CS3219-Docker-Guide

CS3219 SE Toolbox - Containerization

The CS3219 SE Toolbox is a collection of guides and resources to help you get started with the various tools and technologies used CS3219 - Software Engineering Principles and Patterns.

The guide and resources below focuses on containerization using Docker.

Objectives

As you work your way through the Docker Guide you will achieve the following:

1. Setup Docker and learn to interact with the container
    - Setup Docker and test the installation by running a simple container. You can find the instructions in Sections 1.2 and 1.3 of the guide. You may choose to run a container of a lightweight image like BusyBox or Alpine.
    - Run an echo command on the image from outside the container
    - Note how to interact with the container.

Observe the outcomes of each task and note down your observations. You may find it useful when using Docker later for assignment or project.

1. Learn how to run pre-built images, and publish to ports to view web applications
    - Follow the instructions in Section 1.4 of the guide.
    - Try publishing to different ports.
    - Additionally you can try this : Pull and run any Docker image you find interesting in Docker Hub. Explain what it does, and the steps taken to run it.

Observe the outcomes of each task and note down your observations. You may find it useful when using Docker later for assignment or project.

1. Learn how to write a Dockerfile to Dockerize your application. And then build and run your custom image.
    - Write a simple ReactJS web application with an ExpressJS server that displays “Hello <YOUR FULL NAME>, this is running in a Docker container.”
    - Dockerise the above application by writing a Dockerfile. Explain the significance behind the commands used in the Dockerfile.

   - Build and run the image using the Dockerfile you created previously. Map the container port to port 3000 on the host if available.
   - Play around and see if you can run the app on 2 different containers at 2 different ports.

2. Using Docker compose tool

   - Multi-container environments can easily be run using the Docker compose tool.
   - Docker's awesome-compose samples provide a starting point on how to integrate different frameworks and technologies using Docker compose.
   - Check out the samples in awesome-compose repository and try one/some of them out. They are ready to run with docker compose up.
   - Some questions you can try to answer as you gain experience with `awesome-compose` :
     > - How does docker compose simplify the workflow?
     > - What does the set of containers you tried help you to achieve? (I.e., what can you do with those set of containers).
     > - Did you try to compose something different? What difficulties did you face?

Getting Started with Docker

1.1. Introduction

Docker Docs defines Docker as follows:

> *Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.*

Docker allows developers to package their applications and dependencies into a lightweight container by providing a layer of abstraction of OS-level virtualisation on Linux.

Docker containers are relatively well isolated from eachother and the host machine. So, developers can run their applications on any machine that has Docker installed, regardless of the underlying OS.

Unlike virtual machines, containers do not have high overhead and therefore able to efficiently use the system resources.

▶ 🔍 **Click here to read about the common concepts and terminologies used in Docker.**

1.2. Installation and Setup

Install Docker for your respective OS here.

Follow the instructions/install updates (if any). If successful, you will have installed Docker Desktop on your device.

Docker Desktop provides a GUI to help manage containers, applications, images, etc. It can be used as is or as a complementary tool to the Docker CLI.

> 📝 **Note:** Some parts of this manual will refer to `<your username>` in some of the Docker CLI commands. Replace those with your Docker Hub username. If you do not have a Docker Hub account, create one here. If you are using your username for the first time, you may have to login to Docker Hub using the `docker login` command. Sometimes the login command may not work. In that case, you can login to Docker Hub using the Docker Desktop app or restart the terminal and try again.

Test your installation by running the following command in your terminal:

```
docker run hello-world
```

If successful, you should see something similar to the following output:

```
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:
  https://docs.docker.com/get-started/
```

> ⚠️ *Warning* ⚠️ You will face errors if you don't start the Docker daemon before running the command. If you are using Docker Desktop, you can start the daemon by clicking on the Docker icon in your taskbar/open the Docker Desktop app. If you are using Docker CLI, you can start the daemon by running `dockerd` in your terminal (This option better applies to Linux users).

> 📝 **Note:** Running your terminal and Docker at different privilege levels may cause issues. For example, running your terminal as an administrator and docker as a normal user may cause issues. If you face any issues, try running both at the same privilege level.

1.3. Running Your First Container with `docker run`

Now that we have Docker installed and have a basic understanding of Docker, lets run our first container.

> ⏰ **Reminder**: Ensure that your Docker daemon is running before you proceed. You may do this by opening the Docker Desktop app or *running `dockerd` in your terminal (this option is for Linux users)*.

We'll run an Alpine Linux container (since it is a lightweight distribution of linux). You can try out the subsequent steps with other images as well - like BusyBox, Ubuntu, etc.

Enter the command:

```
docker pull alpine
```

This will pull the latest Alpine image from Docker Hub.

If successful, you should see something similar to the following output:

```
Using default tag: latest
latest: Pulling from library/alpine
Digest: sha256:82d1e9d7ed48a7523bdebc18cf6290bdb97b82302a8a9c27d4fe885949ea94d1
Status: Image is up to date for alpine:latest
docker.io/library/alpine:latest

What's Next?
  View summary of image vulnerabilities and recommendations → docker scout quickview alpin
```

> ⚠️ *Warning* ⚠️ If you get a `permission denied` error, check you installation and setup. You may have to run the command as an administrator. If you are using linux you may have to

> prefix your command with `sudo` .

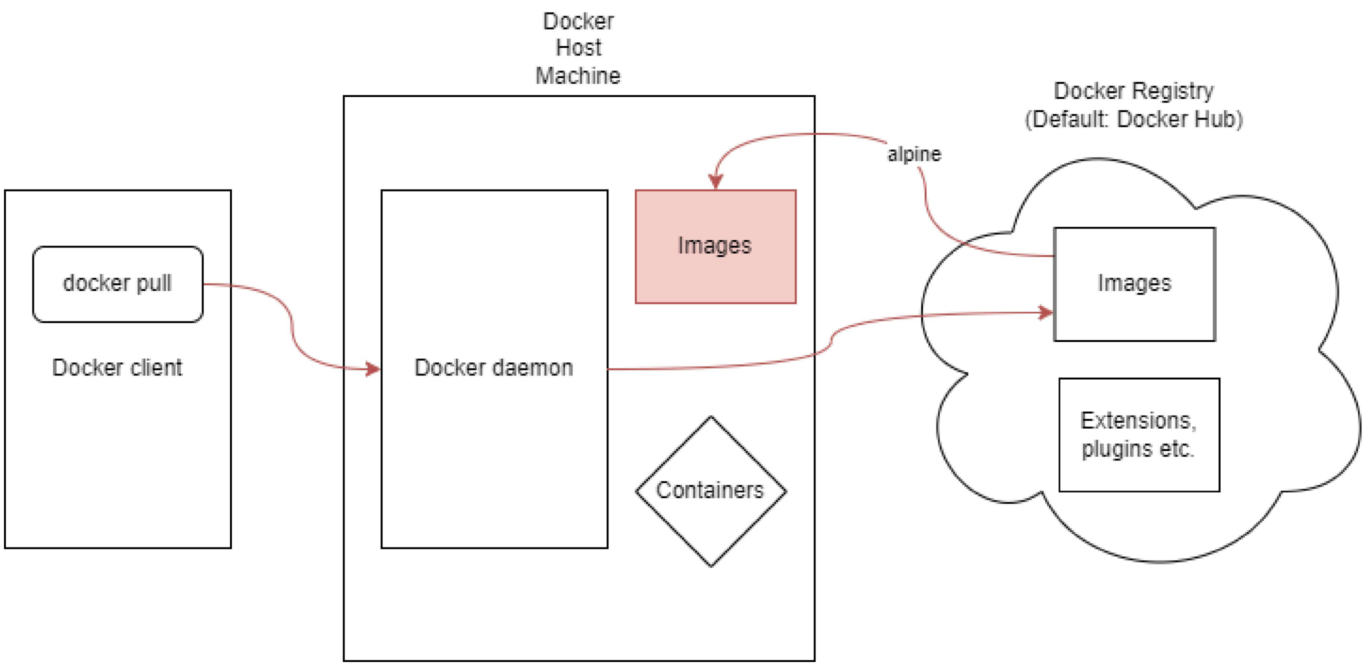


Figure 1.3.1: This an example of what happens when the `docker pull` command is executed to obtain the alpine image.

To check the images you have on your system, run the command:

```
docker images
```

You should see something akin to the following output:

```
REPOSITORY                      TAG       IMAGE ID       CREATED        SIZE
docker/welcome-to-docker        latest    912b66cfd46e   5 weeks ago    13.4MB
postgres                        <none>    696ffaadb338   6 weeks ago    237MB
alpine                          latest    c1aabb73d233   6 weeks ago    7.33MB
hello-world                     latest    9c7a54a9a43c   2 months ago   13.3kB
```

Now we will run a Docker container based on this image that we just downloaded. For this, we'll use the `docker run` command.

```
docker run alpine ls -l
```

You should see something similar to the following output:

```
total 56
drwxr-xr-x    2 root     root          4096 Jun 14 15:03 bin
drwxr-xr-x    5 root     root           340 Jul 28 08:55 dev
drwxr-xr-x    1 root     root          4096 Jul 28 08:55 etc
drwxr-xr-x    2 root     root          4096 Jun 14 15:03 home
....
```

Basically what we did was run the `ls -l` command on the Alpine image. This command lists the contents of the current directory.

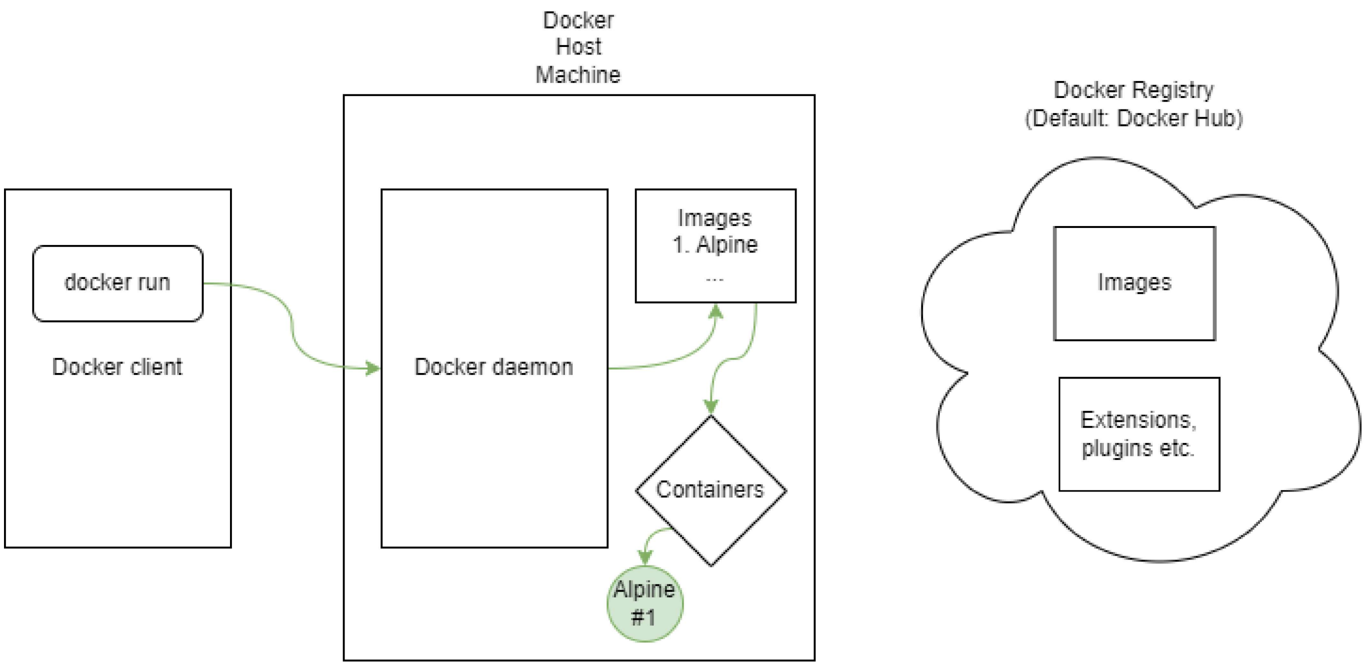


Figure 1.3.2: This an example of what happens when the `docker run` command is executed for an Alpine container.

Lets try some more commands in the container. Run the following:

```
docker run alpine echo "hello from alpine"
```

The output should be:

```
hello from alpine
```

Docker essentially ran the `echo` command in the alpine container and exited it. This is normal behaviour. To stay in the container and keep it running, we can use the `-it` flag. This flag allows us to interact with the container (interactive mode). With this, you can also use the Docker container as a development environment.

```
docker run -it alpine
```

You will enter the containers shell. You can now run commands in the container.

```
/ # ls
bin     dev     etc     home    lib     media   mnt     opt     proc    root    run     sbin    srv
/ # cd bin
/bin # cd ..
/ #
```

You can exit the container by running the `exit` command.

To see the containers you are **currently** running, use the `docker ps` command.

```
docker ps
```

Output:

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

As you can see, nothing is running right now. Try using the `-a` flag to see all containers that have been run on your system.

```
docker ps -a
```

Output:

```
CONTAINER ID   IMAGE         COMMAND                  CREATED          STATUS
6b9fc12296f2   alpine        "/bin/sh"                3 minutes ago    Exited (0) 2 minute
0a57257081b8   alpine        "/bin/sh"                5 minutes ago    Exited (0) 5 minute
de280dc604e1   alpine        "echo 'hello from al…"   12 minutes ago   Exited (0) 12 minut
c009ae1dded7   alpine        "ls -l"                  38 minutes ago   Exited (0) 38 minut
2c323cbabfb5   hello-world   "/hello"                 26 hours ago     Exited (0) 26 hours
```

Yay! You successfully ran your first container. 🎉

Now that you are equipped with the basics, lets get to the interesting part - deploying web applications with Docker.

1.4. Running a Sample Static Website with Docker

In this section, we will pull the `dockersamples/static-site` image from Docker Hub and run a container based on that image. It is a pre-built image that will run a simple HTML static website in an nginx container.

This exercise will help you understand how to run pre-existing images, and how to publish ports so that you can view your web application.

Run the following command:

```
docker run -d dockersamples/static-site
```

> 📝 **Note:** `-d` (or `--detach`) flag runs the container in detached mode, that is, in the background. It runs the container normally, but will bring you to the terminal prompt after the container is started. This is useful when you want to run a container in the background and continue using the terminal.

The output should be a long hex value. This is the full container ID.

```
c3557c35fca64bae767ec7e1b27415425b128dd2cd6af8682fe8a232cfd178ac
```

If the image is not already on your device, the Docker daemon will fetch it from the registry. Then it immediately starts the container and runs it in the background.

How do you see the website that is running? The thing is, we havent specified a port for the Docker engine to publish to. We will have to re-run docker with the `-P` flag to specify the port.

To stop the container, we will retrive the short container ID with the following command:

```
docker ps
```

Ideal output:

```
CONTAINER ID   IMAGE                         COMMAND                  CREATED         STATUS
c3557c35fca6   dockersamples/static-site     "/bin/sh -c 'cd /usr…"   6 minutes ago   Up 6 m
```

Use the container ID to refer to the container you want to stop. In this case, it is `c3557c35fca6`. Run the following command:

```
docker stop c3557c35fca6
```

Then remove it using the following command:

```
docker rm c3557c35fca6
```

Run the following command to run the container again, but specifying the port:

```
docker run --name static-site -e AUTHOR="Your Name" -d -P dockersamples/static-site
```

> 📝 **Note:** `--name` flag allows you to specify a name for the container. `-e` flag allows you to set environment variables. In this case, we set the AUTHOR variable to our name. This will be displayed on the website. `-P` flag publishes all exposed ports to random ports. This is useful when you don't know which port the application will use.

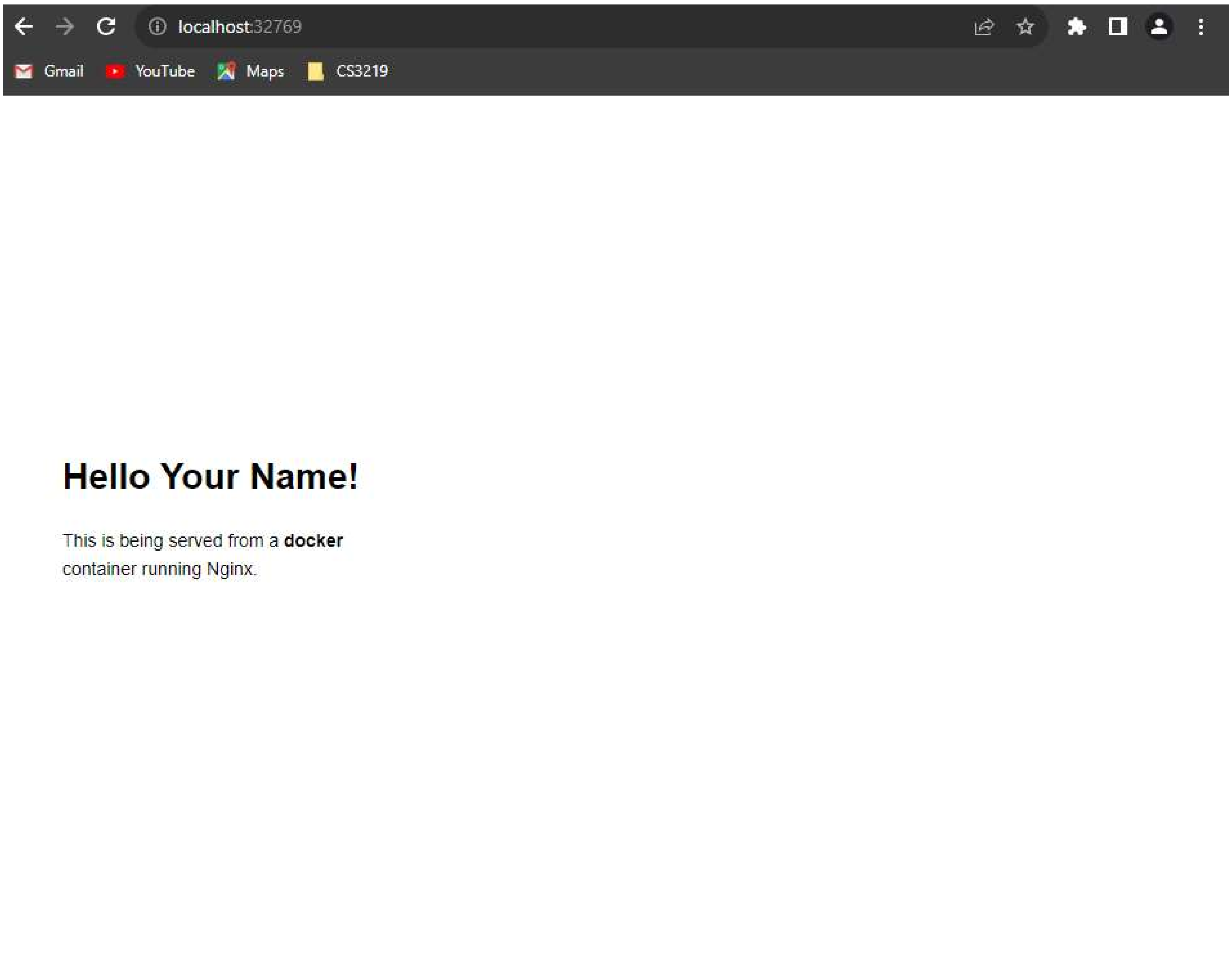
Now see the port using the following command:

```
docker port static-site
```

Output:

```
80/tcp -> 0.0.0.0:32769
443/tcp -> 0.0.0.0:32768
```

Since we are using Docker Desktop, open `http://localhost:[PORT FOR 80/tcp]/` in your browser. In this example it is `http:\\localhost:32769` . You should see the following page:



Hello Your Name!

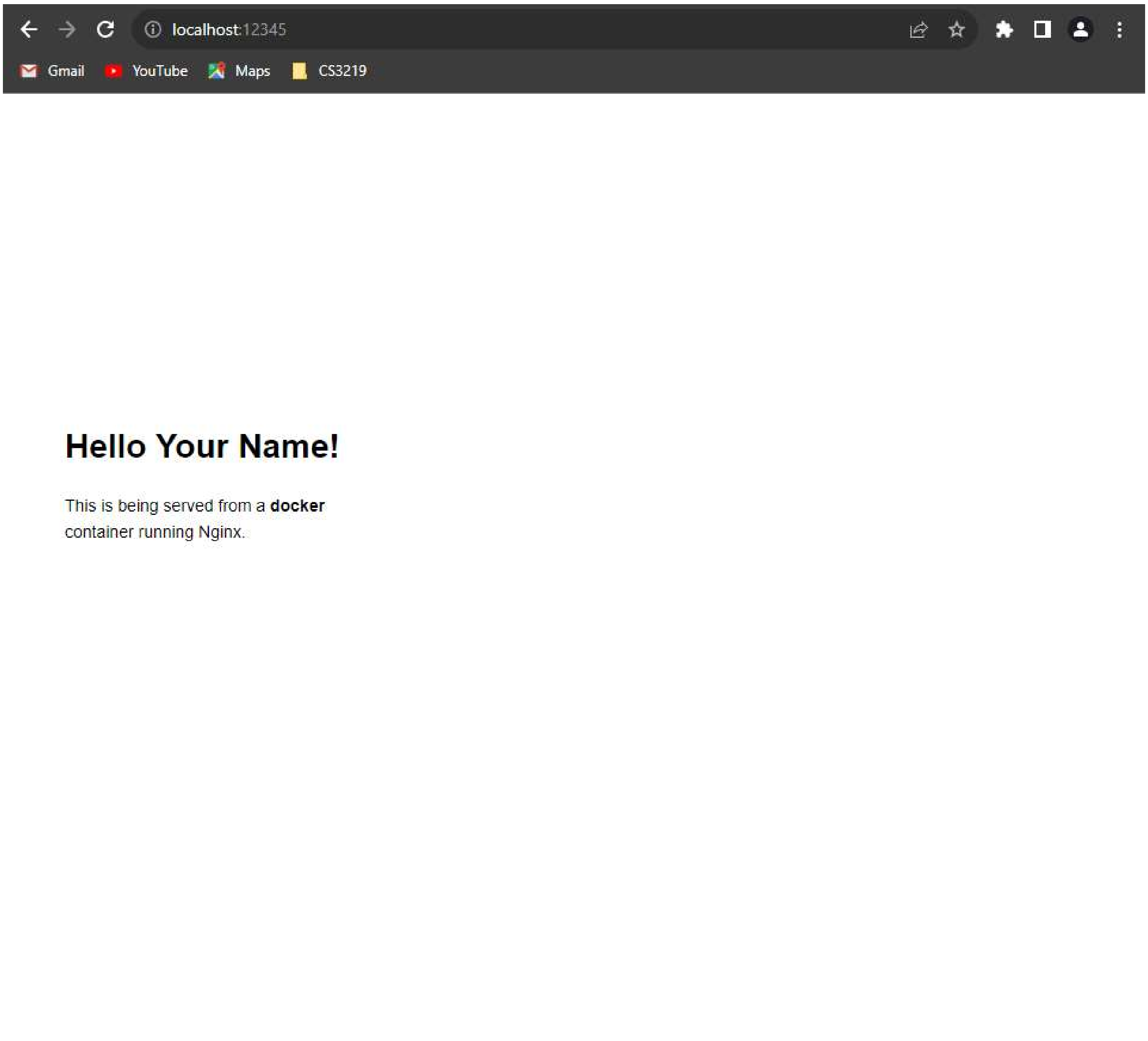
This is being served from a **docker** container running Nginx.

Figure 1.4.1: This is an example of the static website.

You can run another webserver at the same time. Previously, we used random ports. But we can also specify the port we want to use. Run the following command:

```
docker run --name static-site-2 -e AUTHOR="Your Name" -d -p 12345:80 dockersamples/static-
```

Open `http://localhost:12345` in your browser.



Hello Your Name!

This is being served from a **docker** container running Nginx.

Figure 1.4.2: This is an example of the static website running in parallel on port 12345.

> ⏰ **Reminder**: Stop and remove the containers after you are done with them. Run `docker ps` to make sure they are gone.

1.5. Building and Running Your Own Docker Image

In the previous exercise, we ran a static website using an existing image from Docker Hub. In this section, we will build our own image.

▶ 🔍 **Click here to find out more about Images in Docker.**
The goal is to create a Docker image that sandboxes a simple React + ExpressJS application.

First we will put together a simple React App with an ExpressJS server, then dockerize it by writing a Dockerfile. Finally, we will build and run the image.

1.5.1. Creating a Simple React App

Prerequisites: Install NodeJS (with npm) and yarn if you haven't already. We suggest that you use this node version for the purposes of this module => LTS v18.17.0, npm is v9.6.7 and yarn v1.22.19.

If you already have a React app with Express server you'd like to dockerize, you go the the next section.

Inside the folder you want to create the project, run the following commands:

```
mkdir test
cd test
npm init -y
npm install express --save
npm i -g create-react-app
```

This will create a new folder called `test` and initialise a new node project. Then it will install express and create-react-app globally.

You should have `node_modules` , `package-lock.json` and `package.json` in your folder.

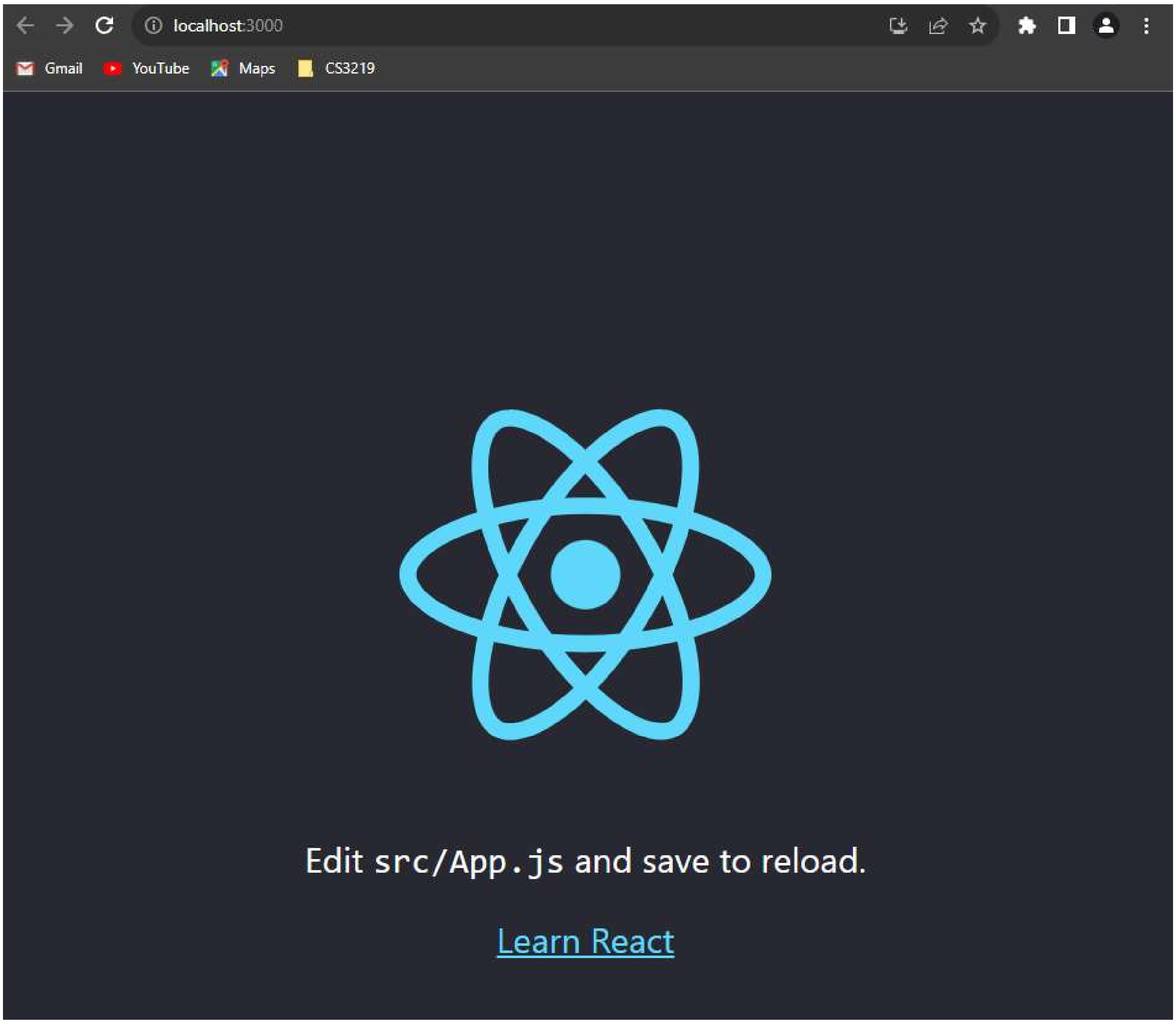
> ⚠️ ***Warning*** ⚠️: If you are using an old version of npm, you may not have `package-lock.json` . This is a reminder to use a newer version of npm.

Now we will create a React app. Run the following commands:

```
create-react-app testapp
cd testapp
yarn start
```

> 📝 **Note:**: You can also use `npm start` instead of `yarn start` . But make sure you configure the predefined command specified in the "start" property of a package's "scripts" object in `package.json` i.e. set it to "node index.js" for the purposes of this application.

`testapp` is the name of the React app. You can name it whatever you want. `yarn start` will start the development server. You should see the following page open in your browser:



Edit `src/App.js` and save to reload.

Learn React

Figure 1.5.1.1: React sample application running on localhost:3000

Use control + c to stop the development server. Build the project using `yarn build` . This will create a `build` folder in your project directory. Find it in `testapp/build` .

Go back to the root folder, `test` . Create a file called `index.js` . We will develop the production server for the app here.

The development server included in create-react-app is not suitable to serve a react app over the internet. It is not optimised for performance and security. We will use ExpressJS to serve the production build of the React app.

Inside `index.js` , add the following code:

```
const express = require('express');
const app = express();

// Serve up production assets
app.use(express.static('testapp/build'));
```

```
// Serve up the index.html if the route is not recognized
const path = require('path');
app.get('*', (req, res) => {
    res.sendFile(path.resolve(__dirname, 'testapp', 'build', 'index.html'));
});

// If not in production, use port 8080 or the environment port
const PORT = process.env.PORT || 8080;
console.log(`Listening on port ${PORT}`);
app.listen(PORT);
```

▶ 🔍 **Click here to find out more about what the code above does**

> ⏰ **Reminder**: `testapp` is the name of the React app. If you named it something else, change the code accordingly.

Run `node index.js` to start the server. Go to http://localhost:8080/ to see the React app running.

Yay! You have successfully created a simple React app with an ExpressJS server. 🎉

1.5.2. Writing a Dockerfile

Now that we have a simple React app with an ExpressJS server, we will write a Dockerfile to containerise it.

A Dockerfile is a simple text document that provides users with a set of clear instructions for building an image using the command line.

Docker can build images automatically by reading the instructions from a Dockerfile.

Create a file called `Dockerfile` in the project folder. With reference to the previous section, that would be in the `test` folder.

In the Dockerfile we will specify the base image, copy the required files, install dependencies, and start the server. Follow the instructions below to write the Dockerfile.

1. Specify the base image. We will use the `node:18` image. This is the latest long term support (LTS) version of node.

   ```
   FROM node:18
   ```

2. Specify the working directory. This is where the files will be copied to inside the image.

   ```
   WORKDIR /usr/src/app
   ```

3. The image comes with NodeJS and npm pre-installed. We will copy the `package.json` and `package-lock.json` files to the working directory. Then we will install the remaining

dependencies.

```
COPY package*.json ./
```

4. Install the dependencies.

```
RUN npm install
```

5. Add this line, but keep it commented. Uncomment it if you are building code for production.

```
# RUN npm ci --omit=dev
```

6. Bundle the app's source code inside the Docker image.

```
COPY . .
```

7. Expose the port 8080 so it can be mapped by the Docker daemon.

```
EXPOSE 8080
```

8. Specify the command to run the app.

```
CMD ["node", "index.js"]
```

At the end, your Dockerfile will look something like this:

```
# Define the image you want to build from.
# In this case, we are using the latest LTS (long term support) version of Node.
FROM node:18

# Create app directory to hold application code inside the image.
WORKDIR /usr/src/app

# The image comes with Node.js and NPM already installed.
# We just need to install the rest of our dependencies.
# Copy package.json and package-lock.json to the app directory on the image.
COPY package*.json ./

# Install dependencies.
RUN npm install

# Uncomment the following line if you are building code for production.
# RUN npm ci --omit=dev

# Bundle the app's source code inside the Docker image.
COPY . .

# Expose port 8080 so it can be mapped by Docker daemon.
EXPOSE 8080
```

```
# Define the command to run your app using CMD which defines your runtime.
CMD [ "node", "index.js" ]
```

Create a `.dockerignore` file in the project folder. This file specifies the files and folders that should be ignored when copying files to the image. Add the following lines to the file:

```
node_modules
npm-debug.log
```

Great job! You have successfully written a Dockerfile. 🎉 Note that Dockerfiles are specific to the application they are building and the environment they are running in. So if you would like to dockerize a different kind of application, you will have to write a different Dockerfile.

1.5.3. Building and Running the Image

> ⏰ **Reminder**: Ensure that your Docker daemon is running before you proceed. You may do this by opening the Docker Desktop app or *running `dockerd` in your terminal (this option is for Linux users)*.

In the directory that contains your Dockerfile, run the following command to build the image:

```
docker build . -t <your username>/test-web-app
```

> 📝 **Note:** The `-t` flag tags the image. This is useful when you want to refer to the image later. You can name it whatever you want. In this case, it is named `test-web-app`. Also remember to replace `<your username>` with your Docker Hub username.

Once the build is complete, check if your image is now listed by docker.

▶ **❓ Do you remember which command you have to run to check the images on your machine?**

The result should contain your image. In this case, it is `<your username>/test-web-app`.

```
REPOSITORY                                TAG        IMAGE ID        CREATED          SIZE
<your username>/test-web-app               latest     5840e4960d7e    15 seconds ago    1.39GB
```

Run your image using the following command:

```
docker run -p 12345:8080 -d <your username>/test-web-app
```

▶ **❓ Can you recall what the -p and -d tags do?**

If you want to see the console logs printed in the container, run the following command:

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

You should see the following output:

```
Listening on port 8080
```

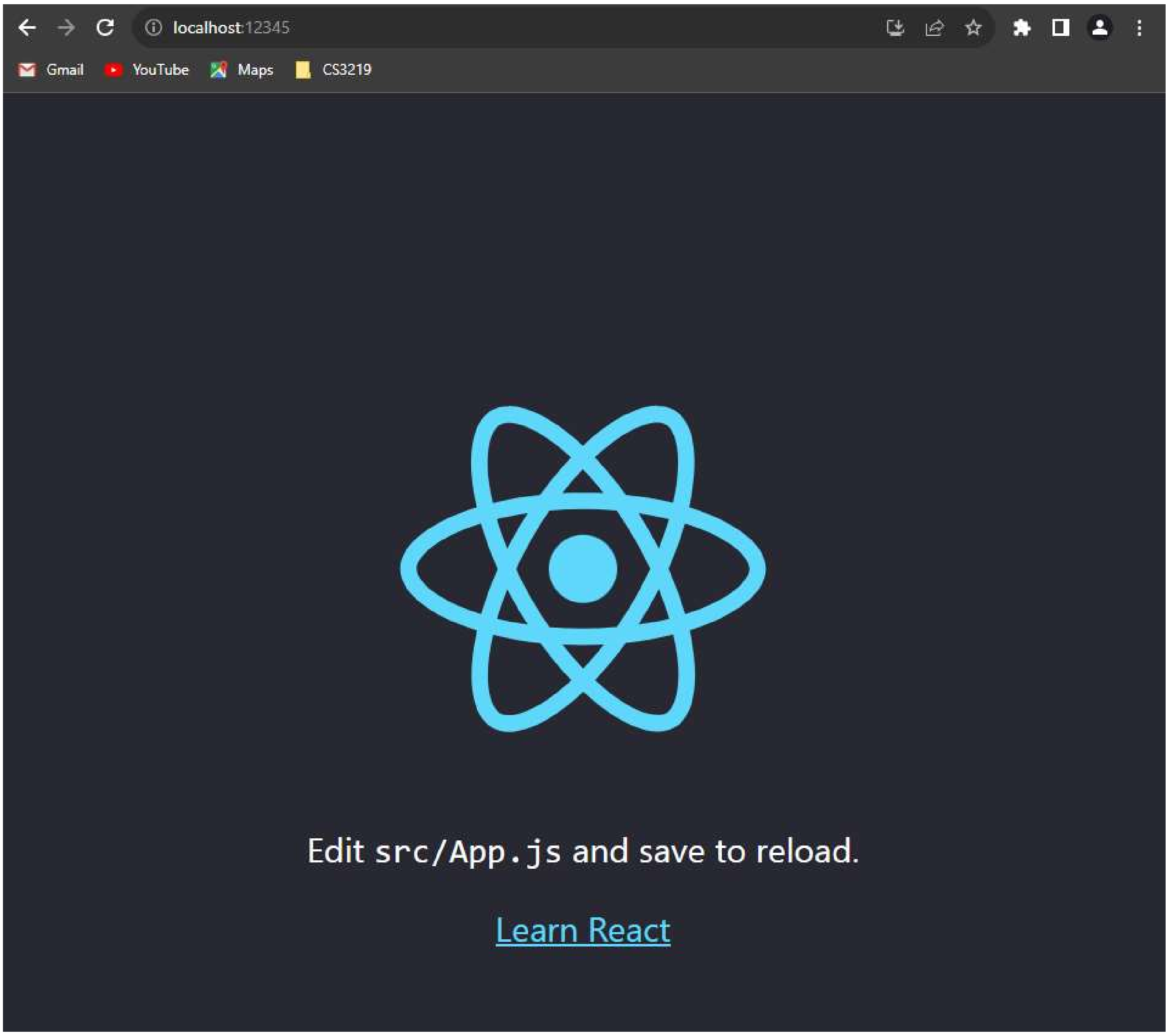
▶ ❓ **How to obtain the container ID?**

One you have run the command to get the container ID, the output also provides some interesting information - *port mappings*. You will get an output like this:

```
   CONTAINER ID   IMAGE                          COMMAND                  CREATED          S
b2bea1366fa2   <your username>/test-web-app   "docker-entrypoint.s…"   8 seconds ago   Up
```

Here, Docker mapped the port 8080 in the container to port 12345 on the host.

Access your web app at `http://localhost:12345/`. You should see the following page:



Edit `src/App.js` and save to reload.

Figure 1.5.3.1: React sample application running on localhost:12345

Yay! You have successfully dockerized your simple React app. 🎉

▶ ❓ **Did you remember to shut down the container once you were done?**

References

The information in this guide has been collated from the following sources:

- Docker Docs
- Docker Labs
- Dockerizing a Node.js Web App
- Deploy a React App with Node.js
- Docker Curriculum - A Docker Tutorial for Beginners
- Github Copilot was used to generate some of the content in the Docker guide.

- ChatGPT helped with providing an outline for the Docker guide.