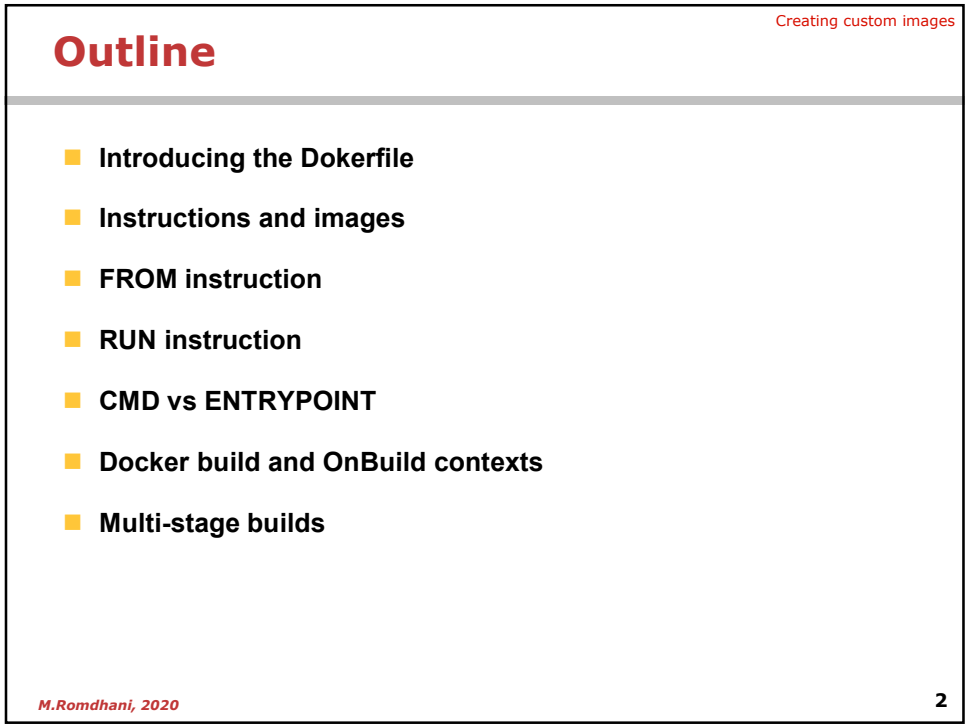

docker
Unit 3
Creating Custom Images
Business
Training

1


Creating custom images
Outline
Introducing the Dokerfile
Instructions and images
FROM instruction
RUN instruction
CMD vs ENTRYPOINT
Docker build and OnBuild contexts
Multi-stage builds
M.Romdhani, 2020
2

2

Introducing the Dokerfile

3

Dockerfile overview

- A Dockerfile is a build recipe for a Docker image.
- It contains a series of instructions telling Docker how an image is constructed.
- The docker build command builds an image from a Dockerfile.

4

Writing our first Dockerfile

- **Our Dockerfile must be in a new, empty directory.**
  - Create a directory to hold our Dockerfile.
- **Create a Dockerfile inside this directory.**
  - Of course, you can use any other editor of your choice
  - Type this into our Dockerfile...

```
FROM ubuntu
RUN apt-get -y update
RUN apt-get install figlet
```

    - **FROM** indicates the base image for our build.
    - Each **RUN** line will be executed by a new container during the build.
- **Build it ...**

```
$ docker build -t figlet .
```

  - **-t** indicates the tag to apply to the image.
  - **.** indicates the location of the build context.

5

What happens when we build the image?

- **The output of docker build looks like this:**

```
$ docker build -t figlet .
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM ubuntu
 ---> f975c5035748
Step 2/3 : RUN apt-get update
 ---> Running in e01b294dbffd
(...output of the RUN command...)
Removing intermediate container e01b294dbffd
 ---> eb8d9b561b37
Step 3/3 : RUN apt-get install figlet
 ---> Running in c29230d70f9b
(...output of the RUN command...)
Removing intermediate container c29230d70f9b
 ---> 0dfd7a253f21
Successfully built 0dfd7a253f21
Successfully tagged figlet:latest
```

  - The output of the RUN commands has been omitted.
  - Let's explain what this output means

6

Sending the build context to Docker

```
Sending build context to Docker daemon 2.048 kB
```

- **The build context is the . directory given to docker build.**
- **It is sent (as an archive) by the Docker client to the Docker daemon.**
- **This allows to use a remote machine to build using local files.**
- **Be careful (or patient) if that directory is big and your link is slow.**
- **You can speed up the process with a .dockerignore file**
  - It tells docker to ignore specific files in the directory
  - Only ignore files that you won't need in the build context!

7

.dockerignore

- **Place .dockerignore in the root of build context with list of file/directory patterns to be excluded from build context**
- **Very much like .gitignore**
- **Helpful when you want to avoid sending heavy libraries to the Docker daemon build context and want to selectively ignore files**
  - Exclude **node_modules** when building an Angular Application.

8

Creating custom images

Executing each step

```
Step 2/3 : RUN apt-get update
 ---> Running in e01b294dbffd
(...output of the RUN command...)
Removing intermediate container e01b294dbffd
 ---> eb8d9b561b37
```

- A container (e01b294dbffd) is created from the base image.
- The RUN command is executed in this container.
- The container is committed into an image (eb8d9b561b37).
- The build container (e01b294dbffd) is removed.
- The output of this step will be the base image for the next one.

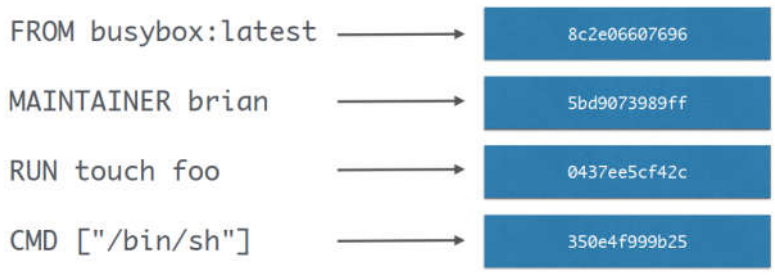
M.Romdhani, 2020 9

9

Creating custom images

Image Layers

- Each instruction creates a new Layer



M.Romdhani, 2020 10

10

The caching system

- **If you run the same build again, it will be instantaneous. Why?**
- **After each build step, Docker takes a snapshot of the resulting image.**
- **Before executing a step, Docker checks if it has already built the same sequence.**
- **Docker uses the exact strings defined in your Dockerfile, so:**
  - `RUN apt-get install figlet cowsay`
    is different from
    `RUN apt-get install cowsay figlet`
  - `RUN apt-get update` is not re-executed when the mirrors are updated

You can force a rebuild with `docker build --no-cache ....`

11

Instructions and images

12

docker build

- **The format of the docker build command is:**

```
$ docker build [options] path|URL|-
```

- **The config options available for the docker build command are**

Options	Description
-f, --file=Path/Dockerfile	File to use as Dockerfile for build
--force-rm=false	Force removal of intermediate build containers
--no-cache=false	Don't make use of the build cache
--pull=false	Pull newer version of base image if one exists
-q, --quiet=false	Build the image very quietly
--rm=true	Remove intermediate containers after successful build
-t, --tag=[]	Provide image name and tag(s) for newly built image
--build-arg=[]	Supply variables for the build process

docker build example

- **This Dockerfile for creating a simple lighttpd http server:**

```
FROM alpine:edge

RUN apk update \
    && apk add lighttpd \
    && rm -rf /var/cache/apk/*

ADD htdocs /var/www/localhost/htdocs

EXPOSE 80

ENTRYPOINT ["lighttpd", "-D", "-f", "/etc/lighttpd/lighttpd.conf"]
```

- **The image can be built with the following command:**

```
$ docker build -t my-lighttpd .
```

- **Note:**
  - The build has the current directory as context
  - All paths are relative to the Dockerfile
  - Each command in the Dockerfile creates a new (temporary container)
  - Every creation step is cached, so repeated builds are fast

Dockerfile instructions

Instruction	Description
FROM	Parent image
MAINTAINER (Deprecated in v19, Replaced by LABEL maintainer)	Specify the image maintainer
ARG	Parameters for contructing the image
ENV	Specify Environnement variables
LABEL	Specify Label meta-data
VOLUME	Mount volumes
RUN	Run a command
COPY	Copy files to the image
ADD	Add files to the image
WORKDIR	Specify the working directory
EXPOSE	Expose ports to be accessed
USER	User name or UID to be used
ONBUILD	Instructions to execute when constructing child images
CMD	Command to execute when starting a container
ENTRYPOINT	The default entry point of the container

15

FROM instruction

16

FROM Instruction

- **The format of the FROM instruction is:**

```
FROM repository[:tag|@digest]
```

  - The FROM instruction must be the first instruction in the Dockerfile, aside from comments
  - Specifies the image from which to base the build of the new image
  - The base image must be fully qualified, and may also specify a tag or an image digest hash
  - The word scratch is a reserved word, and `FROM scratch` results in a no-op instruction with no new image layer

FROM Instruction Best practices

- **Use a Smaller Image Base**
  - You should always opt for smaller images. Images that share layers and are smaller in size are quicker to transfer and deploy.
  - Examples:
    - BusyBox (1M), Alpine (5M) are smaller that ubuntu (60M)
    - If you need a JDK, consider basing your image on the official **openjdk** image, rather than starting with a generic ubuntu image and installing openjdk as part of the Dockerfile.

MAINTAINER Instruction

- **The format of the MAINTAINER instruction is:**

```
MAINTAINER name
```

  - The MAINTAINER instruction sets the author attribute for the built Docker image
  - Use of the MAINTAINER instruction is useful for providing a point of reference for an image's adopters
  - The author attribute can be determined with a docker inspect query of the built image

19

LABEL Instruction

- **The format of the LABEL instruction is:**

```
LABEL key=value ...
```

  - The LABEL instruction enables an image builder to attach metadata to an image
  - Labels can be used for annotating images, which can be used in image search and selection
  - The docker images CLI command can filter its output based on an image's label metadata

20

RUN instruction

21

RUN Instruction

- **The formats of the RUN instruction are:**

```
RUN command                          # Shell Form
RUN ["prog", "arg1", "arg2", ...]    # Exec Form
```

  - The RUN instruction executes a command in a container, and the **diff** becomes a new image layer
  - The shell form uses `'sh -c'` to execute the command
  - Commands can be daisy-chained using `'&&'` for a single RUN instruction, thereby **optimising** image layers
  - A **JSON array** is used for the parameters when using the exec form, which can be used when 'sh' is missing

22

RUN Instruction Best practices

- **Since Docker 1.10 the COPY, ADD and RUN statements add a new layer to your image.**
- **Avoid Adding Unnecessary Layers to Reduce Docker Image Size**
  - You can do this by consolidating multiple commands into a single RUN line and using your shell's mechanisms to combine them together. Consider the following two fragments.
    - This fragment creates two layers in the image

```
RUN apt-get -y update
RUN apt-get install -y python
```

    - While this one create only creates one layer

```
RUN apt-get -y update && apt-get install -y python
```


ENV Instruction

- **The formats of the ENV instruction are:**

```
ENV key value              # Single key/value definition
ENV key=value ...          # Multiple key/value definitions
```

  - The ENV instruction **sets key/value pairs** for use by all **subsequent** Dockerfile instructions
  - The first form sets a **single** key/value pair, whilst the second form sets **multiple** key/value pairs
  - Certain Dockerfile instructions can make use of the key/value pairs using shell-like variable substitution
  - Key value pairs **persist** into any container that is derived from the image, as **environment variables**

ARG Instruction

- **The format of the ARG instruction is:**

```
ARG key[=value]
```

  - The ARG instruction sets a key/value pair at **build time**
  - Works in conjunction with the `--build-arg` config option
  - Key/value pairs defined by the ARG instruction **do not persist**, and are not available to containers
  - If an ARG instruction and an ENV instruction reference the same key, the value defined by the **ENV instruction overrides**
  - ARG and ENV instructions can work in conjunction to enable dynamic key/value pair **persistence** in images

25

EXPOSE Instruction

- **The format of the EXPOSE instruction is:**

```
EXPOSE port ...
```

  - The **EXPOSE** instruction designates a port(s) on which to expose a derived container's **service. EXPOSE is there for documentation purposes**
  - Ports defined with the EXPOSE instruction can be forwarded to random host ports with the **-P** CLI option
  - The EXPOSE instruction is not required for network-connected containers to consume each other's services

26

VOLUME Instruction

- **The formats of the VOLUME instruction are:**

```
VOLUME path ...
VOLUME ["path1", "path2", ...]
```

  - The VOLUME instruction specifies a **mount point** which is used to mount a host directory as a **volume**
  - The VOLUME instruction is a **deterministic** means of creating container volumes
  - Any data located at the designated path in the image **is copied to the volume** on the host
  - Any **amendments to data residing** within the image location **after** the VOLUME instruction, will **be lost**

STOPSIGNAL Instruction

- **The format of the STOPSIGNAL instruction is:**

```
STOPSIGNAL signal
```

  - The STOPSIGNAL instruction provides Docker with the appropriate signal to send to cause a container to exit
  - By, default docker stop sends the SIGTERM signal to the container's process
  - An application may be programmed to handle a different signal to effect a graceful exit
  - The STOPSIGNAL instruction provides flexibility to enable graceful termination

Creating custom images

WORKDIR Instruction

- **The format of the WORKDIR instruction is:**

```
WORKDIR path
```

  - The WORKDIR instruction changes the filesystem context for subsequent Dockerfile instructions
  - It can be used multiple times within a Dockerfile, and can be an absolute or relative path
  - The WORKDIR instruction is used in conjunction with ADD, COPY and RUN instructions during image build
  - Also used in conjunction with CMD and ENTRYPOINT instructions to set the container's working directory

29

Creating custom images

USER Instruction

- **The format of the USER instruction is:**

```
USER username|uid
```

  - The USER instruction sets the username or UID for subsequent RUN, CMD and ENTRYPOINT instructions
  - It can be used multiple times within a Dockerfile
  - Container processes run as UID 0 unless the USER instruction is employed to change the default user
  - The USER instruction is often used after a RUN instruction has added a user to /etc/passwd

30

COPY Instruction

- **The formats of the COPY instruction are:**

```
COPY src ... dest
COPY ["src"], ... ["dest"]
```

  - The COPY instruction copies files and directories into the Docker image
  - The source elements must be relative to, and within, the build context
  - The single destination must be an absolute path, or relative to the working dir or /
  - The source elements can contain wildcard patterns as specified by Go filepath.Match rules

31

ADD Instruction

- **The formats of the ADD instruction are:**

```
ADD src ... dest
ADD ["src"], ... ["dest"]
```

  - The ADD instruction copies files and directories into the Docker image
  - It can also be used to add remote files to the image
  - Wherever possible, the COPY instruction should be used ahead of the ADD instruction
  - If the source for the ADD instruction is a local tar archive, the ADD instruction will extract it to the destination

32

COPY vs ADD

- COPY and ADD serve similar purposes. They let you copy files from a specific location into a Docker image.
- COPY takes in a src and destination. It only lets you copy in a local file or directory from your host (the machine building the Docker image) into the Docker image itself.
- ADD lets you do that too, but it also supports 2 other sources. First, you can use a URL instead of a local file / directory. Secondly, you can extract a tar file from the source directly into the destination.

33

HEALTHCHECK

- AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

34

CMD vs ENTRYPOINT

35

CMD Instruction

- **The formats of the CMD instruction are:**

```
CMD ["arg1", "arg2", ...]
CMD ["prog", "arg1", ...]
CMD prog arg1 ...
```

  - The CMD instruction is used to provide default execution parameters for a container
  - The CMD instruction in a Dockerfile can be overridden at runtime by the Docker CLI
  - A Dockerfile should contain only one CMD instruction
  - It is generally recommended that the exec form is used instead of the shell form

36

ENTRYPOINT Instruction

- **The formats of the ENTRYPOINT instruction are:**

```
ENTRYPOINT [“prog”, “arg1”, ...]
ENTRYPOINT prog arg1 ...
```

  - The ENTRYPOINT instruction is used to run a container just as if it were an executable binary
  - CLI arguments, or the contents of a CMD instruction, are appended to the container's entrypoint
  - All arguments provided via the CLI or CMD instruction, are ignored with the use of the shell form
  - The CLI config option `--entrypoint` can be used to override the entrypoint, invoking the exec format

37

CMD vs ENTRYPOINT

- **CMD defines a default command to run when none is given.**
  - It can appear at any point in the file.
  - Each CMD will replace and override the previous one.
  - As a result, while you can have multiple CMD lines, it is useless
  - Example:

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
CMD figlet -f script hello  # -f defines the font
```

- **Overriding CMD**
  - If we want to get a shell into our container (instead of running figlet), we just have to specify a different program to run:

```
$ docker run -it figlet bash
root@7ac86a641116:/#
```

  - We specified bash. It replaced the value of CMD.

38

CMD vs ENTRYPOINT

- **ENTRYPOINT looks similar to CMD, because it also allows you to specify a command with parameters.**
  - The difference is ENTRYPOINT command and parameters **are not ignored** when Docker container runs with command line parameters.
- **ENTRYPOINT defines a base command (and its parameters) for the container.**
  - The command line arguments are appended to those parameters.
  - Like CMD, ENTRYPOINT can appear anywhere, and replaces the previous value.
- **Using CMD and ENTRYPOINT together**
  - What if we want to define a default message for our container?
  - Then we will use ENTRYPOINT and CMD together.
    - ENTRYPOINT will define the **base command** for our container.
    - CMD will define **the default parameter(s)** for this command.

CMD and ENTRYPOINT Together

- **Our new Dockerfile will look like this:**

```
FROM ubuntu
RUN apt-get update
RUN ["apt-get", "install", "figlet"]
ENTRYPOINT ["figlet", "-f", "script"]
CMD ["hello world"]
```

  - ENTRYPOINT defines a base command (and its parameters) for the container.
  - If we don't specify extra command-line arguments when starting the container, the value of CMD is appended.
  - Otherwise, our extra command-line arguments are used instead of CMD.

Docker build and OnBuild contexts

41

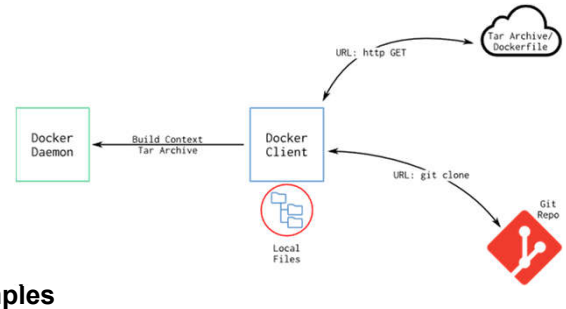
Understanding Build Context

- **To generate an image, the Docker server needs to access the application's Dockerfile, source code, and any other files that are referenced in the Dockerfile itself.**
  - That build context (by default) is the entire directory the Dockerfile is in.
- **Things to remember about the build context:**
  - Files inside the build context are the only files readable by the instructions specified in the Dockerfile.
  - Any **symlinks** that point to external locations will **not be resolved**.
  - If a .**dockerignore** file is specified at the root of the build context, it can be used to exclude files from the build context by adding filtering rules

42

Understanding Build Context

- **Build context sent to daemon as tar archive**
  - That build context (by default) is the entire directory the Dockerfile is in



- **Examples**
  - Git Contexts

```
$ docker build https://github.com/docker/rootfs.git#container:docker
```

  - Tarbal contexts

```
$ docker build http://server/context.tar.gz
```


43

HEALTCHECK

- **The HEALTHCHECK instruction tells Docker how to test a container to check that it is still working.**
  - This can detect cases such as a web server that is stuck in an infinite loop and unable to handle new connections, even though the server process is still running.

HEALTHCHECK [OPTIONS] CMD command (check container health by running a command inside the container)

When a container has a healthcheck specified, it has a health status in addition to its normal status.

This can detect cases such as a web server that is stuck in an infinite loop and unable to handle new connections, even though the server process is still running.

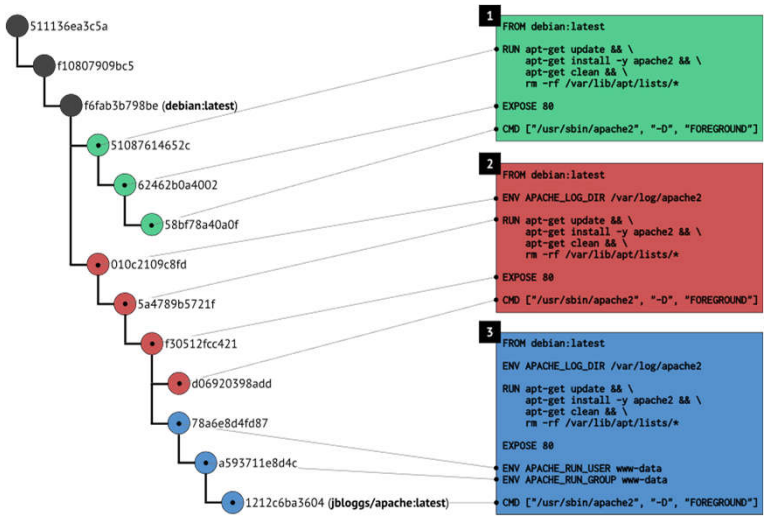
Example.

```
HEALTHCHECK --interval=5m --timeout=3s \
   CMD curl -f http://localhost/ || exit 1
```


44

Creating custom images

The build cache



45

Multi stage builds

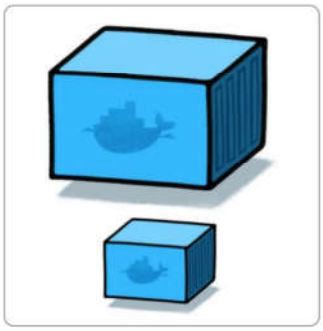
46

What are multi stage builds ?

- **Multi-stage builds are a method of organizing a Dockerfile to minimize the size of the final container.**
  - This is made possible by the image building process into multiple stages
  - Each stage is a separate image, and can copy files from previous stages.

```
# Dockerfile
# build stage
FROM buildbase as build
...
...
...

# production ready stage
FROM runbase
...
COPY --from=build
/artifact /app
```



47

Multi-stage builds

- **At any point in our Dockerfile, we can add a new `FROM` line.**
- **This line starts a new stage of our build.**
- **Each stage can access the files of the previous stages with `COPY --from=....`**
- **When a build is tagged (with `docker build -t ...`), the last stage is tagged.**
- **Previous stages are not discarded: they will be used for caching, and can be referenced.**

48

Multi-stage builds in practice

- Each stage is numbered, starting at 0
- We can copy a file from a previous stage by indicating its number, e.g.:

```
COPY --from=0 /file/from/first/stage /location/in/current/stage
```

- We can also name stages, and reference these names:

```
FROM golang AS builder
RUN ...
FROM alpine
COPY --from=builder /go/bin/mylittlebinary /usr/local/bin/
```


Multi-stage builds in practice

- Building a Java Spring Boot Application in a single image

```
FROM maven:3.5.2-jdk-9
COPY src /usr/src/app/src
COPY pom.xml /usr/src/app
RUN mvn -f /usr/src/app/pom.xml clean package

EXPOSE 8080
ENTRYPOINT ["java","-jar","/usr/src/app/target/myapp-1.0.0-SNAPSHOT.jar"]
```

- Building the Java Spring Boot Application using a multi-stage build

```
FROM maven:3.5.2-jdk-9 AS build
COPY src /usr/src/app/src
COPY pom.xml /usr/src/app
RUN mvn -f /usr/src/app/pom.xml clean package

FROM openjdk:9
COPY --from=build /usr/src/app/target/ myapp-1.0.0-SNAPSHOT.jar
/usr/app/myapp-1.0.0-SNAPSHOT.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","/usr/app/myapp-1.0.0-SNAPSHOT.jar"]
```