SSE Lab 9 - Containers and Docker

Work in Pairs

In this lab, we will try out running software packaged as *containers*. Containers are lightweight, portable, and self-sufficient units that can run applications in isolation from the host system. We can build a *container image* that packages up everything needed to run a particular application (the code, dependencies, etc etc), download that on to a machine (e.g. your laptop) and run in there, without installing the application directly onto the host computer. This gives us great flexibility in running the same application on different machines.

Docker is a popular platform for developing, shipping, and running applications in containers. In this lab we'll use Docker to run some existing applications in containers, and to build our own container images.

Part 1: Installing Docker   (Completing parts 1-4 will give 60% of the marks)

Unfortunately, Docker is not installed on the lab machines (this is because Docker allows users to gain greater access privileges than normal, so you could potentially access files belonging to other users etc).

The easiest option for this lab is probably to install it on your own laptop. See the instructions at https://docs.docker.com/get-docker/ for how to do this, depending on your operating system.

You can't install Docker on the lab machines yourself, as you don't have admin ("root") privileges. However, you did have root privileges when you created a virtual machine in Azure. So, if you wanted to, instead of installing it on your laptop, you could create a Linux VM in Azure like we did last week, and install and run Docker on the VM.

Part 2: Running an Existing Image

There are a lot of existing applications already packaged up as Docker images. A common place for these to be published is Docker Hub - https://hub.docker.com/. Much like when we use Git and GitHub, we can pull public images down from Docker Hub to copy them onto our machine. Then we can run containers on our machine based on these images.

A good way to test that your Docker setup is working is to try the following commands:

```
$ docker pull docker.io/library/hello-world
```

This downloads the "hello-world" image from Docker Hub. As it happens, Docker Hub is the default location to pull from, so equivalently you could just do… `$ docker pull hello-world`

You can then run the application in a container by using the Docker run command:

```
$ docker run hello-world
```

Hopefully you should see

```
Hello from Docker!
This message shows that your installation appears to be working
correctly.
…
```

What happened here was that Docker started a new container based on the hello-world image, running on your computer. It then ran the default program inside that container, which printed some text to the console, and then exited. At this point the container was stopped.

Let's see what we can do if we use a different container.

First of all, let's try running a container from an image of Ubuntu Linux. This means that you can run Linux on your computer, even if your normal operating system is Windows or Mac OS.

Let's fetch the Ubuntu image from Docker Hub.

```
$ docker pull ubuntu
```

This time we'd like to open a terminal so that we can interact with the container as it runs, so we run:

```
$ docker run -it ubuntu bash
```

This creates an interactive session (that's what `-it` does) and opens a bash shell inside the container where you can run commands. When you exit from the shell, the container will stop running.

If you want to go back in to a container later on, to carry on your work, but it has exited, you can use

```
$ docker container ls -a
```

to list recent containers (including those that have exited), then restart your chosen container, and attach a new terminal. We use **docker run** to create a new container, **docker start** to re-start containers that have stopped, and **docker exec** to run commands on an already-running container.

```
$ docker start <CONTAINER_NAME>
$ docker exec -it <CONTAINER_NAME> bash
```

To clean up old containers and reclaim some disk space on your laptop you can use

```
$ docker container prune
```

Part 3: Running a Server in a Container

We can also use a container to run a long-running process, like a web server. Let's try a popular open-source web server, nginx (pronounced "engine-x").

Pull the image from Docker hub and run a container - see https://hub.docker.com/_/nginx

nginx can (amongst other thing) serve static webpages, images, etc. Out of the box it serves a default web page on port 80. But if we open a browser at http://localhost:80 our browser will look for a web server running on port 80 of our own computer, not the container, so you won't see anything.

To fix this, we use a technique called *port forwarding* - we can map a port on the host computer to a port on the container. So if for example we request http://localhost:8080 we can forward that request to the container's port 80, this will make the appropriate request to nginx and we can see our content.

To make this mapping, we add a flag to our Docker run command, e.g.

```
$ docker run -p 8080:80 …
```

If you do this for your nginx container then you should be able to open a browser at http://localhost:8080 and see a (fairly boring) web page.

Updating the web content

There are two ways to edit or add to the web content served by nginx.

1) Edit files inside the container
Inside the container is a directory /usr/share/nginx/html which contains an index.html file.
From another terminal, use docker exec -it bash to attach to the running nginx and create a shell where you can run commands. Navigate to /usr/share/nginx/html and edit the files. You may need to install an editor!

2) Let the container read files from the host computer
When you start the container, you can map a directory inside the container to a directory on the host machine (e.g. your laptop). Then if you edit a file in that directory on your machine, you should see the changes reflected directly on the web page.

To do this add an additional flag to your docker run command

```
  -v /path/to/your/html:/usr/share/nginx/html
```

This means that when the container looks in /usr/share/nginx/html to find a file, it will actually look in /path/to/your/html on your machine. This can be good for development (but not for production).

Try adding your own HTML file and make sure it is served through nginx when you view it in the browser.

Part 4: Building our own Images

We've seen how to use existing images, pulling them from Docker hub and running them on our machine, but what if we want to create and publish our own images?

Docker builds images using recipes that we write in a file called a Dockerfile.

We won't give you a full tutorial on writing Dockerfiles, there are lots of resources online that can help you with that.

Create a directory on your computer, add an empty file called Dockerfile, and create an index.html file along side it with some simple content. **Put your names in the HTML.**

Write a simple Dockerfile that does the following:

- Uses an official Nginx runtime as a base image
- Sets the working directory to /usr/share/nginx/html
- Copies some HTML files into the container at /usr/share/nginx/html
- Exposes port 80 to the outside world
- Sets up a command to run nginx when we run the container

Now ask Docker to build an image based on this recipe. It should start with the plain nginx image, add your HTML content, set a couple of other configuration options, and wrap that all up as a new image.

```
docker build -t nginx-with-my-page .
```

Don't forget the dot at the end, that says where the Dockerfile is. The -t flag gives the image a name.

Run a container based on your new image and check the contents in a browser.

For the submission, take a **screenshot** of your terminal running and your browser window showing your webpage.

Part 5: Publishing our Images (20%)

If we want to publish our image somewhere where it can be used by other people, or on other machines, we can use a Container Registry. You've already used one container registry, Docker Hub. Another popular option is the GitHub Container Register (GHCR). You can use GHCR with your existing GitHub account.

Look up how to push your new image to GHCR. Make the image public so that the markers can check it.

Part 6: Extension - packaging a web application (20%)

Try to package a Python Flask application and publish it to GHCR in the same way.

What to submit:

Once you have completed the parts above, please submit a PDF file to Scientia containing the following:

- The names and DoC username of the people in your pair
- Your screenshots from Part 3 showing Docker running and your web page in the browser
- Your Dockerfile and the link to your image on GHCR from parts 4 and 5
- Dockerfile and link for Part 6 (extension) if you tried that.