

Practical: Containerise an application

Table of Contents


- [Working with a Dockerfile](#)
 - [Step 1: Obtain the Getting Started app](#)
 - [Step 2: Build the app's image](#)
 - [Step 3: Start an app container](#)
 - [Step 4: Inspect your application](#)
 - [Step 5: Modify the application](#)
- [Extension activity](#)
- [References](#)

In this activity, you will be working with a simple todo list manager that runs on Node.js.

Working with a Dockerfile

Step 1: Obtain the Getting Started app

Before you can run the application, you need to get the application source code onto your EC2 instance.

1. Clone the [getting-started-app repository](https://github.com/docker/getting-started-app/tree/main)  (<https://github.com/docker/getting-started-app/tree/main>) using the following command:

```
git clone https://github.com/docker/getting-started-app.git
```

2. View the contents of the cloned repository. You should see the following files and sub-directories.

```
ls -l getting-started-app/
```

```
├─ getting-started-app/  
| └─ .dockerignore  
| └─ package.json  
| └─ README.md  
| └─ spec/  
| └─ src/  
└─ yarn.lock
```

Step 2: Build the app's image

To build the image, you'll need to use a Dockerfile. A Dockerfile is simply a text-based file with no file extension that contains a script of instructions. Docker uses this script to build a container image.

1. In the `getting-started-app` directory, the same location as the `package.json` file, create a file named `Dockerfile`.

You can use the following commands to create a Dockerfile.

```
cd getting-started-app
```

Create an empty file named `Dockerfile`.

```
touch Dockerfile
```

2. Using a text editor or code editor (eg. `nano Dockerfile`), add the following contents to the Dockerfile:

```
# syntax=docker/dockerfile:1

FROM node:22-alpine
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

Here is an explanation of the Dockerfile:

`FROM node:22-alpine`

This line specifies the base image for your Docker image. The `node:22-alpine` image is a lightweight version of the Node.js 18 runtime environment, built on the Alpine Linux distribution. Alpine is known for its small size and security features, making it a popular choice for containerized applications.

`WORKDIR /app`

The `WORKDIR` instruction sets the working directory inside the container. All subsequent commands will be run from this directory. If the directory does not exist, it will be created. In this case, the working directory is set to `/app`.

`COPY . .`

The `COPY` instruction copies files and directories from the host machine (where the Docker build is being executed) into the Docker image. The first `.` refers to the current directory on the host, and the second `.` refers to the current directory inside the container, which is `/app` due to the previous `WORKDIR` instruction. Essentially, this line copies all the application files into the container.

```
RUN yarn install --production
```

The `RUN` instruction executes commands in a new layer on top of the current image and commits the results. Here, it runs `yarn install --production`, which installs the necessary Node.js dependencies defined in the `package.json` file. The `--production` flag ensures that only production dependencies are installed, excluding any development dependencies. This helps to keep the image size smaller and more secure.

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

The `CMD` instruction specifies the default command to run when the container starts. It takes an array of strings as arguments. In this case, it runs the Node.js application by executing `node src/index.js`. This means that when the container starts, it will run the script located at `src/index.js` using the Node.js runtime.

```
EXPOSE 3000
```

The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime. It does not publish the port to the host, but it's a way to document the intended network port. Here, port 3000 is exposed, which is a common port for web applications. This means that the application inside the container will be accessible on port 3000.

3. Build the image using the following commands:

In the terminal, make sure you're in the `getting-started-app` directory.

Build the image.

```
docker build -t getting-started .
```

The `docker build` command uses the Dockerfile to build a new image. You might have noticed that Docker downloaded a lot of "layers". This is because you instructed the builder that you wanted to start from the `node:22-alpine` image. But, since you didn't have that on your machine, Docker needed to download the image.

After Docker downloaded the image, the instructions from the Dockerfile copied in your application and used `yarn` to install your application's dependencies. The `CMD` directive specifies the default command to run when starting a container from this image.

Finally, the `-t` flag tags your image. Think of this as a human-readable name for the final image. Since you named the image `getting-started`, you can refer to that image when you run a container.

The `.` at the end of the `docker build` command tells Docker that it should look for the `Dockerfile` in the current directory.

Step 3: Start an app container

Now that you have an image, you can run the application in a container using the `docker run` command.

1. Run your container using the `docker run` command and specify the name of the image you just created:

```
docker run -dp 8080:3000 getting-started
```

The `-d` flag (short for `--detach`) runs the container in the background. This means that Docker starts your container and returns you to the terminal prompt. You can verify that a container is running with:

```
docker ps
```

or

```
docker container ls -a
```

The `-p` flag (short for `--publish`) creates a port mapping between the host and the container. The `-p` flag takes a string value in the format of `HOST:CONTAINER`, where `HOST` is the address on the host, and `CONTAINER` is the port on the container. The command publishes the container's port 3000 to `127.0.0.1:8080` (`localhost:8080`) on the host. Without the port mapping, you wouldn't be able to access the application from the host.

2. After a few seconds, open your web browser to the public DNS address of your EC2 instance.

Fetch the DNS address from the EC2 console.

Be sure to add `:8080` at the end of the address. Perhaps also remove `http://` or `https://`, in case your browser is trying ports 80 or 443, respectively.

You should see your app.

3. Add an item or two and see that it works as you expect. You can mark items as complete and remove them. Your frontend is successfully storing items in the backend.

Step 4: Inspect your application

At this point, you have a running todo list manager with a few items.

If you take a quick look at your containers, you should see at least one container running that's using the `getting-started` image and on port `8080`. To see your containers, you can use the CLI or Docker Desktop's graphical interface.

Run the following `docker ps` command in a terminal to list your containers.

```
$ docker ps
```

Output similar to the following should appear.

CONTAINER ID	IMAGE	COMMAND	CREATED
df784548666d	getting-started	"docker-entrypoint.s..."	2 minutes ago
ess_mcclintock	Up 2 minutes	127.0.0.1:8080->3000/tcp	prices

Step 5: Modify the application

Our container is running an application that was embedded within it upon being built. If we want to change the application source code, we must rebuild the image. Remember that images are immutable -- they cannot be changed after build.

We want to simply relabel the app.

Using nano, open `getting-started-app/src/static/index.html`.

Change the page title (in the header) from "Todo App" to "CAB432 Todo App".

Before rebuilding the image, refresh the app in your web browser. Notice the title of the tab (or window) did not change.

Let's rebuild the image

Find the ID of the running container

```
docker container ls -a
```

Stop the container then remove it

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

Find the ID of the image:

```
docker image ls
```

Remove the image:

```
docker image rm <id>
```




Go back to steps 2 and 3 to rebuild the image and run it in a container.

Extension activity

Given we have built the image with the application embedded within it, think of ways you could dynamically inject the application source into the container during run-time.

Look at using volumes

References

This practical was adapted from the Docker Guide titled "Containerize an application". The source code for that guide was licenced under the terms of Apache-2.0 licence. The original published page is [here](https://docs.docker.com/guides/workshop/02_our_app/) . The source is [here](https://github.com/docker/docs/blob/main/content/get-started/workshop/02_our_app.md) . The licence is [here](https://github.com/docker/docs/blob/main/LICENSE) . The changes made to the original document was to align with the use of EC2 in CAB432.

TEQSA PRV12079 | CRICOS 00213J | ABN 83 791 724 622