KOÇ UNIVERSITY

COMP 350
Introduction to DevOps

Lecture 7
Dockerfile

Hakan Ayral

Different ways to create a Docker image

- You can create a new docker image in two ways:
  - You can create a container from an available OS base image, work inside it manually (*i.e. connect to a shell inside the container*) and do some changes (*manually add files, install new software packages, etc.*) then save the last state as a new image from which new containers can be created.

or

  - You can create a **Docker file** which is a text file that tells Docker to start from a specific image and at each line gives the Docker commands needed to build the new image.

How to connect to the shell of an already running container?

- One way to attach to a running Docker container is by using the **docker attach** command. This command will attach the container's standard input, output, and error streams to your local terminal.
    1. find out the container id first (use **docker ps**)
    2. then run **docker attach <container-id>** in the terminal and be greeted with a shell prompt, log output, or… nothing!
    - Attaching to an already-running container will only be valid if it's running a shell as its entry point. Most container images will run a different executable, and attaching to such a container will usually result in being able to see the container's output – not getting access to an interactive shell.

How to connect to the shell of an already running container?

- Other way, using **docker exec** is a more successful approach with most images. With this you can start a new process in the container and interact with it. (*Additionally, **docker exec** is a bit easier as you can use the container name instead of the container id.*)
- If you know a specific shell, like **sh** or **bash** is available in the container, you can connect to it with a single line. (*Note you'll need to specify the **-it** switches to make sure you can interact with the running process in the container*)

```
docker exec -it <container-name> /bin/bash
```

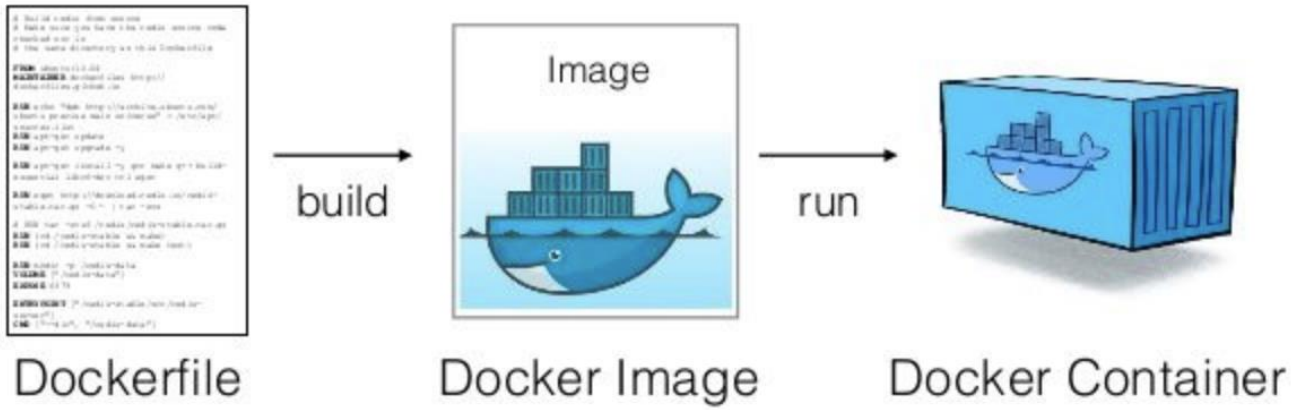
-ti = -t + -i = --tty + --interactive
-i Keep STDIN open even if not attached
-t Allocate a pseudo-tty

- You can use **docker exec** to run other commands as well. For example, if you want to run **tail** to look at the latest entries in a given file (*or run any other command*) you don't need to use an interactive shell. Instead, you can run it with **docker exec** directly:

```
docker exec <container-name> tail /var/log/messages
```

What is a Dockerfile?

- A **Dockerfile** is a text file that contains a set of user-defined instructions on how to build a Docker image.
- When a Dockerfile is called with the **`docker image build`** command, it is used to assemble a container image.



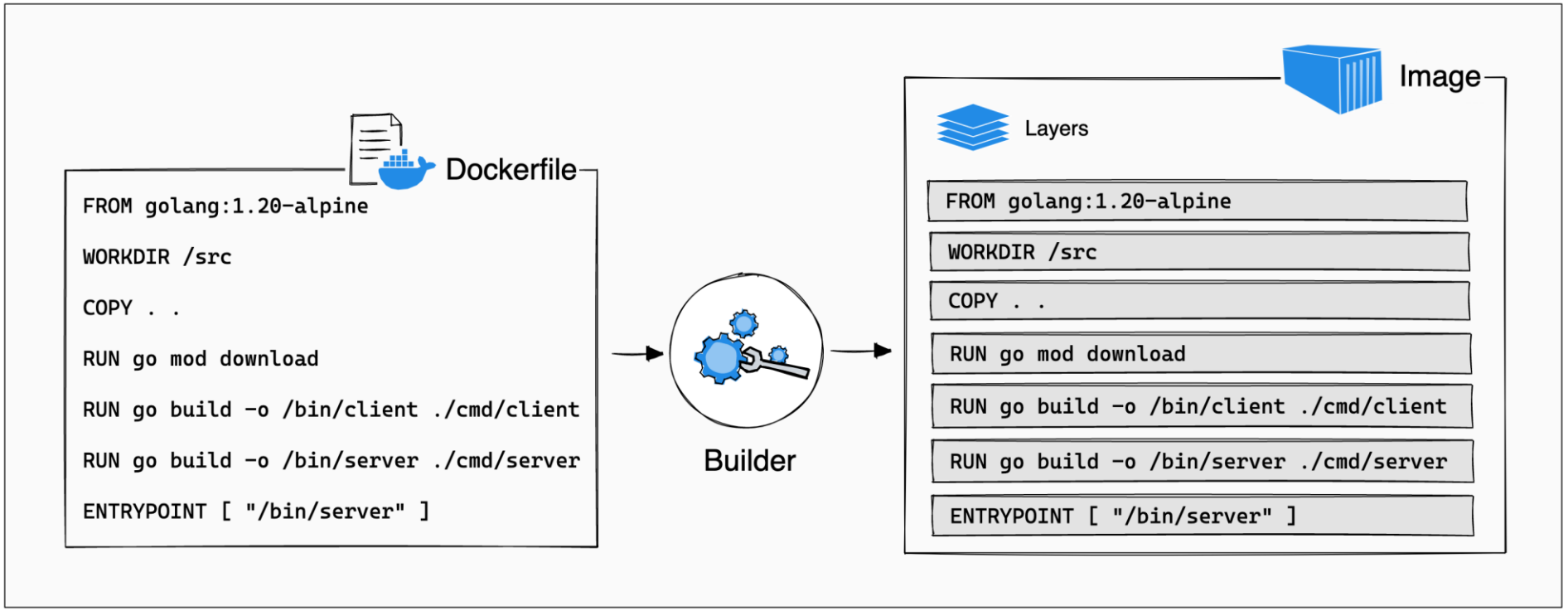
What is a Dockerfile?

- Here is a sample Dockerfile:

```
FROM ubuntu
LABEL maintainer=hayral@ku.edu.tr
LABEL description=”This Dockerfile installs nodejs on ubuntu.”
RUN apt-get update
RUN apt-get install -y nodejs
COPY . /app
WORKDIR /app
EXPOSE 80
CMD ["node","index.js"]
```

Dockerfile lines and image layers

- Each line in the dockerfile correspond to a new layer on the image:



Dockerfile Commands

- Let’s look at some Dockerfile commands:

Command	Description
FROM	Sets the base image for subsequent changes
RUN	Execute commands in a new layer on top of the current image and commit the results
CMD	The command line to be run when a container is launched using this image
LABEL	Adds metadata to an image
EXPOSE	Makes some ports inside the container reachable from outside using port forwarding
ENV	Sets a shell environment variable
ADD	Copy new files, directories, or remote file URLs from outside (i.e. host or internet) into the filesystem of the container
COPY	Copy new files or directories into the filesystem of the container
ENTRYPOINT	Specify default executable.
VOLUME	Creates a mount point and marks it as a externally mounted volume from host (allows persistence of data between launches)
WORKDIR	Sets the working directory for any RUN, CMD, ENTRYPOINT, COPY, and ADD commands
SHELL	Set the default shell of an image.

FROM

- **Usage**
  - **FROM <image>**
  - **FROM <image>:<tag>**
  - **FROM <image>@<digest>**

- **FROM** instruction tells Docker which base image you would like to use as a starting point for your image.
- **FROM** must be the first non-comment instruction in the Dockerfile.
- The tag or digest values are optional.
  - If you omit either of them, the builder assumes a latest by default.
- In the sample docker file we were using Alpine Linux, so we simply have to state the name of the image (and the tag if needed) we wish to use.
- To use the latest official Alpine Linux image, you need to add **`alpine:latest`**

LABEL

- **Usage:**
  - **LABEL <key>=<value> [<key>=<value> ...]**

- **LABEL** instruction adds metadata (i.e. axtra information and comments) to an image.
  - This information can be anything from a version number to a description.
- Labels are additive including LABELs in FROM images (your new image will have all the labels from the base image plus those you added)
- It's also recommended that you limit the number of labels you use.
- You can view the labels' of your images with **`docker image inspect`** command

RUN

- **Usage:**
  - **RUN <command>**
    - This is the shell form, the command is run in a shell, which by default is /bin/sh -c on Linux.
  - **RUN ["<executable>", "<param1>", "<param2>"]**
    - This is the exec form.
- **RUN** instruction allows us to install software and run scripts, commands, and other tasks.
- The exec form makes it possible to avoid shell string munging, and to RUN commands using a base image that does not contain the specified shell executable.
- The default shell for the shell form can be changed using the SHELL command.

COPY and ADD

- **Usage:**
  - **COPY <src> [<src> ...] <dest>**
  - **COPY ["<src>", ... "<dest>"]**
    - this form is required for paths containing whitespace

- Copies new files or directories from <src> and adds them to the filesystem of the image at the path <dest>.
  - <src> may contain wildcards.
  - <src> must be relative to the source directory that is being built (the context of the build).
  - <dest> is an absolute path, or a path relative to WORKDIR.
  - If <dest> doesn't exist, it is created along with all missing directories in its path.
- **COPY** and **ADD** look like they are doing the same task in that they are both used to transfer files to the image. However, there are some differences between these two commands:
  - **COPY** Copies new files or directories from <src> and adds them to the filesystem of the image at the path <dest>.
  - **ADD** Copies new files, directories, or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>; it also automatically uncompresses any compressed files added.

WORKDIR

- **Usage:**
  - **WORKDIR </path/to/workdir>**

- Sets the working directory for any RUN, CMD, ENTRYPOINT, COPY, and ADD instructions that follow it.
- It can be used multiple times in the one Dockerfile.
- If a relative path is provided, it will be relative to the path of the previous WORKDIR instruction.

EXPOSE

- **Usage:**
  - **EXPOSE <port> [<port> ...]**

- Informs Docker that the container listens on the specified network port(s) at runtime.
- Allows us to make a network port inside the container to be available to outside.

VOLUME

- **Usage:**
  - **VOLUME ["<path>", ...]**
  - **VOLUME <path> [<path> ...]**

- Creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.
- Files in a container are read/only, any new files created or any changes are stored temporarily abd they are lost when a new container is launched form the same image.
  - the only way to retain information across different container instances from the same image is to mount a volume to the container which maps an outside storage area to the file system inside the container.
  - Any files written to that mount point will not be lost when the container stops and it will be available if a new container is launched from the same image with same Volume mounting.

ENV

Docker supports environment variables as a practical way of externalizing a containerized app configuration.

When included in a Docker image, environment variables become available to app containers created based on the image.

- **Usage:**
  - **ENV <key> <value>**
  - **ENV <key>=<value> [<key>=<value> ...]**

- The ENV instruction sets the environment variable <key> to the value <value>.
- The value will be in the environment of all “descendant” Dockerfile commands and can be replaced inline as well.
- The environment variables set using ENV will persist when a container is run from the resulting image.
- The first form will set a single variable to a value with the entire string after the first space being treated as the <value> - including characters such as spaces and quotes.

Types of Docker Environment Variables

- A Dockerfile, supports two environment variable types:
  - **ARG** variables usually store important high-level configuration parameters, such as the version of the OS or a library.
    - They are build-time variables, i.e., their only purpose is to assist in building a Docker image.
    - Containers do not inherit **ARG** variables from images.
    - However, users can later retrieve **ARG** values from an image with the docker history command.
  - **ENV** variables store values such as secrets, API keys, and database URLs.
    - Unlike **ARGs**, they persist inside the image and the containers created from that template.
    - Users can override **ENV** values in the command line or provide new values in an **ENV** file.

Tips for building Docker Images Build Time

- Tip #1: Order matters for caching
  - The order of the build steps (Dockerfile instructions) matters, because when a step's cache is invalidated by changing files or modifying lines in the Dockerfile, subsequent steps of their cache will break. Order your steps from least to most frequently changing steps to optimize caching.

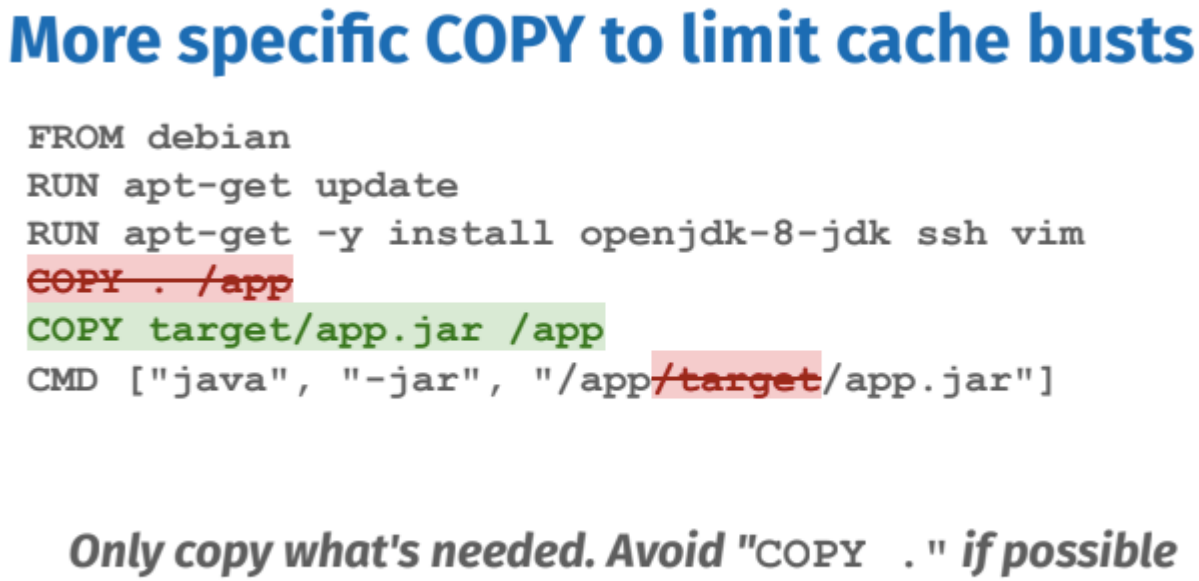
Order matters for caching

```
FROM debian
COPY . /app
RUN apt-get update
RUN apt-get -y install openjdk-8-jdk ssh vim
COPY . /app
CMD ["java", "-jar", "/app/target/app.jar"]
```

Order from least to most frequently changing content.

Tips for building Docker Images Build Time

- Tip #2: More specific COPY to limit cache busts
  - Only copy what's needed. If possible, avoid "COPY ." When copying files into your image, make sure you are very specific about what you want to copy. Any changes to the files being copied will break the cache. In the example above, only the pre-built jar application is needed inside the image, so only copy that. That way unrelated file changes will not affect the cache.



Tips for building Docker Images Build Time

- Tip #3: Identify cacheable units such as apt-get update & install
  - Each RUN instruction can be seen as a cacheable unit of execution. Too many of them can be unnecessary, while chaining all commands into one RUN instruction can bust the cache easily, hurting the development cycle. When installing packages from package managers, you always want to update the index and install packages in the same RUN: they form together one cacheable unit. Otherwise you risk installing outdated packages.

Line buddies: apt-get update & install

```
FROM debian
RUN apt-get update
RUN apt-get -y install openjdk-8-jdk ssh vim
RUN apt-get update \
 && apt-get -y install \
    openjdk-8-jdk ssh vim
COPY target/app.jar /app
CMD ["java", "-jar", "/app/app.jar"]
```

Prevents using an outdated package cache

Tips for building Docker Images
Reduce Image Size

- Tip #4: Remove unnecessary dependencies
  - Remove unnecessary dependencies and do not install debugging tools. If needed debugging tools can always be installed later. Certain package managers such as apt, automatically install packages that are recommended by the user-specified package, unnecessarily increasing the footprint. Apt has the –no-install-recommends flag which ensures that dependencies that were not actually needed are not installed. If they are needed, add them explicitly.

Remove unnecessary dependencies

```
FROM debian
RUN apt-get update \
 && apt-get -y install --no-install-recommends \
    openjdk-8-jdk ssh vim
COPY target/app.jar /app
CMD ["java", "-jar", "/app/app.jar"]
```

Tips for building Docker Images Reduce Image Size

- Tip #5: Remove package manager cache
  - Package managers maintain their own cache which may end up in the image. One way to deal with it is to remove the cache in the same RUN instruction that installed packages. Removing it in another RUN instruction would not reduce the image size.

Remove package manager cache

```
FROM debian
RUN apt-get update \
 && apt-get -y install --no-install-recommends \
    openjdk-8-jdk \
 && rm -rf /var/lib/apt/lists/*
COPY target/app.jar /app
CMD ["java", "-jar", "/app/app.jar"]
```

Tips for building Docker Images Maintainability

- Tip #6: Use official images when possible
  - Official images can save a lot of time spent on maintenance because all the installation steps are done and best practices are applied. If you have multiple projects, they can share those layers because they use exactly the same base image.

Use official images when possible

```
FROM debian
RUN apt-get update \
 && apt-get -y install --no-install-recommends \
    openjdk-8-jdk\
 && rm -rf /var/lib/apt/lists/*
```

```
FROM openjdk
COPY target/app.jar /app
CMD ["java", "-jar", "/app/app.jar"]
```

Tips for building Docker Images Maintainability

- Tip #7: Use more specific tags
  - Do not use the latest tag. It has the convenience of always being available for official images on Docker Hub but there can be breaking changes over time. Depending on how far apart in time you rebuild the Dockerfile without cache, you may have failing builds.
  - Instead, use more specific tags for your base images. In this case, we're using openjdk. There are a lot more tags available so check out the Docker Hub documentation for that image which lists all the existing variants.

Tips for building Docker Images Maintainability

- Tip #8: Look for minimal flavors
  - Some of those tags have minimal flavors which means they are even smaller images. The slim variant is based on a stripped down Debian, while the alpine variant is based on the even smaller Alpine Linux distribution image. A notable difference is that debian still uses GNU libc while alpine uses musl libc which, although much smaller, may in some cases cause compatibility issues. In the case of openjdk, the jre flavor only contains the java runtime, not the sdk; this also drastically reduces the image size.

Look for minimal flavors

REPOSITORY	TAG	SIZE
openjdk	8	624MB
openjdk	8-jre	443MB
openjdk	8-jre-slim	204MB
openjdk	8-jre-alpine	83MB

Just using a different base image reduced the image size by 540 MB