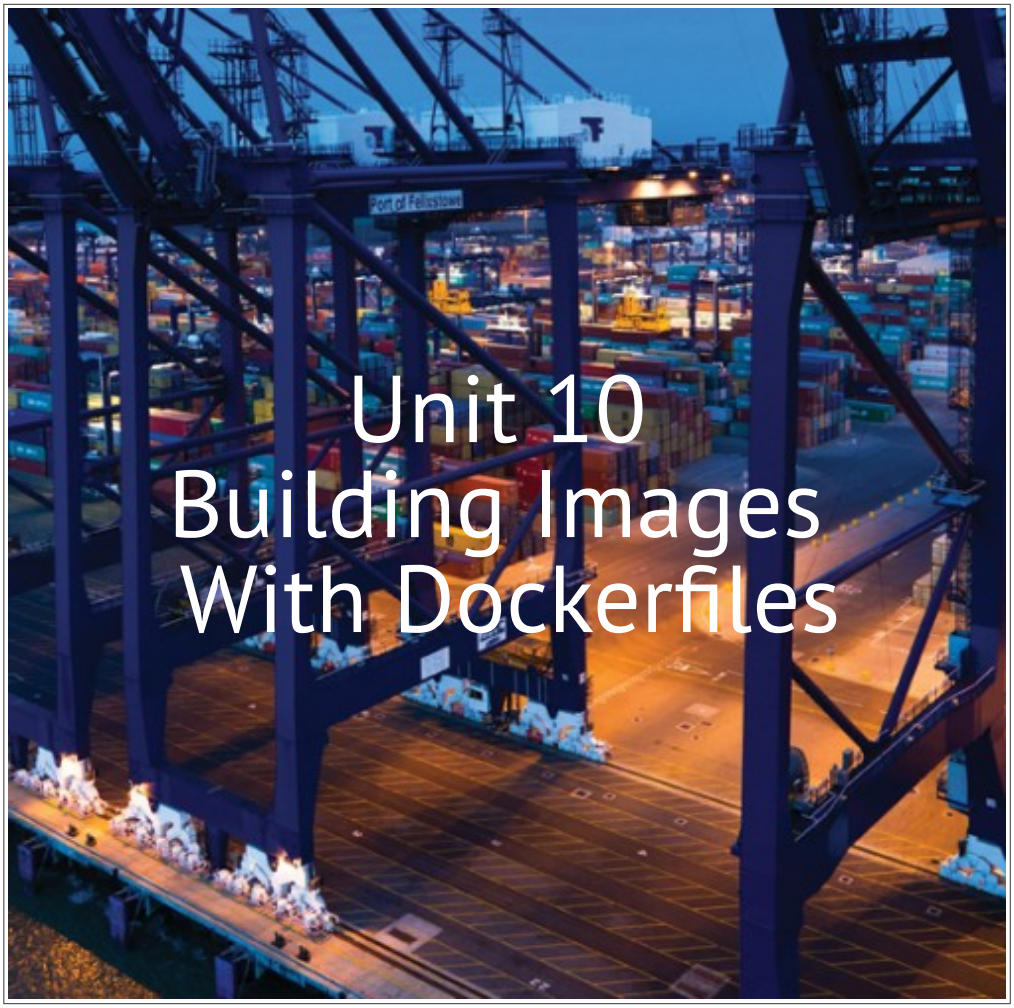


Unit 10
Building Images With Dockerfiles

Objectives

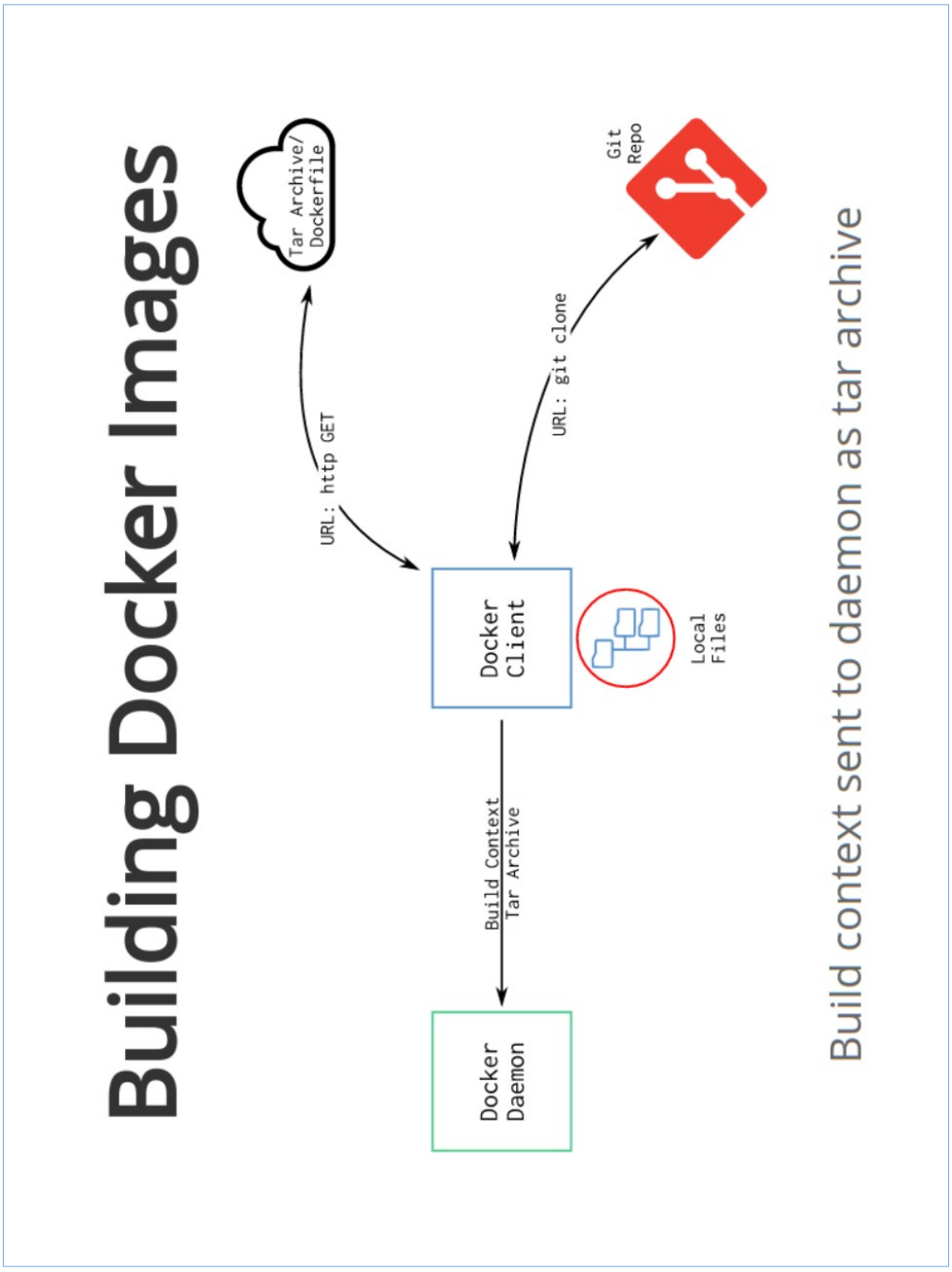
Upon completion of this unit, you will be able to:

- Describe how the image build process works
- Detail the Dockerfile instruction set, and the purpose of each instruction
- Define an appropriate configuration of command-oriented Dockerfile instructions
- Describe how Docker utilises an image cache for build efficiency

Agenda

The following topics will be covered in this unit:

- Overview of Building Images With Dockerfiles
- The `docker build` Command
- The Dockerfile Instruction Set
- The Build Cache
- Lab exercise: Building an Image With a Dockerfile


Building Docker Images
Tar Archive/
Dockerfile
URL: http GET
Git
Repo
URL: git clone
Docker
Client
Local
Files
Build Context
Tar Archive
Docker
Daemon
Build context sent to daemon as tar archive

Building Docker Images

- Previously, we saw how to create Docker images using the `docker commit` command
- However, creating new images based on the delta between a running container's filesystem and the image it is created from, is not considered the best approach to authoring images
- A more structured and refined approach is required for creating images that are efficient, secure and fit for purpose
- Docker provides an alternative, more considered method for authoring images, using a set of Docker-specific instructions and the `docker build` CLI command
- The allowable set of instructions conform to a syntax, and are encapsulated in a text file known as a 'Dockerfile'
- Each instruction is parsed by the daemon, and the relevant instruction is executed within a temporary container, and the resulting filesystem changes are committed as an image layer before the container is removed
- Building an image is tantamount to executing a series of co-ordinated `docker commit` commands, which follow the execution of OS commands in containers based on the image layer produced from the previous Dockerfile instruction
- Most Dockerfile instructions result in the addition, deletion or modification of files, but some instructions are 'no-op' instructions, which result in a new empty image layer even though no filesystem amendments occur
- The `docker build` command needs a Dockerfile from which to build the image, and optionally a build context, which contains other artefacts in the form of files and directories, which will be injected into the image
- The location of the build context is provided as a local filesystem path, a URL or can be piped as a tar archive from STDIN, and along with the Dockerfile, its contents are uploaded to the Docker daemon as a tar archive for the build process
- If a URL references a Git repository, with a Dockerfile at its root, its contents are cloned to a temporary location on the client, before being uploaded to the daemon
- A URL may also reference a remote Dockerfile, or a remote tar archive containing the build context with a Dockerfile at its root

Notes

Local Build Context

```
Dockerfiles/
├── docker-goredchat
│   ├── Dockerfile
│   ├── goredchat.go
│   └── README.md
├── docker-revealjs
│   ├── Dockerfile
│   ├── onbuild
│   │   ├── docker-entrypoint.sh
│   │   └── Dockerfile
│   └── README.md
.   .
```

Example Project Structure

```
Dockerfile
conf*
data/*
*.sh
!docker-entrypoint.sh
docker-compose.yml
!*.yml
.git
test/case??
db/*.sqlite3*
vendor/cache/*
**/tmp.*
```

Example .dockerignore File

Local Build Context

- For a build where the build context is local to the Docker client, Docker assumes the Dockerfile to be called `Dockerfile`, and to be located at the root of the build context
- It's possible to specify an alternate location and name for the Dockerfile, but it must be within the build context
- For the purposes of consistency and clarity, it is recommended that all local host files for each authored image, should reside in a unique directory with the Dockerfile at the root
- If the Docker daemon and client are remote from each other, then care should be taken to ensure that only artefacts relevant to the build are contained within the build context, in order to avoid unnecessary bandwidth usage and lengthy build times
- During the course of image development, it's possible that the build context may contain a number of transient and spurious files, and it could become laborious to remove them for each build iteration
- Any files in the build context that are not required for the build, can be excluded from the context (i.e. from being uploaded to the daemon), if their names adhere to a pattern defined in a file called '`.dockerignore`'
- The `.dockerignore` file must reside at the location of the root of the build context
- The `.dockerignore` file is a text file, and it contains a list of excluded filename patterns separated by a newline
- The patterns must conform to the rules associated with Golang's `filepath.Match` function
- The filename patterns specified are treated as absolute in relation to the root of the build context
- The `!` operator excludes a particular pattern for omission, but the desired result may be overridden if another pattern also matches and contradicts later in the file
- The recursive globbing operator `**` matches any ensuing pattern in all directories within the build context

Notes

Git Build Context

A URL for a remote Git repo can contain a fragment, with a context and optional sub-directory for the build:

```
https://path/to/repo/repo.git#context:sub-dir
```

URL	Commit	Sub-Directory
repo.git	refs/heads/master	/
repo.git#tag	refs/tags/tag	/
repo.git#branch	refs/heads/branch	/
repo.git#232e26f	SHA1 hash = 232e26f	/
repo.git:sub-dir	refs/heads/master	/sub-dir
repo.git#tag:sub-dir	refs/tags/tag	/sub-dir
repo.git#branch:sub-dir	refs/heads/branch	/sub-dir
repo.git#232e26f:sub-dir	SHA1 hash = 232e26f	/sub-dir

Git Build Context

- As we've already seen, a remote Git repository can be specified as the context for a Docker image build
- The URL specified for the location of the build can contain a fragment, which can reference either a branch, a tag or a commit hash, and optionally a sub-directory within the repository
- The format of the remote URL takes the form:

```
https://path/to/repo/repo.git#context:sub-dir
```

- The following examples show how the construction of the URL determines the context of the build:

URL	Commit	Sub-Directory
`https://path/to/repo/repo.git`	`refs/heads/master`	`/`
`https://path/to/repo/repo.git#tag`	`refs/tags/tag`	`/`
`https://path/to/repo/repo.git#branch`	`refs/heads/branch`	`/`
`https://path/to/repo/repo.git#232e26f`	`SHA1 hash = 232e26f`	`/`
`https://path/to/repo/repo.git#:sub-dir`	`refs/heads/master`	`/sub-dir`
`https://path/to/repo/repo.git#tag:sub-dir`	`refs/tags/tag`	`/sub-dir`
`https://path/to/repo/repo.git#branch:sub-dir`	`refs/heads/branch`	`/sub-dir`
`https://path/to/repo/repo.git#232e26f:sub-dir`	`SHA1 hash = 232e26f`	`/sub-dir`

- The URL can be specified using the git protocol (`git://`), the ssh protocol (`git@`), as well as the `https` protocol

Notes

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:

Client 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

- The format of the docker build command is:

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

- The command line options available for the `docker build` command are:

`-f, --file=Path/Dockerfile`: specifies the file to use for the Dockerfile, which is assumed to be a file called `Dockerfile` at the root of the build context, but this option specifies an alternate location within the build context
`--force-rm=false`: specifies that the intermediate containers created by each Dockerfile instruction, will always be removed on successful completion, or otherwise
`--no-cache=false`: specifies that the image cache is bypassed when building an image, thereby creating a new image layer each time, even if an equivalent, sibling already exists
`--pull=false`: specifies that prior to the build, a check is made to determine if the base image is newer than its version in the local repository, and if so it is downloaded first
`-q, --quiet=false`: specifies that the build process be conducted very quietly, only providing significant output on build failure
`--rm=true`: specifies that intermediate containers are removed after a successful build
`-t, --tag=[]`: specifies a repository name (and optional tags) for the successfully build image
`--build-arg=[]`: specifies an environment variable for use during the build process

- If the path for the `-f`, `--file` configuration option is relative rather than absolute, it must be relative to the local current working directory, or in the case of a remote build context (i.e. via URL), must be relative to the root of the remote context
- The following additional configuration options are available to apply to the containers run during the build process, and have the same meanings as those described in Unit 4; `--cpu-shares`, `--cgroup-parent`, `--memory`, `--memory-swap`, `--shm-size`, `--ulimit`, `--cpu-period`, `--cpu-quota`, `--cpuset-cpus`, `--cpuset-mems`, `--isolation`

Notes

Image Build Example

This Dockerfile shows the instructions for creating a simple implementation of the lighttpd http server:

```
FROM debian

RUN apt-get -qq update && \
    DEBCONF_NOWARNINGS=yes apt-get install -y lighttpd > /dev/null && \
    rm -rf /var/lib/apt/lists/*

EXPOSE 80

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

The image can be built with the following command:

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

In this example, the build context is empty

Image Build Example

- The following simplified Dockerfile content, shows the instructions required to create an image for the lighttpd http server

```
FROM debian

RUN apt-get -qq update && \
    DEBCONF_NOWARNINGS=yes apt-get install -y lighttpd > /dev/null && \
    rm -rf /var/lib/apt/lists/*

EXPOSE 80

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

- Executing a `docker build` command, provides the following output on `STDOUT`:

```
$ docker build -t lighttpd .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM debian
 ---> f50f9524513f
Step 2 : RUN apt-get -qq update && DEBCONF_NOWARNINGS=yes apt-get
install -y lighttpd > /dev/null && rm -rf /var/lib/apt/lists/*
 ---> Running in 3a2f6d21e6d5
 ---> 984b28810809
Removing intermediate container 3a2f6d21e6d5
Step 3 : EXPOSE 80
 ---> Running in 7010e91c2c24
 ---> 8875a41beb91
Removing intermediate container 7010e91c2c24
Step 4 : ENTRYPOINT lighttpd -D -f /etc/lighttpd/lighttpd.conf
 ---> Running in 72460c095bee
 ---> 887fe534d792
Removing intermediate container 72460c095bee
Successfully built 887fe534d792
```

Notes

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

- Every Dockerfile must have a `FROM` instruction
- The `FROM` instruction references the base image from which to base the new image being built, and takes the form:

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

- The image specified must have a fully qualified image name (an image ID cannot be used), but a tag or digest hash is optional
- In the event that a tag is omitted, Docker assumes 'latest', and returns an error if a specified tag does not exist
- Apart from whitespace and comments (which are benign, and designated with a `#`), the `FROM` instruction must be the first instruction in a Dockerfile
- It is possible, although uncommon, to have multiple `FROM` instructions in a single Dockerfile, which produces as many image branches as there are `FROM` instructions
- In such situations, each image branch will comprise of the image layers associated with a `FROM` statement up to, but excluding, the next `FROM` statement or the end of the Dockerfile, whichever occurs first
- Care should be taken in choosing a base image to work with, in order to ensure fitness for purpose, and good provenance
- The official, curated images on the Docker Hub registry are a good starting point, but many organisations maintain proprietary base images
- The reserved word `scratch`, refers to an empty image, and when used in conjunction with `FROM`, results in a no-op instruction, with no image layer produced
- `FROM scratch` is used to for creating base images, or very minimal images
- Historically, a library `scratch` image existed on the Docker Hub registry of size `0` bytes, but since `scratch` became a reserved word, it can longer be pulled

Notes

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

MAINTAINER Instruction

- The `MAINTAINER` instruction sets the author attribute in the image's metadata, and is a no-op instruction
- The `MAINTAINER` instruction takes the form:

```
MAINTAINER name
```

- It's useful to provide the name and email address of the image's maintainer, especially in the public namespace of the Docker Hub registry, so that third parties can address their queries to the maintainer
- The author attribute can be retrieved from an image using the command:

```
$ docker inspect -f '{{.Author}}' nbrown/revealjs
Nigel Brown <nigel@windsock.io>
```

Notes

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

RUN Instruction

- The `RUN` instruction executes a command in a container, and the resulting filesystem amendments are written to a new image layer
- The `RUN` instruction takes one of two forms:

```
RUN command
RUN ["prog", "arg1", "arg2", ...]
```

- The first form is termed the 'shell form', and a command shell is invoked to execute the command
- In the shell form, Docker uses `/bin/sh -c` to execute the command supplied
- Commands are often daisy-chained together to optimise the number of layers in the final image, e.g.

```
RUN apt-get update && rm -rf /var/lib/apt/lists/*
```

- A backslash (\) can be used to continue the definition of a RUN instruction onto a new line, thereby enhancing the readability of a Dockerfile
- The second form is termed the 'exec form', where a specified command or program is executed, but not in a command shell
- The purpose of the exec form is to cater for images that do not contain the `/bin/sh` executable
- No command shell, means no environment variable substitution or other command shell processing, which is inherently possible with the shell form
- The parameters for the exec form need to be enclosed in double quotes, as they are parsed as a JSON array, and the JSON standard requires double quotes
- If an alternative command shell is required to execute commands via a Dockerfile, this can be achieved with the exec form, e.g.

```
RUN ["/bin/zsh", "-c", "rm", "-f", "/tmp/data"]
```

Notes

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**

ENV Instruction

- The `ENV` instruction sets a key/value pair or pairs, much like a shell environment variable, and can take one of two forms:

```
ENV key value
ENV key=value ...
```

- The first form sets a single key/value pair, where ALL characters (including spaces and quotes) immediately after the space following the key, form the value
- The second form allows for setting multiple key/value pairs in a single instruction, and quoting and the use of the backslash (\) character can be applied for escaping characters, as you might with the command line
- When there are multiple variables to set, it's better to use the second form to reduce layer numbers, improving readability with the backslash ('\') continuation character
- The key/value pairs are referenced in a Dockefile, using the usual shell-like notation; `$key` or `${key}`, and can be escaped using '\' to enumerate to a literal equivalent
- Two other bash-like variable constructs can be used; `${key:-word}` and `${key:+word}`, the former sets `key` to `word` if `key` is not set, and the latter sets `key` to `word` if set, and to an empty string otherwise
- Once set in the Dockerfile, a key/value pair is available for use inside intermediate containers during a build, and can therefore be used in the commands specified with the `RUN` instruction, e.g.

```
ENV VERSION 1.4.25
RUN wget http://www.memcached.org/files/memcached-${VERSION}.tar.gz
```

- Any key/value pairs set in a Dockerfile, also persist (as environment variables) into any container that is derived from the resultant image being built
- The following instructions can make use of key/value pairs within a Dockerfile: `ADD`, `COPY`, `ENV`, `EXPOSE`, `LABEL`, `ONBUILD`, `STOPSIGNAL`, `USER`, `VOLUME` and `WORKDIR`*

* The `ONBUILD` instruction must reference one of the other instructions that can make use of key/value pairs

Notes

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 **persistance** in images

ARG Instruction

- The ARG instruction enables a user to define a key/value pair for use within a Dockerfile at the point of build, and works in conjunction with the --build-arg option
- The ARG instruction takes the following form, where value is an optional default value, which is applied if the --build-arg configuration option is not used:

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

- If a key called VERSION is to be used in a Dockerfile, then it can be defined in the Dockerfile (e.g. ARG VERSION or ARG VERSION=1.4.21), and set dynamically at build time:

```
$ docker build --build-arg VERSION=1.4.25 -t memcached .
```

- As with the ENV instruction, a key/value pair is only valid for use after the ARG instruction sets it, and unlike with the ENV instruction, key/value pairs defined with the ARG instruction do not persist beyond the image build
- Additionally, if a key/value pair is defined with both the ARG and ENV instruction, the ENV instruction overrides the ARG instruction
- However, the ENV and ARG instructions can be used in conjunction to ensure that a key and its value provided by the --build-arg configuration option is propagated to any containers derived from the built image
- Consider a Dockerfile containing the following:

```
ARG VERSION
ENV VERSION=${VERSION:-1.4.21}
```

- If the --build-arg configuration option is unused, the key VERSION has the value 1.4.21, but if --build-arg VERSION=1.4.25 is used, VERSION has the value 1.4.25, which persists in the image
- A pre-defined set of keys is provided by Docker, that do NOT need an ARG instruction: HTTP_PROXY, http_proxy, HTTPS_PROXY, https_proxy, FTP_PROXY, ftp_proxy, NO_PROXY, no_proxy

Notes

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**
- **All 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

EXPOSE Instruction

- The `EXPOSE` instruction is used to specify which of an image-derived containers' ports are exposed for the provision of its services
- The `EXPOSE` instruction has one or more port values separated by whitespace, and takes the form:

```
EXPOSE port ...
```

- Ports of an image-derived container that are designated with the `EXPOSE` instruction are earmarked for external consumption, but the `EXPOSE` instruction does not by itself make a derived container's service consumable
- Ports designated for exposure with the `EXPOSE` instruction, can be forwarded to a random host port at runtime with the `-P`, `--publish-all` configuration option
- If a container's port is to be forwarded to a specific port on the host with the `-p`, `--publish` configuration option, it is not necessary to declare the container's port exposed using the `EXPOSE` instruction
- An `EXPOSE` instruction is not required in an image for a derived container to have its service consumed by another container on the same network

Notes

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**

VOLUME Instruction

- The `VOLUME` instruction specifies a mount point in the image, which will be used to bind-mount a directory from the host into a derived container's filesystem, which bypasses the normal COW or union filesystem
- It is a deterministic method for creating a container volume, which can be shared with other containers, when they are invoked with the `--volumes-from` Docker CLI command line option
- Any data at the location specified in the image will be copied to the derived container's directory on the host
- The syntax for the `VOLUME` instruction can take one of two forms:

```
VOLUME path ...
VOLUME [“path1”, “path2”, ...]
```

- The latter form must use double quotes around the volumes specified, as required by JSON for a JSON array
- The volume, that is the directory on the host, and its contents will persist until no containers reference it, and it is explicitly removed along with the sole container that references it, using the `docker rm` command line option, `-v`, `--volumes`
- Alternatively, provided no containers reference it, it can also be removed using the `docker volume rm` sub-command
- Care should be taken when setting up volumes in a Dockerfile; any volume preparation must occur before the `VOLUME` instruction, and not after it
- Any operations such as adding files, changing ownership or permissions, after the `VOLUME` instruction, will be conducted in a volume associated with an intermediate container, and the changes will not be written to the image layer

Notes

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

STOPSIGNAL Instruction

- The `STOPSIGNAL` instruction provides Docker with the signal it should use in order to make the container exit (i.e. `docker stop`)
- The format of the `STOPSIGNAL` instruction is:

```
STOPSIGNAL signal
```

- The 'signal' can be a number or a name that corresponds to valid signal syscalls found in the kernel syscall table (e.g. `15` or `SIGTERM`)
- The `docker stop` command sends the `SIGTERM` signal to the main container process, but if this is not handled by the process, it will be ignored
- If a process does not respond to the `SIGTERM` signal after 10 seconds, it is followed by a `SIGKILL`, which could introduce unexpected side effects if an application doesn't exit gracefully
- Sometimes, an application is programmed to handle a different signal in order to exit gracefully, and the `STOPSIGNAL` instruction provides the image builder with the ability to change the default `SIGTERM` signal used by Docker, for one that matches the requirements of the application

Notes

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

LABEL Instruction

- The `LABEL` instruction enables the image builder to annotate their images with labels
- The format of the `LABEL` instruction is:

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

- The `LABEL` instruction allows for adding metadata to images, which can be used for describing images, and annotating images for searching and selecting with the Docker CLI, amongst other things
- Spaces in the value component of the label can be escaped with a '\', or with the use of quotes, whilst the backslash ('\') continuation character can extend the `LABEL` instruction over more than one line:

```
LABEL Description="A Ghost Blogging Platform Docker Image" \
      Version=0.7.8
```

- Multiple labels should be defined with a single `LABEL` instruction in order to maintain image efficiency with a single layer
- An image's labels can be inspected with the `docker inspect` command:

```
$ docker inspect -f '{{json .Config.Labels}}' ghost | jq '.'
{
  "Version": "0.7.8",
  "Description": "A Ghost Blogging Platform Docker Image"
}
```

- As we have seen see previously, the docker images CLI command can filter its output based on labels:

```
$ docker images -f label="Version=0.7.8"
REPOSITORY     TAG         IMAGE ID      CREATED          SIZE
ghost          0.7.8       85d91e744437  14 minutes ago   125.1 MB
```

Notes

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**

WORKDIR Instruction

- The WORKDIR instruction sets the working directory for any subsequent ADD, COPY, RUN, CMD and ENTRYPOINT instructions in a Dockerfile
- The format of the WORKDIR instruction is:

```
WORKDIR path
```

- Thus, the WORKDIR instruction used in conjunction with an ADD, COPY or RUN instruction, is relevant during an image build, whereas the WORKDIR instruction used in conjunction with a CMD and/or ENTRYPOINT instruction, is relevant when invoking a container
- There can be multiple WORKDIR instructions in a single Dockerfile
- The context starts at '/', and each WORKDIR instruction changes the context for the relevant Dockerfile instructions, up to the next WORKDIR instruction, where the context will change once more
- Environment variables in the Dockerfile will be expanded for the WORKDIR instruction
- Absolute and relative paths can be used, but relative paths will be relative to '/' or the last WORKDIR instruction

```
FROM fedora:latest
# Change working dir to /var/log
WORKDIR var/log
RUN touch myapp.log
ENV APPDIR=/var/lib/myapp
RUN mkdir $APPDIR
# Change working dir to /var/lib/myapp
WORKDIR $APPDIR
CMD myapp
```

Notes

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

USER Instruction

- The `USER` instruction sets the username or UID for subsequent `RUN`, `CMD` and `ENTRYPOINT` instructions
- The format of the `USER` instruction is:

```
USER username|uid
```

- The main process that runs in a container derived from a valid image, does so with UID `0`, unless the `USER` instruction is employed to change that behaviour – it is good practice, where possible, to change the process' user to one other than UID `0`
- The `USER` instruction can be used multiple times within a Dockerfile
- Whilst it's possible to specify a random UID instead of a username, some applications may not start if their owner does not have an entry in `/etc/passwd`
- Whilst specifying a username that doesn't exist in `/etc/passwd` will result in a successfully built image, a derived container will not start, and Docker will error instead
- Ordinarily, the `USER` instruction is employed following a `RUN` instruction which adds a user to the image's `/etc/passwd` file with a utility such as `useradd`

Notes

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

COPY Instruction

- The `COPY` instruction copies files and directories to a specified location in the image, and takes one of two forms:

```
COPY src ... dest
COPY [“src”], ... [“dest”]
```

- The second form is required when a source or destination path contains whitespace
- There may be multiple source elements specified, which must be relative to, and contained within, the build context
- There must be a single destination specified, which must be an absolute path, or a path relative to the current directory, as specified by a preceding `WORKDIR` instruction, or `/`
- If a specified source element is a directory, its contents (and not the directory itself) are copied to the destination
- Wildcards can be used to specify source elements using Go's `filepath.Match` patterns
- If there are multiple source elements, whether defined explicitly or enumerated, the destination must be a directory and end with a '`/`'
- If the source element is a file, and the destination has a trailing '`/`', the file is copied to the destination with the directory component of its path stripped:

```
source/path/to/file -> dest/file
```

- If the source element is a file, and the destination is also to be a file, then the file is copied as the destination, and will be located in the the current directory:

```
source/path/to/file -> dest
```

- If the specified destination doesn't exist in the image, it is created along with any intermediary directories in the path
- The metadata associated with source files and directories is preserved during the copy, but the files and directories are owned by UID `0` in the destination

Notes

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

ADD Instruction

- Similar to the COPY instruction, the ADD instruction adds files and directories to the image, and takes the same forms as the the COPY instruction::

```
ADD src1 src2 ... dest
ADD [“src1”, “src2”, ... “dest”]
```

- The ADD instruction existed before the COPY instruction in the Docker Engine's history, and addresses additional capabilities when compared with the COPY instruction
- However, the ADD instruction has been deemed confusing, and it is recommended that wherever possible, the COPY instruction should be used ahead of the ADD instruction
- All that applies to the COPY instruction in terms of behaviour, also applies to the ADD instruction
- Unlike the COPY instruction, the ADD instruction also allows for the addition of files and directories from a remote location specified by a URL:

```
ADD https://github.com/krallin/tini/releases/download/v0.9.0/tini
```

- If the source is a URL, and it has an HTTP Last-Modified Header, the timestamp from the header will be used to set the file's modification time when it is copied
- Additionally, if the source element(s) are local (un)compressed (gzip, bzip2 or xz) tar archives, the ADD instruction will automatically extract the archive at the destination, and perform a union with the destination, with any conflicts resolved in favour of the source
- Remote archives are NOT extracted automatically by the ADD instruction
- With regard to image layers, it is considered inefficient to use the ADD instruction to retrieve a remote (un)compressed archive, which will inevitably require additional commands to extract the files, which can be achieved with a single RUN instruction (e.g. wget/curl -> tar –x -> rm)

Notes

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

CMD Instruction

- The `CMD` instruction is used to provide default execution parameters for containers, and has three different formats:

```
CMD [“arg1”, “arg2”, ...]
CMD [“prog”, “arg1”, ...]
CMD prog arg1 ...
```

- The first form provides the default arguments to an `ENTRYPOINT` instruction, and must be in the JSON format, including the double quotes
- The second and third forms provide a program, executable or command to execute along with arguments, in exec and shell formats, respectively
- Unlike the exec form, the shell form invokes the command in a shell using `/bin/sh -c`, and normal shell processing behaviour occurs (e.g. variable substitution)
- If a command is specified as an argument to the Docker CLI commands `docker create` or `docker run`, it will override a command specified with the `CMD` instruction
- Hence, if a Dockerfile has a `CMD` instruction:

```
CMD [“haproxy”, “-f”, “/etc/haproxy/haproxy.cfg”]
```

  it could be overridden at container creation time with:

```
docker run -t --rm haproxy -c -f /etc/haproxy/haproxy.cfg
```

- If a Dockerfile contains more than one `CMD` instruction, the last instruction in the sequence will apply during the build
- It is recommended to use the exec form rather than the shell form, as the command that is executed when using the shell form is `sh -c command`, and `sh` is PID `1` in the container, which will be the recipient of signals instead of the command
- Naturally, using the shell form will also cause problems if your image doesn't contain `sh`

Notes

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**

ENTRYPOINT Instruction

- The ENTRYPOINT instruction is used to run a container in the form of an executable, and has two forms:

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

- Any command line parameters provided to docker create or docker run, or in their absence the contents of a CMD instruction, will be treated as if they were appended to the ENTRYPOINT instruction
- For example, an image called find, with an ENTRYPOINT ["/usr/bin/find", "/var"], could form the basis for a container invoked with the following

```
$ docker run -t find:latest -name *.log
```

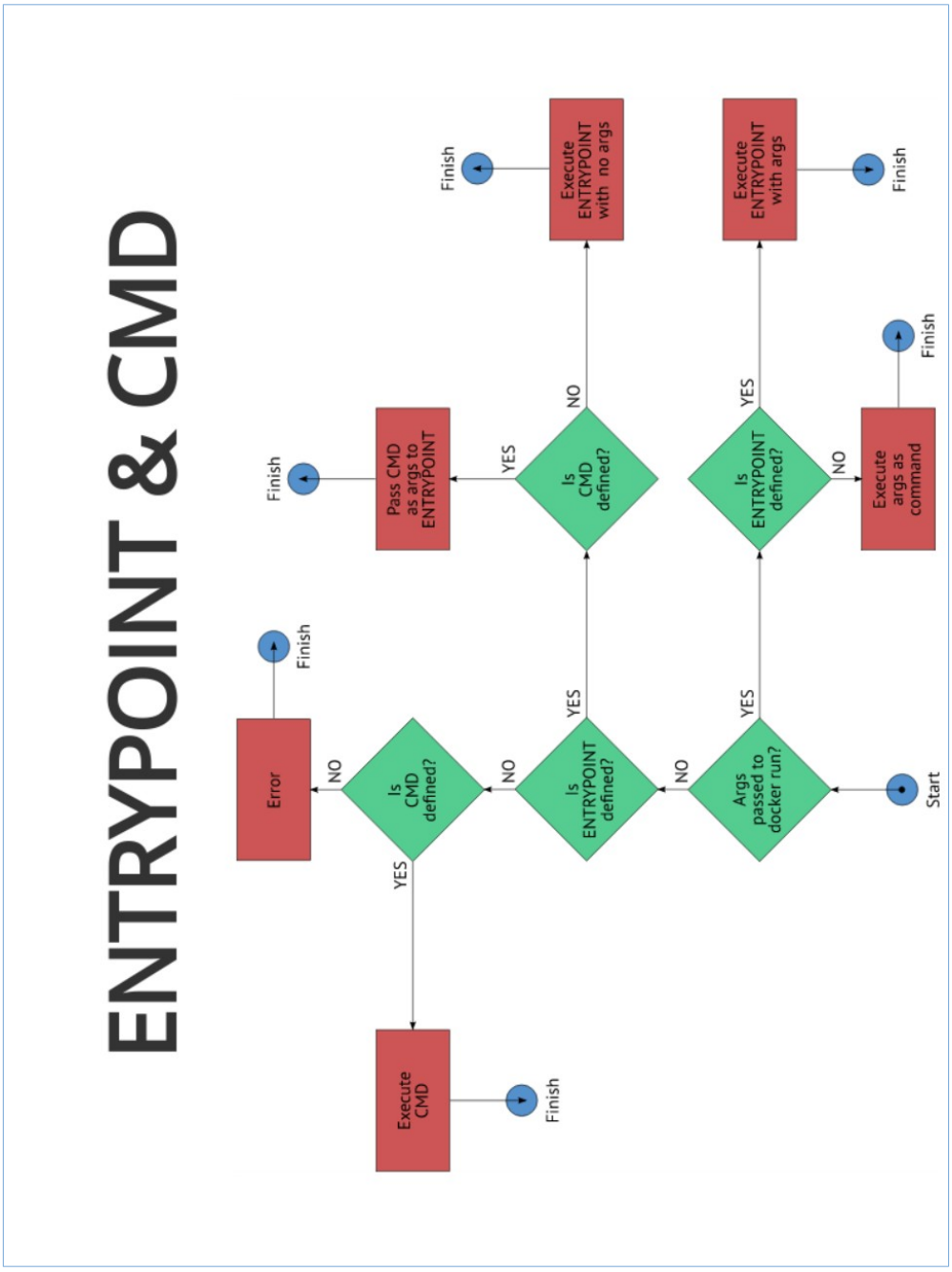
- This would execute the following command inside the container:

```
find /var -name *.log
```

- If the shell form of the ENTRYPOINT instruction is used, the parameters provided as default arguments with the CMD instruction, or any arguments provided with docker create or docker run, are ignored
- The shell form invokes the command or program with /bin/sh -c, and as such it is not PID 1 in the PID namespace associated with the container, and cannot directly catch any signals (e.g. SIGTERM from docker stop)
- The ENTRYPOINT specified in a Dockerfile can be overridden using the --entrypoint command line option for docker create or docker run, which will invoke the exec form
- Arguments can be passed to the revised ENTRYPOINT as arguments to docker create or docker run, e.g.

```
$ docker create -t --entrypoint="/usr/bin/du" find:latest -h -d 1 .
```

Notes


ENTRYPOINT & CMD
Start
Args passed to docker run?
YES
NO
Is ENTRYPOINT defined?
YES
NO
Is CMD defined?
YES
NO
Error
Finish
Execute CMD
Finish
Pass CMD as args to ENTRYPOINT
Finish
Execute ENTRYPOINT with no args
Finish
Is ENTRYPOINT defined?
YES
NO
Execute ENTRYPOINT with args
Finish
Execute args as command
Finish

ENTRYPOINT & CMD

- A Docker image must have either a `CMD` instruction or an `ENTRYPOINT` instruction, or can have both
- It can be confusing when being introduced to the `ENTRYPOINT` and `CMD` instructions – which should you use in which scenario?
- In essence, the `CMD` instruction should be used to provide a default command or arguments, which are expected to be overridden on occasions, by arguments supplied to the Docker CLI commands at runtime
- Whilst the entrypoint can be overridden with the `--entrypoint` configuration option, generally, the `ENTRYPOINT` instruction should be used when the container is to be invoked like a binary, with the possible use of alternative arguments
- The flowchart depicts the relationship between the `ENTRYPOINT` and `CMD` instructions
- If no arguments are provided to the Docker CLI command, and no `ENTRYPOINT` or `CMD` instructions are provided in the image, then the daemon will return an error
- If no arguments are provided to the Docker CLI command, and no `ENTRYPOINT` instruction is provided, but a `CMD` instruction is, what is specified in the `CMD` instruction is executed in a derived container
- If no arguments are provided to the Docker CLI, but an `ENTRYPOINT` instruction exists, then if a corresponding `CMD` instruction exists for the image, then the `ENTRYPOINT` instruction is executed with the `CMD` instruction (arguments) appended; otherwise the `ENTRYPOINT` instruction is executed with no further arguments
- If the Docker CLI command provides user specified arguments, and no `ENTRYPOINT` instruction is provided, the passed arguments are executed as the command
- If the Docker CLI command provides user specified arguments, and an `ENTRYPOINT` instruction exists in the image, the arguments are appended to the command specified by the `ENTRYPOINT` instruction
- Care should be taken with the instruction forms when using both instructions in a single Dockerfile; remember the shell form invokes commands with `sh -c`

Notes

Entrypoint Scripts

- Using **entrypoint scripts** is considered good practice, and caters for **multiple use scenarios**
- The **official Redis image** has an entrypoint script with a default argument provided by a CMD intsruction:

```
ENTRYPOINT ["/entrypoint.sh"]
CMD ["redis-server"]
```

- Entrypoint script for Redis image:

```
#!/bin/bash
set -e

if [ "$1" = 'redis-server' ]; then
        chown -R redis .
        exec gosu redis "$@"
fi

exec "$@"
```

Entrypoint Scripts

- It's considered good practice to use the `ENTRYPOINT` instruction in conjunction with a script, to cater for a number of possible scenarios, without causing confusion to the image user
- Let's take the official Redis image as an example; we may want to cater for three different scenarios:

    1. Run the container as a Redis server, i.e. `redis-server`
    2. Run the container as a Redis server with arguments, i.e. `redis-server --appendonly yes`
    3. Simply run an interactive bash command shell in the container

- This is achieved with an `ENTRYPOINT` instruction pointing to a script called `entrypoint.sh`, and a `CMD` instruction specifying an argument for the script, which is `redis-server`
- A snippet from the Dockerfile shows the relevant instructions:

```
ENTRYPOINT ["/entrypoint.sh"]
CMD ["redis-server"]
```

- The script, `entrypoint.sh`, contains the following:

```
#!/bin/bash
set -e

if [ "$1" = 'redis-server' ]; then
      chown -R redis .
      exec gosu redis "$@"
fi

exec "$@"
```

Notes

Redis Scenario 1

```
#!/bin/bash
set -e

if [ "$1" = 'redis-server' ]; then
        chown -R redis .
        exec gosu redis "$@"
fi

exec "$@"
```

Container Execution:

```
# Command executed -> entrypoint.sh redis-server
$ docker run -d redis
```

- Docker uses the argument provided by the CMD instruction, **redis-server**
- Changes file ownership, executes redis-server as user redis in container, and with **PID 1**

Redis Scenario 1

- A reminder of the `entrypoint.sh` script:

```
#!/bin/bash
set -e

if [ "$1" = 'redis-server' ]; then
      chown -R redis .
      exec gosu redis "$@"
fi

exec "$@"
```

- The container is invoked with the command:

```
$ docker run -d redis
```

- No arguments are provided, therefore the command provided by the `ENTRYPOINT` instruction is executed with additional arguments provided by the `CMD` instruction:

```
entrypoint.sh redis-server
```

- The script tests the first argument it's passed, and if it's the string '`redis-server`', it changes the ownership of the directory tree to the user '`redis`', and executes with sudo-like privileges, replacing the script as PID `1` in the container
- As `redis-server` is now PID `1` inside the container, it is possible to send signals to it in order to influence its execution; specifically, the `redis-server` process will respond to the command `docker stop`, and halt its execution gracefully

Notes

Redis Scenario 2

```
#!/bin/bash
set -e

if [ "$1" = 'redis-server' ]; then
        chown -R redis .
        exec gosu redis "$@"
fi

exec "$@"
```

Container Execution:

```
# Command executed -> entrypoint.sh redis-server --appendonly yes
$ docker run -d redis redis-server --appendonly yes
```

- Docker uses the arguments provided by the Docker CLI command, **redis-server --appendonly yes**
- Again, changes file ownership, executes redis-server as user redis in container, and with **PID 1**

Redis Scenario 2

- A reminder of the `entrypoint.sh` script:

```
#!/bin/bash
set -e

if [ "$1" = 'redis-server' ]; then
      chown -R redis .
      exec gosu redis "$@"
fi

exec "$@"
```

- The container is invoked with the command

```
$ docker run -d redis redis-server --appendonly yes
```

- This time, arguments are provided, therefore the command provided by the `ENTRYPOINT` instruction is executed, but the additional arguments provided by the `CMD` instruction are ignored in favour of those provided by way of the Docker CLI:

```
entrypoint.sh redis-server --appendonly yes
```

- As in scenario 1, the script tests the first argument it's passed, and if it's the string '`redis-server`', it changes the ownership of the directory tree to the user '`redis`', and executes `redis-server` along with the remaining arguments with sudo-like privileges, replacing the script as PID `1` in the container
- As before, since `redis-server` is now PID `1` inside the container, it is possible to control the process using signals

Notes

Redis Scenario 3

```
#!/bin/bash
set -e

if [ "$1" = 'redis-server' ]; then
        chown -R redis .
        exec gosu redis "$@"
fi

exec "$@"
```

Container Execution:

```
# Command executed -> entrypoint.sh /bin/bash
$ docker run -it --rm redis /bin/bash
```

- Docker uses the arguments provided by the Docker CLI command, **/bin/bash**
- The test for redis-server **fails**, so the script falls through to last line and executes its argument as **PID 1**

Redis Scenario 3

- A reminder of the `entrypoint.sh` script:

```
#!/bin/bash
set -e

if [ "$1" = 'redi-server' ]; then
      chown -R redis .
      exec gosu redis "$@"
fi

exec "$@"
```

- The container is invoked with the command

```
$ docker run -it --rm redis /bin/bash
```

- Again, arguments are provided, therefore the command provided by the `ENTRYPOINT` instruction is executed, but the additional arguments provided by the `CMD` instruction are ignored:

```
entrypoint.sh /bin/bash
```

- The script tests the first argument it's passed, but this time it's not the string '`redis-server`', and so it falls through to the last line of the script, executes `/bin/bash`, and replaces the script as PID `1` in the container
- The command shell `/bin/bash` is now PID `1` inside the container, so it is possible to control the process using signals

Notes

ONBUILD Instruction

The format of the ONBUILD instruction is:

```
ONBUILD instruction
```

- The ONBUILD instruction sets a **deferred trigger** to execute a Dockerfile instruction **at a later point in time**
- The deferred instruction is executed in a **build** for an **image derived** from the current image
- ONBUILD instructions are executed at the same time as the **FROM instruction** is executed
- All Dockerfile instructions can be the recipient of the ONBUILD instruction, except the **FROM, MAINTAINER and ONBUILD** instructions

ONBUILD Instruction

- The `ONBUILD` instruction allows for specifying a Dockerfile instruction whose execution is deferred, and has the form:

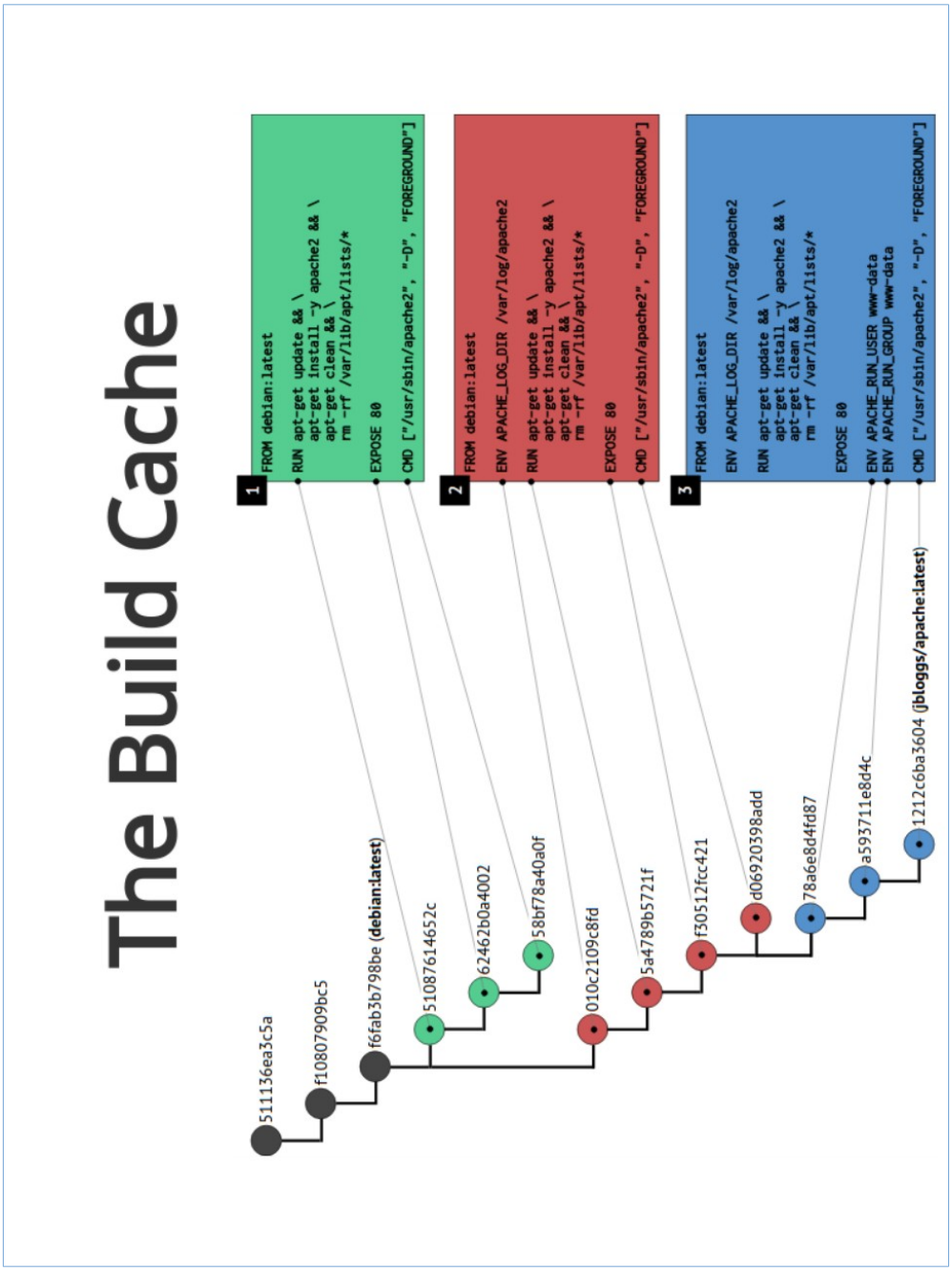
```
ONBUILD instruction
```

- Such an instruction is deferred for execution until a build takes place for an image that is derived from the base image, that contains the `ONBUILD` instruction
- This allows for creating images that behave in a similar fashion, but are dependent on specific data that is only available at a later date
- For example, it could be used for a generic application build environment, where source code needs to be made available via a `COPY` command (but is different from one application to another), and hence is unknown for the build of the generic, base image:

```
ONBUILD COPY . /go/src/app
```

- The Dockerfile instructions deferred in this manner are added to the image's metadata, and are triggered when a subsequent build of an image which uses the image as a base takes place
- The deferred instructions are executed in sequential order, as per their place in the base image history, as part of processing the `FROM` instruction in the derived image being built
- If any of the deferred instructions fail, the build of the derived image fails
- The deferred `ONBUILD` instructions can be examined with `docker inspect`, and will only apply to a direct descendent of the base image
- Environment variables can be used within an `ONBUILD` instruction, provided the variable is a component of the deferred instruction
- All Dockerfile instructions can be used with an `ONBUILD` instruction, except the `FROM` and `MAINTAINER` instructions, and the `ONBUILD` instruction itself

Notes


The Build Cache
511136ea3c5a
f10807909bc5
f6fab3b798be (debian:latest)
510876146 52c
62462b0a4002
58bf78a40a0f
010c2109c8fd
5a4789b5721f
f30512fcc421
d06920398add
78a6e8d4fd87
a593711e8d4c
1212c6ba3604 (jbloggs/apache:latest)
1
FROM debian:latest
RUN apt-get update && \
apt-get install -y apache2 && \
apt-get clean && \
rm -rf /var/lib/apt/lists/*
EXPOSE 80
CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
2
FROM debian:latest
ENV APACHE_LOG_DIR /var/log/apache2
RUN apt-get update && \
apt-get install -y apache2 && \
apt-get clean && \
rm -rf /var/lib/apt/lists/*
EXPOSE 80
CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
3
FROM debian:latest
ENV APACHE_LOG_DIR /var/log/apache2
RUN apt-get update && \
apt-get install -y apache2 && \
apt-get clean && \
rm -rf /var/lib/apt/lists/*
EXPOSE 80
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]

The Build Cache

- Some use cases of Docker, particularly relating to development, require a continual cycle of image building, and because lengthy build times don't aid productivity, Docker reuses existing image layers whenever it can, in order to optimise the build process
- The set of locally held image layers in the Docker host's local repositories, is called the build cache
- When Docker is able to reuse an image layer it informs as output during the build, e.g.,

```
Step 2 : COPY app.conf /etc
 ---> Using cache
 ---> 27c8d9b26d33
```

- An initial build of an image can take some time, but if changes are minimal thereafter, subsequent builds can take next to no time
- When a Dockerfile instruction is processed, Docker checks to see whether any sibling images (layers) were built using the exact same instruction, and if so, no new image is created
- If a new image is created, the cache is invalidated, and new images are created for every subsequent instruction
- Hence, if the instruction being processed is `EXPOSE 80`, and is also found to be the instruction used to build a previous child image of the parent image, the child image becomes part of the current image build
- The `COPY` and `ADD` instructions are treated slightly differently; as well as assessing the instruction similarity, Docker also checks (via a checksum) whether the file contents (including metadata) are the same , and if the content is identical, the cache is maintained (last accessed, and last modified times are ignored)
- It's tempting to assume that the contents after the execution of a `RUN` instruction will also be assessed, but this is not the case
- Use of the `ARG` instruction will invalidate the cache if the value passed in changes from one build to the next
- The build cache can be by-passed with the use of the `--no-cache` configuration option of the `docker build` command

Notes