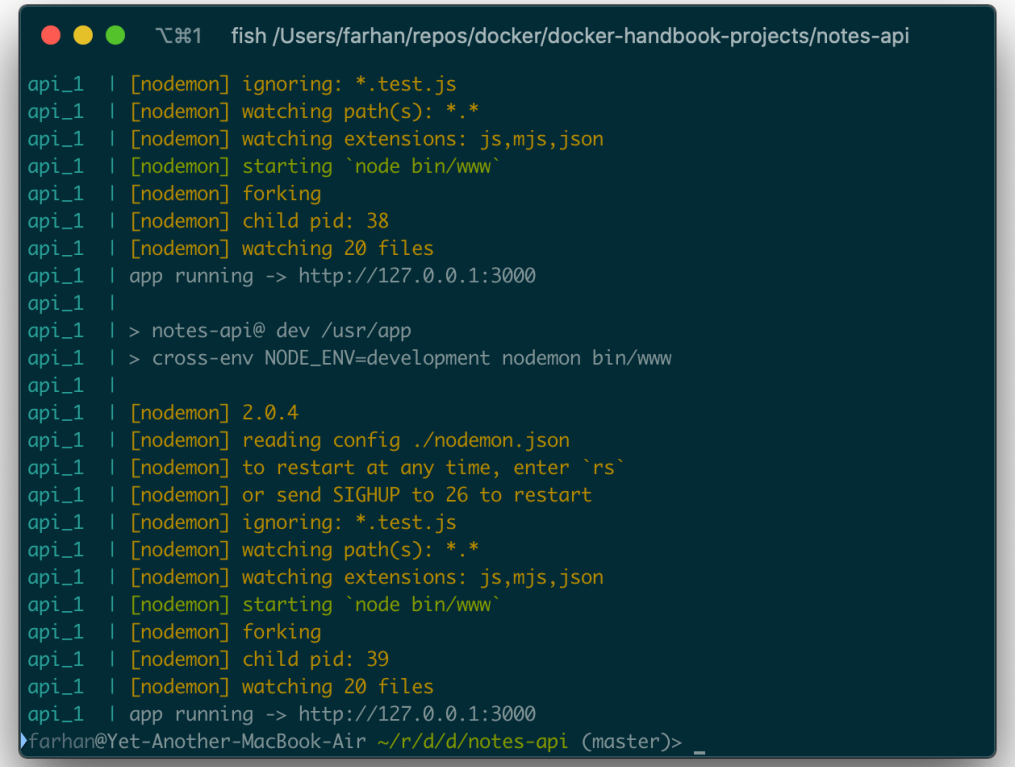
You can also use the `logs` command to retrieve logs from a running service. The generic syntax for the command is as follows:

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

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

```
docker-compose logs api
```

You should see a wall of text appear on your terminal window.



Output from `docker-compose logs api` 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` key combination or closing the window. The container will keep running even if you exit out of the log window.

Stopping Running Services

Services running in the foreground can be stopped by closing the terminal window or hitting the `ctrl + c` key combination. For stopping services in in the background, there are a number of commands available. I'll explain each of them one by one.

- `docker-compose stop` - attempts to stop the running services gracefully by sending a `SIGTERM` signal to them. If the services don't stop within a grace period, a `SIGKILL` signal is sent.
- `docker-compose kill` - stops the running services immediately by sending a `SIGKILL` signal. A `SIGKILL` signal can not be ignored by a recipient.
- `docker-compose down` - attempts to stop the running services gracefully by sending a `SIGTERM` signal and removes the containers afterwards.

If you want to keep the containers for the services you can use the `stop` command. If you want to removes the containers as well use the `d`

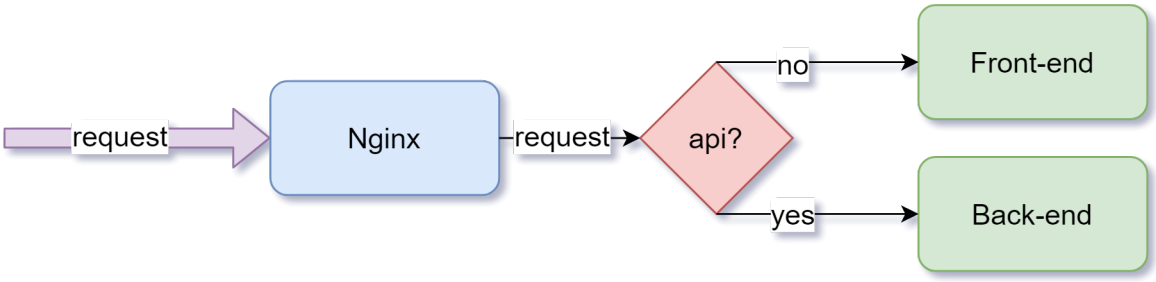
`own` command.

Composing a Full-stack Application

In this sub-section, we'll be adding a front-end application to our notes API and turn it into a complete 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 services and the corresponding Dockerfile.

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



Inner workings of the full-stack notes application

Instead of accepting requests directly like we previously did, in this application, all the requests will be first received by a Nginx server. Nginx will then see if the requested end-point has `/api` in it. If yes, Nginx will route the request to the back-end or if not, Nginx will route the request to the front-end.

application it doesn't run inside a container. It runs on the browser, served from a container. As a result, Compose networking doesn't work as 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 scope of this article. But if you want to have a look at it, go ahead and checkout the `/nginx/default.conf` file. Code for the `/nginx/Dockerfile.dev` for the is as follows:

```
FROM nginx:stable

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

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

Let's start writing the `docker-compose.yml` file by defining the services you're already familiar with. The `db` and `api` service. Create the `docker-compose.yml` file in the project root and put following code in there:

```
version: "3.8"

services:
    db:
        image: postgres:12
        volumes:
```

```
    environment:
        POSTGRES_PASSWORD: 63eaQB9wtLqmNBpg
        POSTGRES_DB: notesdb
api:
    build:
        context: ./api
        dockerfile: Dockerfile.dev
    volumes:
        - /usr/app/node_modules
        - ./api:/usr/app
    environment:
        DB_CONNECTION: pg
        DB_HOST: db ## same as the database service name
        DB_PORT: 5432
        DB_USER: postgres
        DB_DATABASE: notesdb
        DB_PASSWORD: 63eaQB9wtLqmNBpg
```

As you can see, these two services are almost identical to the previous sub-section, the only difference is the `context` of the `api` service. That's because codes for that application now resides inside a dedicated directory named `api`. Also, there is no port mapping as we don't want to expose the service directly.

The next service we're going to define is the `client` service. Append following bit of code to the compose file:

```
##
## make sure to align the indentation properly
##

client:
    build:
        context: ./client
        dockerfile: Dockerfile.dev
    volumes:
```

```
    environment:
        VUE_APP_API_URL: /api
```

We're naming the service `client`. Inside the `build` block, we're setting the `/client` directory as the `context` and giving it the Dockerfile name.

Mapping of the volumes is identical to what you've seen in the previous section. One anonymous volume for the `node_modules` directory and one named volume for the project root.

Value of the `VUE_APP_API_URL` variable inside the `environtment` will be appended to each request that goes from the `client` to the `api` service. This way, Nginx will be able to differentiate between different requests and will be able to re-route them properly.

Just like the `api` service, there is no port mapping here, because we don't want to expose this service either.

Last service in the application is the `nginx` service. To define that, append following code to the compose file:

```
##
## make sure to align the indentation properly
##

nginx:
    build:
        context: ./nginx
        dockerfile: Dockerfile.dev
    ports:
        - 80:80
```

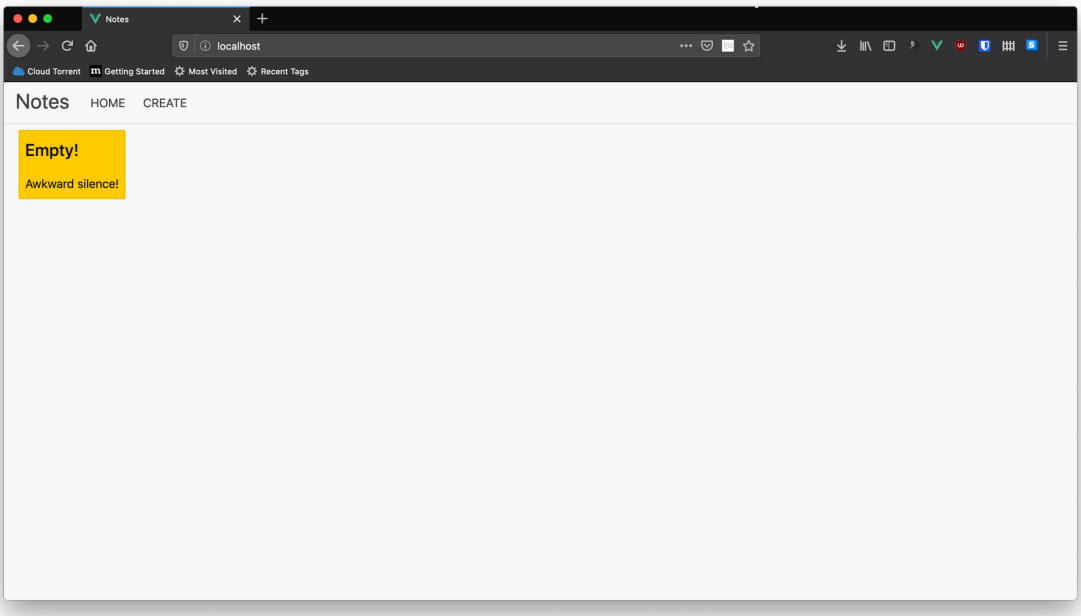
Content of the `Dockerfile.dev` has already been talked about. We're naming the service `nginx`. Inside the `build` block, we're setting the `/nginx` directory as the `context` and giving it the Dockerfile name.

As I've already shown in the diagram, this `nginx` service is going to handle all the requests. So we have to expose it. Nginx runs on port 80 by default. So I'm mapping port 80 inside the container to port 80 of the host system.

We're done with the full `docker-compose.yml` file and now it's time to run the service. Start all the services by executing following command:

```
docker-compose up
```

Now visit `http://localhost:80` and voilà!



https://localhost:80/

Try adding and deleting notes to see if the application works properly or not. Multi-container applications can be a lot more complicated than this, but for this article, this is enough.

Conclusion

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

To stay updated with my upcoming works, follow me @frhnhsin ✌

Farhan Hasin Chowdhury

Programmer, Voracious Reader and Video Game Enthusiast

Donate

videos, articles, and interactive coding lessons - all freely available to the public. We also have thousands of freeCodeCamp study groups around the world.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can make a tax-deductible donation here.

Trending Guides

JavaScript Closure	JavaScript Promise
CSS Box Shadow	What is GitHub?
Python List Append	Python Sort List
JavaScript Array Sort	Comments in JSON
Symlink in Linux	What is Kanban?
Linux Grep Command	Python Write to File
What is DNS?	CSS Media Queries
Primary Key SQL	HTML Entities
SQL Update Statement	Excel VBA
Screenshot on PC	LOOKUP in Excel
What is a Proxy Server?	Arrow Function JavaScript
Cat Command in Linux	Remove Duplicates in Excel
CSS Background Image	dllhost.exe COM Surrogate
HTML Background Color	Boolean Algebra Truth Table
CSS Comment Example	Video Chat for Android

Our Nonprofit