Dockerfiles

In this lesson, we'll discuss Dockerfiles.

WE'LL COVER THE FOLLOWING

- Overview
- An example for a Dockerfile
- File system layers in the example
- Problem with caching and layers
- Docker multi-stage builds
- Immutable server with Docker
- Docker and tools: Puppet, Chef or Ansible

Overview

The creation of Docker images is done via files named `Dockerfile`. One of Docker's strengths is that Dockerfiles are easy to write and therefore, the rolling out of software can be automated without any problems.

The typical components of a `Dockerfile` are:

- `FROM` defines a base image on which the installation is based. A base image for a microservice usually contains a Linux distribution and basic software, such as the JVM, for example.
- `RUN` defines commands that execute to create the Docker image. In essence, a `Dockerfile` is a shell script that installs the software.
- `CMD` defines what happens when the Docker container is started. Typically, only one process should run in one Docker container. This is started by `CMD`.
- `COPY` copies files in the Docker image. `ADD` does the same; however, it can

also unpack archives and download files from a URL on the Internet. `COPY` is simpler to understand because it does not extract archives, for example. Also, from a security perspective, it can be problematic to download software from the Internet into Docker containers. Therefore, `COPY` should be given preference over `ADD`.

- `EXPOSE` exposes a port of the Docker container. This can then be contacted by other Docker containers or can be tied to a port of the Docker host.

A comprehensive reference is available on the Internet which contains additional details to the commands in `Dockerfile`.

An example for a Dockerfile

A simple example of a `Dockerfile` for a Java microservice looks like this:

```
FROM openjdk:11.0.2-jre-slim
COPY target/customer.jar .
CMD /usr/bin/java -Xmx400m -Xms400m -jar customer.jar
EXPOSE 8080
```

- The first line defines the base image with `FROM`. It is downloaded from the public Docker hub. The image contains a Linux distribution and a Java Virtual Machine (JVM).

- The second line adds a JAR file to the image with `COPY`. A JAR file (Java ARchive) contains all components of a Java application. It has to be available in a sub directory `target` below the directory in which the `Dockerfile` is stored. The JAR file is copied into the root directory of the container.

- The `CMD` entry determines which process should be started when the container is started. In this example, a Java process runs the JAR file.

- Finally, `EXPOSE` makes a port available to the outside. This is the port under which the application is available. `EXPOSE` only means that the container provides the port. It is then available on the internal Docker network. Access from outside is only possible when this is enabled at the start of the container.

The Docker image can be built with the command `docker build --tag=microservice-customer microservice-customer`.

`docker` is the command line tool with which most functionalities of Docker can be controlled. The created Docker image has the tag `microservices-customer` as defined by the `--tag` parameter.

The `Dockerfile` has to be in the sub directory `microservice-customer`. The name of this directory is the second parameter.

File system layers in the example

The first image in this lesson shows that a Docker image consists of multiple layers.

Although no layers have been defined in `Dockerfile`, the image `microservices-customer` contains multiple layers. Each line of the `Dockerfile` defines a new layer. These layers are reused. Thus, if `docker build` is called again, Docker will go through the `Dockerfile` again. However, it will find that all actions in `Dockerfile` have already been executed once. As a result, nothing happens.

If the `Dockerfile` was modified in such a way that, after the `COPY`, another line with a `COPY` of another file is inserted, Docker would reuse the existing layer with the first `COPY`, but the second `COPY` and all further lines would then create new layers.

In this manner, Docker only re-creates the layers that need to be rebuilt. This not only saves storage space but is also much faster.

Problem with caching and layers

A `Dockerfile` for obtaining a Ubuntu installation with updates looks like this:

```
FROM ubuntu:15.04
RUN apt-get update ; apt-get dist-upgrade -y -qq
```

First, a Ubuntu base image is loaded from the public Docker hub on the Internet. The commands `apt-get update` and `apt-get dist-upgrade -y -qq` are used to update the package index and then install all packages with updates. The options ensure that `apt-get` does not ask the user for permission and

outputs only a few messages on the console.

The two commands are separated in the line by a `;`. This causes them to be executed one after the other. A new file system layer is created only after both commands have been executed. This is useful for creating more compact images with fewer layers.

However, this `Dockerfile` also has a problem. If the Docker image is built again, no current updates will be downloaded. Instead, nothing happens because the images are already there.

Layer caching is based only on commands. Docker does not recognize that the external package index has changed. To ignore the existing images and force the rebuilding of the images, the parameter `--no-cache=true` can be passed to `docker build`.

Docker multi-stage builds

Everything needed to build a Docker image can also be found in the Docker image. Therefore, all of it is available at runtime of the Docker container.

If code is being compiled in a `Dockerfile`, the compiler is also available at runtime. This is unnecessary and can even be a security problem. If the container is compromised, the attacker can compile code inside the container with the original tools, which might allow more attacks.

It is not easy to delete all of the build environment because today there is usually a complex tool chain for building software.

Building the software outside of Docker might also not be an option. Docker is based on Linux; therefore, on macOS, you would need to run a cross compiler to generate Linux binaries.

To solve this problem, Docker has **Multi Stage Builds**. They make it possible to compile the program in one phase of the build in a Docker image, and to transfer only the compiled program to the next phase into a different Docker image.

Afterwards, the build tools are no longer available at runtime. They do not have to be deleted, and they do not have to be installed on the host machine.

This lesson in the next chapter shows a Docker Multi Stage Build using a Go program as example.

Immutable server with Docker

Immutable server is an idea that predates Docker. The idea of an immutable server is that **a server will never be changed**; therefore, the software on the server will never be updated or modified. The server will always be completely rebuilt from scratch.

In this way, the state of the server can be reconstructed cleanly. For each server, an installation script installs all the needed software on a basic OS image.

However, **immutable servers are hard to implement**. It is very cumbersome to completely reinstall a server. The process might take minutes or hours. That is far too much time compared to, for example, just changing a configuration file.

This is exactly where **Docker helps**. Because of the optimizations, only the necessary steps are taken so that *immutable servers* can also be an option from this perspective.

A `Dockerfile` describes how to create a Docker image starting from a base image. With each build, it will seem as if the complete Docker image is being created. Behind the scenes, however, optimizations ensure that only what is really necessary is built.

For example, if in the very last step a configuration file is added and just that configuration file has been modified, Docker is smart enough to reuse the results of all other installation steps and just add the new configuration file. This just takes a few seconds.

Docker is conceptually as clear as an immutable server but much more efficient for actually implementing them.

Docker and tools: Puppet, Chef or Ansible

Besides immutable servers, there are other ways to handle the installation of software.

Idempotent installation means that an installation script provides the same results no matter how often it runs. For an idempotent installation, there are no steps like “install the Java package,” but rather a definition of the desired state: “ensure that the Java package is installed”. If the installation is run on a fresh OS, Java would be installed. If the installation is run on a system that already has Java installed, nothing happens.

Idempotent installation is particularly **useful to enable updates**. For each update, the script would check if all software is installed in the correct version. If that is not the case, the correct version is installed. Tools such as **Puppet, Chef, or Ansible** support the concept of idempotent installation.

A `Dockerfile` is a very easy way to install software. The use of tools such as Puppet, Chef, or Ansible for the installation of software in a Docker image is possible but does not make a lot of sense. In particular, it is not necessary to use the update functionalities of these tools, because the image is typically freshly built with the *immutable server* approach.

This approach is easier than writing Puppet, Chef or Ansible scripts because defining the desired state is usually quite complex. The `Dockerfile` only describes how to build an image, whereas the other tools must also enable updates of the servers and are therefore more difficult to use.

QUIZ

1 Which of the following best describes how the `COPY` command works?

In the next lesson, we'll study Docker Compose.