Docker and Kubernetes

Session 2

Building Docker images

Agenda

- Dockerfile
- Build parameter
- Environment variables
- CMD and ENTRYPOINT
- Practice building images

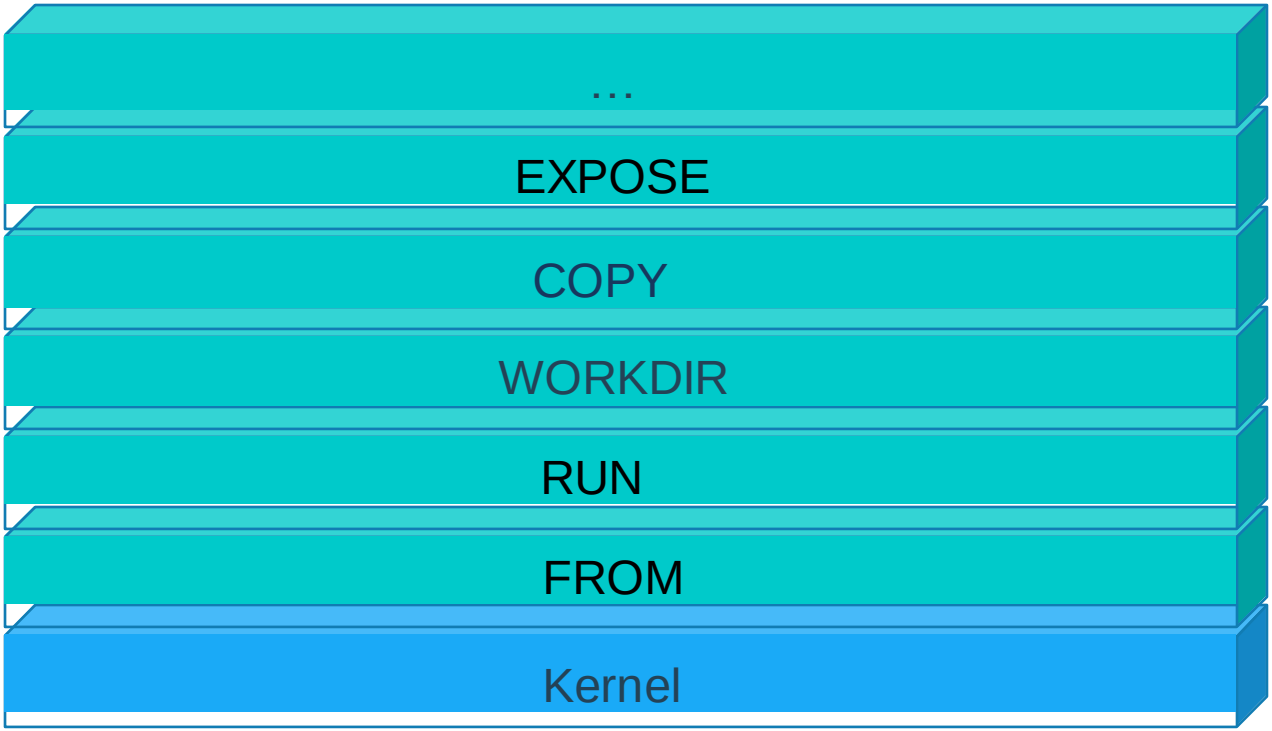
Dockerfile – Linux Example

Dockerfile

```
# Create image based on the official Node 6 image from dockerhub
FROM node:latest

# Create a directory where our app will be placed
RUN mkdir -p /usr/src/app

# Change directory so that our commands run inside this new directory
WORKDIR /usr/src/app

# Copy dependency definitions
COPY package.json /usr/src/app

# Install dependecies
RUN npm install

# Get all the code needed to run the app
COPY . /usr/src/app

# Expose the port the app runs in
EXPOSE 4200

# Serve the app
CMD ["npm", "start"]
```

- Instructions on how to build a Docker image
- Looks very similar to “native” commands
- Important to optimize your Dockerfile

Each Dockerfile Command Creates a Layer



Docker Image Pull: Pulls Layers

```
Alexander@DESKTOP-90ATKET MINGW64 ~/Docker/Demo
$ docker pull nginx:latest
latest: Pulling from library/nginx
bc95e04b23c0: Pull complete
f3186e650f4e: Pull complete
9ac7d6621708: Pull complete
Digest: sha256:b81f317384d7388708a498555c28a7cce778a8f291d90021208b3eba3fe74887
Status: Downloaded newer image for nginx:latest
```

Dockerfile format

Here is the format of the Dockerfile:

```
# Comment
INSTRUCTION arguments
```

The instruction is not case-sensitive. However, convention is for them to be UPPERCASE to distinguish them from arguments more easily.

Docker runs instructions in a Dockerfile in order. A Dockerfile must begin with a FROM instruction. This may be after parser directives, comments, and globally scoped ARGs

Dockerfile format

Here is the format of the Dockerfile:

```
# Comment
INSTRUCTION arguments
```

The instruction is not case-sensitive. However, convention is for them to be UPPERCASE to distinguish them from arguments more easily.

Docker runs instructions in a Dockerfile in order. A Dockerfile must begin with a FROM instruction. This may be after parser directives, comments, and globally scoped ARGs

FROM

The **FROM** instruction initializes a new build stage and sets the Base Image for subsequent instructions. As such, a valid Dockerfile must start with a **FROM** instruction. The image can be any valid image – it is especially easy to start by pulling an image from the Public Repositories.

FROM can appear multiple times within a single Dockerfile to create multiple images or use one build stage as a dependency for another. Simply make a note of the last image ID output by the commit before each new **FROM** instruction. Each **FROM** instruction clears any state created by previous instructions.

Understand how ARG and FROM interact

FROM instructions support variables that are declared by any ARG instructions that occur before the first **FROM**.

```
ARG CODE_VERSION=latest
FROM base:${CODE_VERSION}
CMD /code/run-app
FROM extras:${CODE_VERSION}
CMD /code/run-extras
```

An **ARG** declared before a **FROM** is outside of a build stage, so it can't be used in any instruction after a **FROM**. To use the default value of an **ARG** declared before the first **FROM** use an **ARG** instruction without a value inside of a build stage:

```
ARG VERSION=latest
FROM busybox:$VERSION
ARG VERSION
RUN $VERSION > image_version
```

RUN

RUN has 2 forms:

- **RUN** <command> (shell form, the command is run in a shell, which by default is /bin/sh -c on Linux or cmd /S /C on Windows)
- **RUN** ["executable", "param1", "param2"] (exec form)

The default shell for the shell form can be changed using the SHELL command.

```
RUN ["/bin/bash", "-c", "echo hello"]
```

WORKDIR

```
WORKDIR /path/to/workdir
```

The **WORKDIR** instruction sets the working directory for any **RUN, CMD, ENTRYPOINT, COPY** and **ADD** instructions that follow it in the Dockerfile. If the **WORKDIR** doesn't exist, it will be created even if it's not used in any subsequent Dockerfile instruction.

The WORKDIR instruction can be used multiple times in a Dockerfile. If a relative path is provided, it will be relative to the path of the previous WORKDIR instruction

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

COPY

COPY has two forms:

```
COPY [--chown=<user>:<group>] <src>... <dest>
COPY [--chown=<user>:<group>] ["<src>",... "<dest>"]
```

The COPY instruction copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>.

Multiple <src> resources may be specified but the paths of files and directories will be interpreted as relative to the source of the context of the build.

EXPOSE

The **EXPOSE** instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified.

By default, **EXPOSE** assumes TCP. You can also specify UDP:

```
EXPOSE 80/udp
```

ENV

The **ENV** instruction sets the environment variable <key> to the value <value>.

The value will be interpreted for other environment variables, so quote characters will be removed if they are not escaped. Like command line parsing, quotes and backslashes can be used to include spaces within values

```
ENV MY_NAME="John Doe"
ENV MY_DOG=Rex\ The\ Dog
ENV MY_CAT=fluffy
```

CMD

The **CMD** instruction has three forms:

- CMD ["executable","param1","param2"] (exec form, this is the preferred form)
- CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
- CMD command param1 param2 (shell form)

ENTRYPOINT

An **ENTRYPOINT** allows you to configure a container that will run as an executable.

ENTRYPOINT has two forms:

The exec form, which is the preferred form:

```
ENTRYPOINT ["executable", "param1", "param2"]
```

The shell form:

```
ENTRYPOINT command param1 param2
```

Understand how CMD and ENTRYPOINT interact

The table below shows what command is executed for different **ENTRYPOINT / CMD** combinations:

	No ENTRYPOINT	ENTRYPOINT exec_entry p1_entry	ENTRYPOINT ["exec_entry", "p1_entry"]
No CMD	*error, not allowed*	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD ["exec_cmd", "p1_cmd"]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD ["p1_cmd", "p2_cmd"]	p1_cmd p2_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry p1_cmd p2_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd

Q&A