Containers

As described in lecture 4 containers are a staple of packaging applications in a self-contained fashion. Linux containers contain everything a Linux application needs to run on a Linux kernel in an isolated fashion. This exercise will run you through the basics of packaging an application in a container and running it on a Linux server.

This guide is an abridged version of "Getting started with Docker for the Inquisitive Mind" by Janos Pasztor, one of the lecturers on this course. We highly recommend reading the original version for more detail.

Installing Docker

One of the most popular small-scale container runtimes, and indeed the first to popularize build recipes is Docker. Docker can be installed on Linux as well as Windows and MacOS, but the latter two have several issues a user needs to work around so we recommend using an Ubuntu Linux 20.04 virtual machine for the purposes of this exercise.

To install Docker you can either follow the installation guide or use the convenience script to do so:

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
```

Please note that this script is not recommended for production use, but will be accepted in a solution to the project work.

Creating a basic Dockerfile

In order to package an application you can create a `Dockerfile` containing the instructions to install the dependencies and package an application. This file consists of the following format:

```
COMMAND PARAMETERS
```

The first command is the `FROM` command. This command specifies which base image to use, for example:

```
FROM ubuntu:20.04
```

As mentioned before, containers are not virtual machines. This command only pulls in the userland piece of an Ubuntu system. It can be used to install libraries and packages required for an application to run.

As this file has now been created we can build our container image as follows:

```
docker build -t myimagename .
```

> **Warning**
>
> The dot at the end is important! It tells Docker to look for the Dockerfile in the current directory.

Now that the image is built it can be launched:

```
docker run -ti myimagename
```

> **Tip**
>
> The `-ti` parameter is only required for interactive sessions. For non-interactive sessions it is not required.

Inside the container you can do operations just as you normally would. For example, you could run `apt update`, then `apt install mc`, and finally `mc` to launch the midnight commander. Once you are finished with the interactive session you can use the `exit` command or `Ctrl` + `D` to exit.

However, as discussed in lecture 4 the main benefit containers bring to the table is reusability. If you treat a container like you would treat a virtual machine, updating, installing things manually, you lose the main benefit of using containers.

Let's extend our `Dockerfile` to install the `nginx` webserver:

```
FROM ubuntu:20.04

ENV DEBIAN_FRONTEND=noninteractive

RUN apt update
RUN apt install nginx
```

There are two new commands here: the `ENV` and the `RUN` command. The `ENV` command instructs Docker to set an environment variable. The `DEBIAN_FRONTEND=noninteractive` environment variable instructs the `apt` and `dpkg` packaging utilities in Ubuntu to run in non-interactive mode and not ask questions. Instead, these commands will use default values. The `RUN` command simply runs the specified command.

If you run the `docker build .` command now the build will fail:

```
...
After this operation, 60.8 MB of additional disk space will be used.
Do you want to continue? [Y/n] Abort.
The command '/bin/sh -c apt install nginx' returned a non-zero code: 1
```

As you can see, the `apt install` command is asking for permission to continue with the installation. This teaches us an important aspect of containerization: the `Dockerfile` must contain only commands that run *automatically*. Let's fix the file:

```
...
RUN apt install -y nginx
```

The `-y` is specific to `apt` and means that the user wishes to answer `yes` to everything.

OK, so let's run the build again:

```
docker build -t mynginx .
```

```
...
 ---> 85e4def234e1
Successfully built 85e4def234e1
```

So, we run our image:

```
docker run -ti -p 80:80 mynginx
```

The `-p` flag tells Docker to map the port 80 of the container to the port 80 of the host. This will allow you to access the service on the IP address of your server. However, we didn't tell Docker to run nginx, so we receive a shell and nothing else. Sure enough, we can *start* nginx manually, but that's not what we want. Before we exit our container let's find out where nginx is located:

```
which nginx
```

This will print the full path of the nginx binary which we will need in a moment. Let's exit the container with `Ctrl` + `D`.

In order to make Docker actually *run* nginx we need to employ the `CMD` command in our `Dockerfile`:

```
CMD ["/usr/sbin/nginx"]
```

A word about `CMD`

The `CMD` command has a second form: `CMD nginx parameters here`. In this form Docker cannot pass the parameters directly to the underlying execv call that will ultimately launch the application. `execv` expects all parameters in a separated list. The binary itself has to be passed as a full path as the first parameter.

To transform the `CMD` parameter Docker launches a *shell* in the container, for example Bash. This shell will then parse the parameters and ultimately launch the desired program.

This may seem like a reasonable approach at first, but can cause problems. The first process in a container is responsible for handling signals, such as the `TERM` signal that indicates that a process should stop gracefully. Bash does not do this. Instead, Bash will ignore the signal and Docker will wait for 30 seconds and then forcefully kill the container.

This is the same reason why you need to be careful when running a shell script in your `CMD`. If you want to do that you must either handle signals correctly, or launch your subsequent program with the `exec` stanza. The `exec` stanza *replaces* the current shell with the new program instead of launching a child process:

```
#!/bin/bash

set -e

# Do some initialization here

exec /usr/sbin/nginx
```

There is also a second command in Docker called `ENTRYPOINT`. The `ENTRYPOINT` command allows you to specify an additional program that acts as a *wrapper* for `CMD`. Let's say you have the `CMD` from above, and you specify the `ENTRYPOINT` of `/init.sh`. In this case the full command that will be launched is `/init.sh /usr/sbin/nginx`. In other words, the contents of `CMD` are passed to the `ENTRYPOINT`.

Getting nginx running

Back to the nginx example. If you launch the container you will see that it exits immediately. As mentioned before, the first process in a container has a special role. This special role is not only to handle signals, but also extends to running the container. If the first process exits the container will stop.

In our case, nginx is doing something called *daemonization*. Daemonization on Linux means that a process launches a second copy of itself and exits from the first copy in order to give back control over the console to the user. In a container we want exactly the opposite: we do not want nginx to daemonize. This is achieved by setting the `daemon off;` parameter in the configuration file, or passing it via the command line:

```
CMD ["/usr/sbin/nginx", "-g", "daemon off;"]
```

If you now build and run the container you will see that it stays up and you can access the web service on the IP address of your machine running Docker.

Applying configuration files

Having a container with only nginx in it is not very useful, so let's see how we can put some *content* on the web server. First, let's figure out where the *document root* of our web server is. We do this by *overriding* the `CMD` for out container and launching a shell in it:

```
docker run -ti mynginx /bin/bash
```

This will land us in a shell inside a new container. In the container we can search for the document root:

```
grep -nr 'root ' /etc/nginx | grep -v '#'
```

This will give us the following result:

```
/etc/nginx/sites-available/default:41:  root /var/www/html;
```

So, the document root is in `/var/www/html`. Let's create an `index.html` file which we will copy to the container in the next step:

```
<h1>Hello world!</h1>
```

Copying files to the container:

```
COPY index.html /var/www/html/index.html
```

If you rebuild and relaunch the container you will see that the default nginx page is replaced by our `Hello world!`.

Before we proceed, let's document that our container exposes a service on port 80:

```
EXPOSE 80
```

This is not strictly necessary and only serves a documentation purpose, but it is nevertheless useful.

Building a software in a container

To summarize: in order to containerize our application we need to install the dependencies of your application (e.g. a web server), then copy your compiled application in your container, and then set your `CMD` to launch your application.

You also have the option to compile your application inside the container. To avoid creating extremely large containers with the build tools we will use multistage builds.

Let's create a very simple Go application in a file called `main.go`:

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello world!")
}

func main() {
    http.HandleFunc("/", handler)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

This program will launch a simple web server on port 8080. Let's also add a `go.mod` file to enable Go modules:

```
module github.com/yourusername/smaple

go 1.14
```

Let's create a `Dockerfile` that compiles this application:

```
FROM alpine AS build
RUN apk add --update go
RUN mkdir -p /srv/myapp
WORKDIR /srv/myapp
COPY main.go .
COPY go.mod .
ENV CGO_ENABLED=0
RUN go build -tags netgo -a -v main.go
RUN chmod +x main
```

In the next stage we will copy our compiled application from the first stage and run it:

```
FROM alpine AS run
RUN apk add --no-cache libc6-compat
```

```
COPY --from=build /srv/myapp/main /app
CMD ["/app"]
EXPOSE 8080
```

Finally, let's run it:

```
docker build -t myapp .
docker run -d -p 8080:8080 myapp
```

There you go! That's your app built and running in a single file. You can see a few additional commands above, feel free to look them up in the official documentation.

Volumes

One last item we need to talk about are *volumes*. Containers are designed to be *immutable*. When a new version comes along we simply destroy the existing container and create a new one. Some services like database, however, need to store data in a persistent fashion. That's what we need volumes for.

In the container world we can mount volumes directly from the host machine, or we can mount a network-based storage as discussed in the lectures. In our example we will showcase how to mount a volume from a local folder using Docker.

Assuming we want to launch a webserver we can mount the document root from our previous example:

```
docker run -d -p 80:80 -v /srv/www:/var/www/html mynginx
```

In this case the `/srv/www` folder of the host machine will be mounted in `/var/www/html` inside the container. This can be used to persist data, but also during development when files change frequently.

Security hardening

One more aspect of containerization we need to talk about is security. With our configuration above our container is running as root. While the container has a security boundary, running as root can still present a security risk. In fact, enterprise Kubernetes setups like OpenShift do not allow containers to run as root.

Let's change our Go container so it doesn't run as root:

```
...
FROM alpine AS run
RUN addgroup -S --gid 1000 app && adduser -S app --uid 1000 -G app
COPY --from=build /srv/myapp/main /app
CMD ["/app"]
USER 1000:1000
EXPOSE 8080
```

Note that we are specifying the user ID and group ID to run as in numeric form (`1000`). The `USER` command allows user names to be passed as well, but in a hardened Kubernetes setup this will not be accepted.