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 retrieve 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

To follow code changes directly in our container we can use volumes. To be able to use volumes with our container `-v` flag comes into play.

```
docker container run -it -p 3001:3001 --rm --name my_container_2 -d -v
"$PWD:/app" deeplinks_resolver:0.1
```

This will take everything from our working directory to the running container. It's also possible to pass multiple volume commands.

6.8. Debugging tips and tricks

Interact with a running container:

```
docker container exec -it my_container sh
```

We can also interact in other ways:

```
docker container exec -it my_container ruby --version # 2.5.5p157
```

We can also create files inside the container:

```
docker container exec -it my_container_1 touch foo.txt
```

The problem is file ownership and permissions. Sometimes docker assigns file ownership as `root:root`. We can pass a user parameter to avoid this problem.

```
docker container exec -it my_container_1 --user "$(id -u):$(id -g)" touch
foo.txt
```

`id -u` will get user ID. `id -g` will get group id.

6.9. Linking containers with Docker networks

Lets create another image:

```
docker image build -t deeplinks_resolver:0.2 .
```

Lets pull redis image:

```
docker pull redis:3.2-alpine
```

There are 2 types of networks:

- Internal networks: LAN
- External networks: WAN

A container can run on any network. We can check networks like this:

```
docker network ls
```

We can inspect any network as follows:

```
docker network inspect bridge
```

Lets start redis container and our deeplinks#resolver container in background:

```
docker container run --rm -itd -p 6379:6379 --name redis redis:3.2-alpine
docker container run --rm -itd -p 3000:3000 --name deeplinks_resolver
deeplinks_resolver:0.2
```

Check IP address of a container:

```
docker exec redis ipconfig
```

But this is a little bit problematic. Because in this case, we have to configure/hardcode IP addresses manually, and if one of the IP addresses change, our containers will break. Docker has a better solution for this. We can configure our own bridge, so our containers can communicate with each other.

```
docker network create --driver bridge first_network
docker network inspect first_network
```

Lets stop our running containers first, and then start them with `--net` flag:

```
docker container run --rm -itd -p 6379:6379 --name redis --net
first_network redis:3.2-alpine
docker container run --rm -itd -p 3000:3000 --name deeplinks_resolver --
net first_network deeplinks_resolver:0.2
```

Inspect the network now:

```
docker network inspect first_network
```

Lets try pinging our redis server from deeplinks_resolver:

```
docker exec deeplinks_resolver ping redis
```

6.10. Persisting data to your Docker host

We will lose data generated by our app when we stop the container. We can create named volumes to persist data, and use container with databases etc.

```
docker volume create deeplinks_resolver_redis
```

List all volumes:

```
docker volume ls
```

Inspect a volume:

```
docker volume inspect deeplinks_resolver_redis
```

We can attach volumes with `-v` flag, when we are starting the container:

```

```

```
docker container run --rm -itd -p 3000:3000 --name deeplinks_resolver --
net first_network -v deeplinks_resolver_redis:/data deeplinks_resolver:0.2
```

`/data` folder isn't always necessary, but by default `redis` image is looking at this folder as a volume. We found this information from `redis` image documentation.

Normally containers supposed to be stateless and portable! Therefore we shouldn't be storing anything in our container. But databases are an exception to this.

6.11. Sharing data between containers

We can expose any folder as a volume like this:

```
VOLUME ["/app/public"]
```

We can also expose any folder with `-v` flag:

```
docker container run --rm -itd -p 3000:3000 --name deeplinks_resolver --
net first_network -v $PWD:/app -v /app/public/
```

Then we can point volumes from other containers:

```
docker container run --rm -itd -p 6379:6379 --name redis --net
first_network -v deeplinks_resolver_redis:/data --volumes-from
deeplinks_resolver redis:3.2-alpine
```

Lets verify the volume:

```
docker container exec -it redis sh
ls /app/public
```

> You can see the changes immediately in volumes!

6.12. Optimizing your Docker images

If we have a `.dockerignore` file exists, during `COPY/ADD` instructions, Docker is going to remove files matching pattern defined in this file. If we have `WORKDIR` set, ignoring will start from this folder. Here is a ready to use script for optimizing `alpine`:

```
FROM python:2.7-alpine

RUN mkdir /app
WORKDIR /app

COPY requirements.txt requirements.txt

RUN apk add --no-cache --virtual .build-deps \
    && pip install -r requirements.txt \
    && find /usr/local \
        \( -type d -a -name test -o -name tests \) \
        -o \( -type f -a -name '*.pyc' -o -name '*.pyo' \) \
        -exec rm -rf '{}' + \
    && runDeps="$( \
        scanelf --needed --nobanner --recursive /usr/local \
                | awk '{ gsub(/,/, "\nso:", $2); print "so:" $2 }' \
                | sort -u \
                | xargs -r apk info --installed \
                | sort -u \
    )" \
    && apk add --virtual .rundeps $runDeps \
    && apk del .build-deps

COPY . .

LABEL maintainer="Nick Janetakis <nick.janetakis@gmail.com>" \
      version="1.0"

VOLUME ["/app/public"]

CMD flask run --host=0.0.0.0 --port=5000
```

6.13. Running scripts when a container starts

`ENTRYPOINT [""]` declaration allows you to run a script, after your docker container starts.

Foe example when we are using postgreSQL, we will probably want to create a database, a database user and password in our container. We can achieve this with `ENTRYPOINT` declaration. Official postgreSQL image already offers an `ENTRYPOINT` for us.

`docker-entrypoint.sh` file:

```
#!/bin/sh

set -e

echo "The Dockerfile ENTRYPOINT has been executed!"
```

Dockerfile declarations to state entry point:

```
# copy entry point
COPY docker-entrypoint.sh /
RUN chmod +x /docker-entrypoint.sh
ENTRYPOINT ["/docker-entrypoint.sh"]
```

Default `ENTRYPOINT` for `alpine` image is `/bin/sh -c`. We are overriding this one by defining an `ENTRYPOINT` in our Dockerfile.

`ENTRYPOINT` allows us to run commands after our container runs. So basically we can use that for:

- Running database migrations
- Changing nginx config

etc. like tasks that has to be run after we start the container.

6.14. Cleaning up after yourself

Show general status of Docker on your system:

```
docker system df
```

Cleanup everything automatically:

```
docker system prune
```

Make it cronjob friendly with `-f` flag. It won't ask for confirmation this way:

```
docker system prune -f
```

We can stop multiple containers:

```
docker container stop aaa bbb ccc
```

Stop all containers together:

```
docker container stop $(docker container ls -a -q)
```