Chapter 6: Docker in the Real World

6.1. Introduction

Nothing important in this chapter.

6.2. A simple web-application with Docker

Nothing important in this chapter.

6.3. Creating a Dockerfile (part 1)

Docker images are basically stack of layers and Dockerfile is just a recipe.

The first instruction in a Dockerfile must be `FROM`. `FROM` allows us to import/define a base image. A base image could be another docker image, or we can create one from scratch. It's highly recommended to use an official image.

`RUN` allows you to run any command, that you can run on your OS, without docker.

`WORKDIR` expects you to pass a directory, and any other instruction we run from this point on will be on the context of the path we set here.

`COPY` simply copies files from the directory where your Dockerfile exists. You can't copy files that are above Dockerfile, you can only copy files that are below your Dockerfile.

Dockerfile after this chapter:

```
FROM ruby:2.5-alpine

RUN mkdir /app
WORKDIR /app

COPY Gemfile Gemfile.lock /app
RUN bundle install --jobs 4 --retry 3
COPY . .
```

6.4. Creating a Dockerfile (part 2)

Docker caches every layer. So it makes sense to copy `Gemfile` and `Gemfile.lock` before copying entire project folder. Otherwise if we configure our project like this:

```
FROM ruby:2.5-alpine

RUN mkdir /app
WORKDIR /app

COPY . .
RUN bundle install --jobs 4 --retry 3
```

It's going to install our dependencies every time from scratch and it's gonna take more time.

`LABEL` command enables you to attach arbitrary meta data to your image, that you can retreive later.

```
LABEL maintainer="M. Serhat Dundar <msdundars@gmail.com>"
      version="0.1"
```

`CMD` is a little bit interesting. It defines the default command to be run when the docker image gets started. `CMD` is different than the `RUN` command, because it's executed when the docker image gets ran, as opposed to the `RUN` command, that is executed when the docker image gets built. We can use `CMD` to run the server when image gets run.

```
CMD bundle exec rails server -b "0.0.0.0" -p 3000
```

Dockerfile after this chapter:

```
FROM ruby:2.5-alpine

RUN mkdir /app
WORKDIR /app

COPY Gemfile Gemfile.lock /app/
RUN bundle install --jobs 4 --retry 3
COPY . .

LABEL maintainer="M. Serhat Dundar <msdundar@babbel.com>" \
  version="0.1"

CMD bundle exec rails server -b "0.0.0.0" -p 3000
```

6.5. Building and pushing Docker images

Get help:

```
docker --help
```

Build the image:

```
docker image build -t deeplinks_resolver .
```

Inspect an image:

```
docker image inspect deeplinks_resolver
```

Build the image with a tag:

```
docker image build -t deeplinks_resolver:1.0 .
```

List all docker images on your computer:

```
docker image ls
```

Delete an image:

```
docker image rm deeplinks_resolver:1.0
```

> We can use repository, repository+tag and image ID options to delete an image.

We don't need to type full ID to delete an image, just some characters are fine:

```
docker image rm 1234
```

Login to Docker hub:

```
docker login
```

Before pushing our image to hub we must tag it with our username:

```
docker image tag deeplinks_resolver msdundar/deeplinks_resolver:latest
```

Push the latest version of the image:

```
docker image push msdundar/deeplinks_resolver:latest
```

Pull the image from docker hub:

```
docker pull msdundar/deeplinks_resolver
```

6.6. Running Docker containers

List all running containers:

```
docker container ls
```

Run a docker container:

```
docker container run
```

Run a docker container in interactive mod and with ports:

```
docker container run -it -p 3000:3000 -e RAILS_ENV=development -e
SOMETHING_ELSE=foo deeplinks_resolver:0.1
```

> `-it` flag enables terminal colors, CTRL+C key and makes docker container interactive The first port is the bind port under docker host, and the second port is the bind port withing the docker container -e flag allows us to pass environment variables

We can hit CTRL+C to stop our container. Then we will not be seeing our container if we run `docker container ls`.

Stopped containers don't use disk space, but it's always a good idea to destroy stopped containers.

We can list stopped containers as follows:

```
docker container ls -a
```

We can delete stopped containers as follows:

```
docker container rm CONTAINER_ID|CONTAINER_NAME
```

But of course it's annoying to remove them manually. So we can automate this process with `--rm` flag:

```
docker container run -it -p 3000:3000 --rm --name my_container
deeplinks_resolver:0.1
```

We can also use `-d` to run our containers in detached mode:

```
docker container run -it -p 3000:3000 --rm --name my_container -d
deeplinks_resolver:0.1
```

Our container will continue to run in background. Since we can not make CTRL+C for a container running in background, we can stop it first, then delete:

```
docker container stop 12345
docker container rm 12345
```

We can get log output of a running/stopped container:

```
docker container logs 12345
```

It's also possible to tail logs from a container:

```
docker container logs -f 12345
```

Real time metrics about running containers:

```
docker container stats
```

We can start two containers from the same image, but their names has to be unique and they have to use different ports:

```
docker container run -it -p 3000:3000 --rm --name my_container_1 -d
deeplinks_resolver:0.1
docker container run -it -p 3001:3001 --rm --name my_container_2 -d
deeplinks_resolver:0.1
```

We can make our containers restart if something goes wrong. This is actually useful in production systems:

```
docker container run -it -p 3000:3000 --restart on-failure --name
my_container_1 -d deeplinks_resolver:0.1
```

> We can't use `-rm` and `--restart` flags together.

6.7. Live code loading with volumes

6.8. Debugging tips and tricks

6.9. Linking containers with Docker networks

6.10. Persisting data to your Docker host

6.11. Sharing data between containers

6.12. Optimizing your Docker images

6.13. Running scripts when a container starts

6.14. Cleaning up after yourself