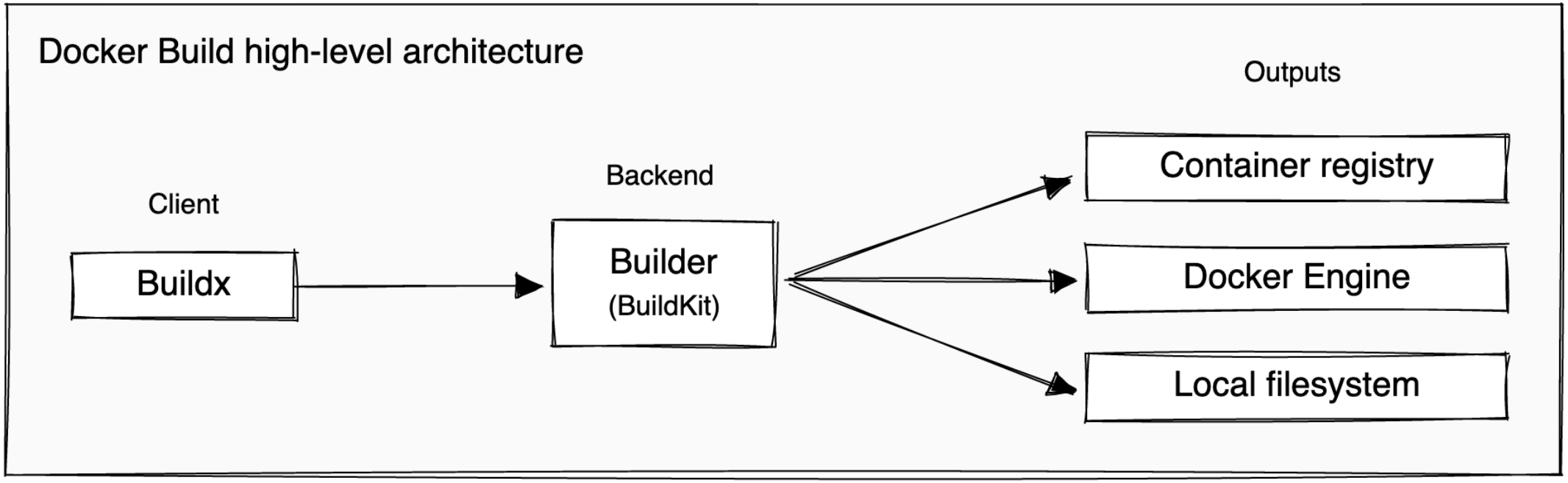
LAB-03

Build Docker Images

https://docs.docker.com/build/
https://docs.docker.com/build/guide/
https://docs.docker.com/reference/dockerfile/

Olarn Rojanapornpun

Docker Build architecture



- Buildx is the client and the user interface for running and managing builds
- BuildKit is the server, or builder, that handles the build execution.

https://docs.docker.com/build/architecture/

`build` command


```
docker image build [OPTIONS] PATH | URL | -
```

- The docker `build` command builds Docker images from a Dockerfile and a "context"
- A build's context is the set of files located in the specified PATH or URL
- The URL parameter can refer to three kinds of resources:
  Git repositories, pre-packaged tarball contexts, and plain text files
- By default, the docker build command looks for a Dockerfile at the root of the build context
- The `-f`, `--file`, option lets you specify the path to an alternative file to use instead
- Relative path are interpreted as relative to the root of the context

```
# build the image using Dockerfile in current directory and tag the image with 'hello'
docker build -t hello  .
# use the file FE.Dockerfile in FrontEnd directory instead
docker build -f FE.Dockerfile FrontEnd
```

https://docs.docker.com/reference/cli/docker/image/build/

Dockerfile

- Docker builds images by reading the instructions from a Dockerfile
- The default filename to use for a Dockerfile is Dockerfile, without a file extension
- The instruction name is not case-sensitive, but is CAPITAL by convention

Sample Dockerfile

```
FROM node:14.14.0-alpine3.12
COPY . /nodejs/.
WORKDIR /nodejs
RUN npm install
ENV VERSION 1.0
EXPOSE 8081
CMD ["node", "/nodejs/main.js"]
```

FROM instruction


```
FROM [--platform=<platform>] <image>[:<tag>] [AS <name>]
```

- The `FROM` instruction initializes a new build stage and sets the base image for subsequent instructions
- A valid Dockerfile must start with a `FROM` instruction
- `ARG` is the only instruction that may precede `FROM` in the Dockerfile

https://docs.docker.com/reference/dockerfile/#from

FORMS

- The `RUN`, `CMD`, and `ENTRYPOINT` instructions all have two possible forms:
- The exec form can be used to invoke commands using a specific command shell, or any other executable. It uses a JSON array syntax, where each element in the array is a command, flag, or argument (uses double-quotes). It is best used to specify an `ENTRYPOINT` instruction.

```
INSTRUCTION ["executable","param1","param2"] (exec form)
```

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

- The shell form is more relaxed, and emphasizes ease of use, flexibility, and readability. The shell form automatically uses a command shell, whereas the exec form does not.

```
INSTRUCTION command param1 param2 (shell form)
```

```
RUN source $HOME/.bashrc && \
echo $HOME
```

https://docs.docker.com/reference/dockerfile/#shell-and-exec-form

RUN instruction


```
# Shell form:
RUN [OPTIONS] <command> ...
# Exec form:
RUN [OPTIONS] [ "<command>", ... ]
```

- The `RUN` instruction will execute any commands to create a new layer on top of the current image
- The shell form is most commonly used

```
RUN <<EOF
apt-get update
apt-get install -y curl
EOF
```

- The available `[OPTIONS]` for the `RUN` instruction are: `--mount`, `--network`, `--security`

https://docs.docker.com/reference/dockerfile/#run

CMD instruction


```
CMD ["executable","param1","param2"] # exec form
# (exec form, as default parameters to ENTRYPOINT)
CMD ["param1","param2"]
CMD command param1 param2 # shell form
```

- The `CMD` instruction sets the command to be executed when running a container from an image
- There can only be one `CMD` instruction in a Dockerfile. If you list more than one `CMD`, only the last one takes effect
- `CMD` doesn't execute anything at build time, but specifies the intended command for the image
- The purpose of a `CMD` is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an `ENTRYPOINT` instruction as well.
- If `CMD` is used to provide default arguments for the `ENTRYPOINT` instruction, both the `CMD` and `ENTRYPOINT` instructions should be specified in the exec form.

https://docs.docker.com/reference/dockerfile/#cmd

ENTRYPOINT instruction


```
CMD ["executable","param1","param2"] # exec form
# (exec form, as default parameters to ENTRYPOINT)
CMD ["param1","param2"]
CMD command param1 param2 # shell form
```

- An `ENTRYPOINT` allows you to configure a container that will run as an executable
- Command line arguments to `docker run <image>` will be appended after all elements in an exec form `ENTRYPOINT`, and will override all elements specified using `CMD`

https://docs.docker.com/reference/dockerfile/#entrypoint
https://docs.docker.com/reference/dockerfile/#understand-how-cmd-and-entrypoint-interact

Exercise 1

1. Build '209lab3:ex1-1' image based on openjdk:17-jdk-alpine. Configure the image to run `sh` as default.
2. Build '209lab3:ex1-2' image based on busybox. Configure the image to execute `echo hello world` when run the container. Allow different string to be display.

COPY instruction

- `COPY` has two forms. The latter form is required for paths containing whitespace.

```
COPY [OPTIONS] <src> ... <dest>
COPY [OPTIONS] ["<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.
- Each `<src>` may contain wildcards
- The `<dest>` is an absolute path, or a path relative to `WORKDIR`, into which the source will be copied inside the destination container.

- The available `[OPTIONS]` for the `RUN` instruction are: `--from`, `--chown`, `--chmod`, `--link`, `--parents`, `--exclude`

https://docs.docker.com/reference/dockerfile/#copy

COPY instruction

`COPY` obeys the following rules:

- The `<src>` path is resolved relative to the build context. If you specify a relative path leading outside of the build context, such as `COPY ../something /something`, parent directory paths are stripped out automatically. The effective source path in this example becomes `COPY something /something`
- If `<src>` is a directory, the entire contents of the directory are copied, including filesystem metadata.

  > **Note**
  >
  > The directory itself isn't copied, only its contents.

- If `<src>` is any other kind of file, it's copied individually along with its metadata. In this case, if `<dest>` ends with a trailing slash `/`, it will be considered a directory and the contents of `<src>` will be written at `<dest>/base(<src>)`.
- If multiple `<src>` resources are specified, either directly or due to the use of a wildcard, then `<dest>` must be a directory, and it must end with a slash `/`.
- If `<src>` is a file, and `<dest>` doesn't end with a trailing slash, the contents of `<src>` will be written as filename `<dest>`.
- If `<dest>` doesn't exist, it's created, along with all missing directories in its path.

https://docs.docker.com/reference/dockerfile/#copy

Exercise 2

1. Build ‘dateapi:v1’ image based on openjdk:17-jdk-alpine.
   Copy .jar file and configure the image to run the jar file. Allow the port to be specified during run
2. Build ‘fetchdate:v1’ image based on nginx:alpine.
   Copy dist/ to the image (replace default html)