DOCKER

CSE5006 – LAB 2

TABLE OF CONTENTS

1. WHAT IS DOCKER?

Software environments can vary drastically, with a near-infinite number of possible combinations of operating systems, libraries, executables, and configurations. Unfortunately, this can often lead to a situation where applications developed locally work differently in a production environment. Environment-dependent bugs are usually subtle and difficult to resolve.

Docker offers a way of bundling up an application along with its software environment into a self-contained image. This means that a developer can, with confidence, assert that an application running locally will run the same way on a production server.

2. RUNNING A "HELLO WORLD" CONTAINER

When you buy a computer with no software installed, the first thing you would usually do is install an operating system from a disc or USB. For example, if you had a Windows 7 DVD lying around, you might pop it in and install Windows.

In Docker, an image is a bit like that Windows 7 DVD. However, all Docker images are based on Linux and are typically more customized and lightweight (usually no more than a few hundred megabytes). You can "install" an image by creating a container from it with the Docker command-line tool. One of the cool things about Docker is that you can create many different containers on one computer with isolated environments. For example, you could have one container for compiling Java programs, another container for running scientific simulations, and yet another for playing NetHack. If you "accidentally" delete all of the NetHack container's files in blind rage after losing a game, the Java container will be unaffected. Later, when you are feeling a bit calmer, you can create a new NetHack container from the original image.

A good way to think about images and containers in Docker is like classes and objects in object-oriented programming (recall that a class is like a blueprint from which objects are instantiated). Docker images are like classes, and containers are like object instances.

Docker Hub is an online registry hosting thousands of free Docker images that can be downloaded and used to create containers. Let's take a look at how we would go about doing exactly that.

Firstly, we will tell Docker to download the hello-world image. You can read about what's in this particular image by visiting https://hub.docker.com/_/hello-world/.

```
docker pull hello-world
```

After a few moments, you should be informed that the image has been downloaded. Now we can create a new container from the image.

```
docker create --name=my-container hello-world
```

To check that the container was created successfully, issue the following command.

```
docker ps --all
```

This will list all Docker containers on the system, including information like ID, status, and the image from which they were created. This command is incredibly useful, and I recommend running it often to check up on your containers.

Now it's time to actually run the main program inside the container.

```
docker start --attach my-container
```

If all went well, you should be presented with a "Hello from Docker" message accompanied by some other informative text. Note that although the program has terminated, the container you created still exists. Using a Docker command you have already learned (hint: it rhymes with "locker tree-mess hall"), check the status of the container. Does the status make sense given what we have done?

To actually remove the container from the system, another command is required.

```
docker rm my-container
```

Confirm that the container is no longer present by listing Docker containers. For convenience, Docker provides a handy shortcut command which pulls the image if it doesn't exist, creates a container, runs it, and removes it upon termination. This allows us to achieve what previously took us four commands (pull, create, start, rm) with only one.

```
docker run --rm --name=my-container hello-world
```

You should see the same message as before.

2.1. EXERCISE 1

Create (but don't run) a hello-world container without specifying a name (omit the --name option completely). What is reported in the name field when you list Docker containers? Try removing the container using its ID rather than its name. Does it work? Put your answers from this and all other exercises in a text file called EXERCISES and make sure that it's committed to your repository.

3. CREATING A FORTUNE COOKIE IMAGE

For the remainder of this lab, we will be creating a simple web application. Based on what you learned in the first lab, use Git to keep track of changes and store your code on GitHub. Your demonstrator will be checking your use of version control, so make commits at your discretion as you work through the lab.

3.1. HUMBLE BEGINNINGS

Using images from Docker Hub is great when you want to run pre-existing software. However, when it comes to running your own custom programs, Docker provides a solution. Similar to object-oriented programming, Docker allows you to extend existing images, inheriting all the components present in the base image. This way, you can customize the image by adding whatever you need, including dependencies and your own applications.

Today, our goal is to create a very simple website that displays a new fortune cookie message every time the page is loaded, and then build it into a Docker image. To start, we need to establish a basic project structure.

1. Create a new directory in Lab02 called fortune-cookie, and initialize a Git repository.
2. Start Visual Studio Code and open the newly created folder.
3. From inside Visual Studio Code, create three new files called **Dockerfile**, **package.json**, and **server.js**.

```
# Base this image on an official Node.js long term support image.
FROM node:18.16.0-alpine

# Install Tini using Alpine Linux's package manager, apk.
RUN apk add --no-cache tini

# Use Tini as the init process. Tini will take care of system stuff
# for us, like forwarding signals and reaping zombie processes.
ENTRYPOINT ["/sbin/tini", "--"]
```

The first part of the file tells Docker that we want our image to be based on **node:18.0.0-alpine**, which is an official Alpine Linux-based image containing Node.js. More on Node.js later; for now, suffice to say that we will be using it to make our web application.

The rest of the file tells Docker how we want to modify the base image. Here we are downloading a small program called Tini, which will act as the init process. It is a good idea to use a proper init process with Docker, but we won't go into the details here. However, you should pay attention to how we add Tini to the Docker image. The RUN step runs a command as if we had a terminal inside the Docker image - in this case, we are simply downloading and installing Tini with Alpine Linux's package manager, **apk**. The **ENTRYPOINT** step tells Docker that when we run a container, we want it to use Tini as the init process.

Start a terminal in the fortune-cookie directory and build our custom Docker image.

```
docker build --tag=fortune-cookie .
```

You should see each step run in order, ending with a message that indicates the image was successfully built. The tag option gives our image a name, just like the hello-world image we encountered earlier. We can confirm that the image is now available on the system by listing all Docker images.

```
docker images
```

To test the image, we can run it by telling it to start a shell session inside a container (sh is a shell that's already installed in the base image).

```
docker run --rm -it fortune-cookie sh
```

You will be presented with a shell prompt. You can verify that you are indeed inside the image by checking that Tini is available.

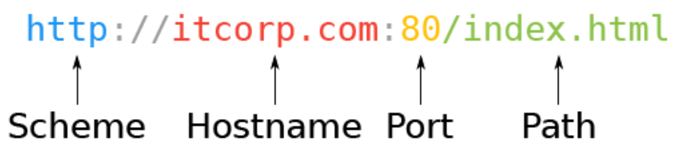
```
tini -h
```

If you see usage instructions for Tini, then everything is going according to plan. Our image (and therefore the container) is still quite bare at the moment. Press **Ctrl+D** to exit the shell so we can continue our work.

3.2. A QUICK ASIDE: NETWORKING ESSENTIALS

Computers are connected over a network using sockets. This is also true for connecting to web servers, as we will demonstrate in this section.

If you provide me with a URL, such as http://itcorp.com:80/index.html, for instance, I can already navigate to the associated website using a web browser. However, what may be less familiar to you is how this process involves low-level socket communication to retrieve the relevant content. First, let's break down the anatomy of a URL.



Part	*Description*
Scheme	The protocol to use when communicating with the server
Hostname	The name of the server we wish to connect to.
Port	Port number on the server to use for communication.
Path	The resource we are requesting from the server.

One part of URLs that you might not encounter frequently is the port number. This is because when an HTTP URL doesn't explicitly include a port, the default value of port 80 is assumed. So, http://itcorp.com:80/index.html and http://itcorp.com/index.html are equivalent. Port numbers enable multiple server applications to operate on the same host simultaneously, as each application can be assigned a different port.

Although the hostname is a convenient and human-readable way of expressing the name of a server, computers actually communicate using IP addresses. To obtain an IP address from a domain name, we need to query a Domain Name System (DNS) that will map the hostname to the corresponding IP address. We can use the **nslookup** command to perform this lookup explicitly.

```
nslookup itcorp.com
```

Under the "Non-authoritative answer" section of the output, you should see the IP address associated with itcorp.com. Now we have the two essential pieces of information required to initialize socket communication with the server: the IP address and the port number (default HTTP port 80) of the web application.

```
telnet <ip_address> 80
```

The telnet command utilizes sockets to connect to the application running on port 80 of the itcorp.com server. Please type in the following HTTP request, which includes the path of the desired resource. Our web browsers handle this process automatically when we visit a website.

```
GET /index.html HTTP/1.0
```

Hit enter twice. You should be presented with the same HTML response you received when entering the URL in a web browser.

3.2.1. EXERCISE 2

Try finding "the" IP address for amazon.com with **nslookup**. What do you find, and why do you think that might be the case for a big company like Amazon? Don't forget to note your answers.

3.3. RUNNING A SIMPLE NODE.JS APPLICATION

Node.js is a runtime environment for developing JavaScript applications that run on servers. Dependencies for a Node application are specified in the **package.json** file, which we will create now. Since this is a simple project, we only have one dependency - Express, which is a minimalist web framework. While we're at it, we can also add a project name and version number and indicate that the project is not intended for public release.

```
{
    "name": "fortune-cookie",
    "version": "1.0.0",
    "private": true,
    "dependencies": {
        "express": "4.15.3"
    }
}
```

With that over with, we can create a simple web application just to check that Express is working. Create **server.js** and add a basic "Hello World" server program.

```
// Require the Express module
const express = require('express');
// Create an Express web application
const app = express();

// Specify how to respond to GET /
app.get('/', (req, res) => {
    res.send('Hello World!');
});

// Start listening for HTTP requests on port 3000
app.listen(3000, () => {
    console.log('Server started');
});
```

Let's take a moment to break this down. Firstly, we require the Express module and use it to create an app object. Secondly, we specify that when the server receives a request on the index of the site, it should respond with the message "Hello World!". Lastly, we start the server and listen for requests on port 3000 (as opposed to the default HTTP port 80).

Now that we actually have some custom software, we need to add it to our Docker image. Open up the Dockerfile again and add the following lines to the bottom, below the steps already there.

```
# ...steps from before...

# Create a working directory for our application.
RUN mkdir -p /app
WORKDIR /app

# Install the project's NPM dependencies.
COPY package.json /app/
RUN npm --silent install
RUN mkdir /deps && mv node_modules /deps/node_modules

# Set environment variables to point to the installed NPM modules.
ENV NODE_PATH=/deps/node_modules \
PATH=/deps/node_modules/.bin:$PATH

# Copy our application files into the image.
COPY . /app

# Switch to a non-privileged user for running commands.
RUN chown -R node:node /app /deps
USER node

# Expose container port 3000.
EXPOSE 3000

# Set the default command to use for `docker run`.
# `npm start` simply starts our server.
CMD [ "npm", "start" ]
```

Build the image again, then run the container with port 3000 published on the host.

```
docker run --rm -it --publish=3000:3000 fortune-cookie
```

Publishing the port is necessary to make the host machine forward socket connections to the container image. Navigate to http://localhost:3000 to see the application in action. When you're done, press Ctrl+C in the terminal to terminate the web server.

3.3.1. EXERCISE 3

Look up the Docker run reference to find out how to map container port 3000 to the default HTTP port 80 on the host. Note that **-p** is simply a shorthand for typing **--publish**. If done correctly, you should be able to access your web application in the web browser without specifying a port number.

3.4. ADDING FORTUNE COOKIE MESSAGES

Since the Docker image we're using is Alpine-based, we can add new software to the system using Alpine's package manager, **apk**. Fortunately for us (pun intended), there is a program available called fortune, which prints out messages in the style of a fortune cookie. We can install that program into our image by adding another **RUN** step to our Dockerfile. Insert the following **RUN** step just after the **ENTRYPOINT** step.

```
# ...

# Install Fortune.
RUN apk add --no-cache fortune

# ...
```

Rebuild the image. Note that Tini is not downloaded again. This is because each step of the Dockerfile results in its own cached layer, which can be reused in subsequent builds.

Now we can tweak our **server.js** code to call fortune and respond with that message instead of a hello.

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

// Create a new Express web application
const app = express();

// Set handler for the index of the website
app.get('/', (req, res) => {
    // Run the system `fortune` command and respond with the message
    child_process.exec('fortune', (error, message) => {
        if(error === null) {
            res.send(message);
        } else {
            res.send('Error: ' + error);
        }
    })
});

// Start web application server
app.listen(3000, () => {
    console.log('Server started');
});
```

Build the image once more and run it. Open the website in a browser window and refresh a few times. You should be presented with a new fortune each time. Make sure that you add, commit, and push all changes to your repository at this point.

3.4.1. EXERCISE 4

Modify the server application so that it returns the current date and time along with the fortune message. In JavaScript, you can get the current date and time as a string by calling the **Date()** function. Note: you should only ever call **res.send()** once per request, so make sure that you build the entire response string and send it with a single function call.

3.5. USING DOCKER COMPOSE

Docker Compose is a handy tool that allows you to specify Docker command line options in a configuration file. This means that instead of copy-pasting a bunch of options all the time, you can specify them in a file once.

Docker Compose automatically looks for a file called **docker-compose.yml** for all of the information it needs to start your application. This file can contain one or more services, each of which results in a single container being created. Since our application only requires a single container for the moment, we will create a single entry. We will call our service "web" and nest all options under that section (**correct indentation is important**).

```yaml
version: "3.8"

services:
    web:
        build: .
        ports:
            - "3000:3000"
```

This makes the commands for building and running much simpler. From now on, we will be using these Docker Compose commands instead of **docker build** and **docker run**.

```
docker compose build

docker compose up
```

It is not necessary to run **docker-compose build** again until the image needs to change. Besides **build** and **up**, Docker Compose has another useful command called **run**. This will start a new service instance and run the command specified. For example, if we wanted to run **fortune** directly, we could do so.

```
docker compose run --rm web fortune
```

This command will run the command **fortune** inside a web container, replacing the **CMD** line defined in the Dockerfile. The **--rm** option removes the container once it exits.

3.6. SHARING DIRECTORIES VIA VOLUMES

Currently, we copy our application source files into the image at build time using a **COPY** step in the Dockerfile. This is great for creating a production image, but it's not very convenient during development time when changes are being made rapidly. Fortunately, there is an alternative - volumes. Volumes in Docker are like shared folders. You simply specify a location on the host to map to a location in the container, and they will automatically be kept in sync while the container is running.

Add a **volumes** section to the **web** service in your **docker-compose.yml**.

```
version: "3.8"

services:
    web:
        build: .
        ports:
            - "3000:3000"
        volumes:
            - ".:/app"
```

This new section can be read as: "create a volume that maps the current directory on the host to the /app directory in the container."

Here's a step-by-step summary of what happens with our application files:

- During image build time,
    - Dependencies are installed in **/app/node_modules** by the **npm** install RUN step.
    - The installed dependencies are moved to **/deps/node_modules**, and the NODE_PATH environment variable is set appropriately so that our application knows where to find them.
    - The rest of the source files are copied into /app. This means that a fixed snapshot of the application is present inside the image for when we want to deploy the image onto a production server.
- During container start time,
    - A volume is created which maps to **/app** in the container, overwriting its previous contents with the directory on the host. We only want to do this during development, because source files are rapidly changing – in production we would just use the version of the application bundled in the image.
    - Since we moved the project dependencies to **/deps/node_modules**, our dependencies from build time will still be presented in the container.

Use Docker Compose to build the image and bring the service up. It should run like sunshine and rainbows, just as before. However, we haven't quite achieved our goal yet - try making an edit to server.js and reload the web page. Nothing changes... it didn't work... panic mode... PANIC! Whoa, calm down there. The changes are actually present in the container, but since we haven't restarted the Node server, it will continue using the old code already loaded in memory. To get around this limitation, we will use a wrapper program called Nodemon. Nodemon monitors source files and restarts Node when it detects a change.

1. Add "nodemon" version 1.11.0 as a dependency below Express in **package.json**, like so:

```
"dependencies": {
    "express": "4.15.3",
    "nodemon": "1.11.0"
}
```

2. In the Dockerfile, change the CMD step to start Nodemon.

```
CMD [ "nodemon", "-L", "-x", "npm start" ]
```

3. Once again, build and bring the application up using Docker Compose. The output should indicate that Nodemon has started and is watching ***.*** (which basically means all files).
4. Make a change in **server.js**, such as adding a prefix to the message saying "Your fortune is: ". Save the file and refresh the web page in your browser (without rebuilding the image or performing any other actions). If everything went well, the changes should be immediately visible.

One thing to note is that the current setup still requires an image rebuild when new dependencies are added because downloading dependencies is part of the image creation process.

Add, commit and push your changes to GitHub once again.

3.6.1. EXERCISE 5

You may have noticed that the time shown from **Date()** is in UTC, not local time. This is because the default time zone set in the **node:18.16.0-alpine** Docker image is UTC. One way to change this behavior is to mount the time zone settings from the host into the Docker image. Add a line to **docker-compose.yml** which mounts **/etc/localtime** into the container.

```
volumes:
    - ".:/app"
    - "/etc/localtime:/etc/localtime:ro"
```

The "ro" part tells Docker that we want read-only access to the file - this prevents the container from being able to change the host's time zone setting. Rebuild and run the application. Refresh the web page and check that the time is displayed in local time (you should see AEST in parentheses).