Week 2, Web Application Development

Azamat Serek, PhD, Assist.Prof.

Dockerfile

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

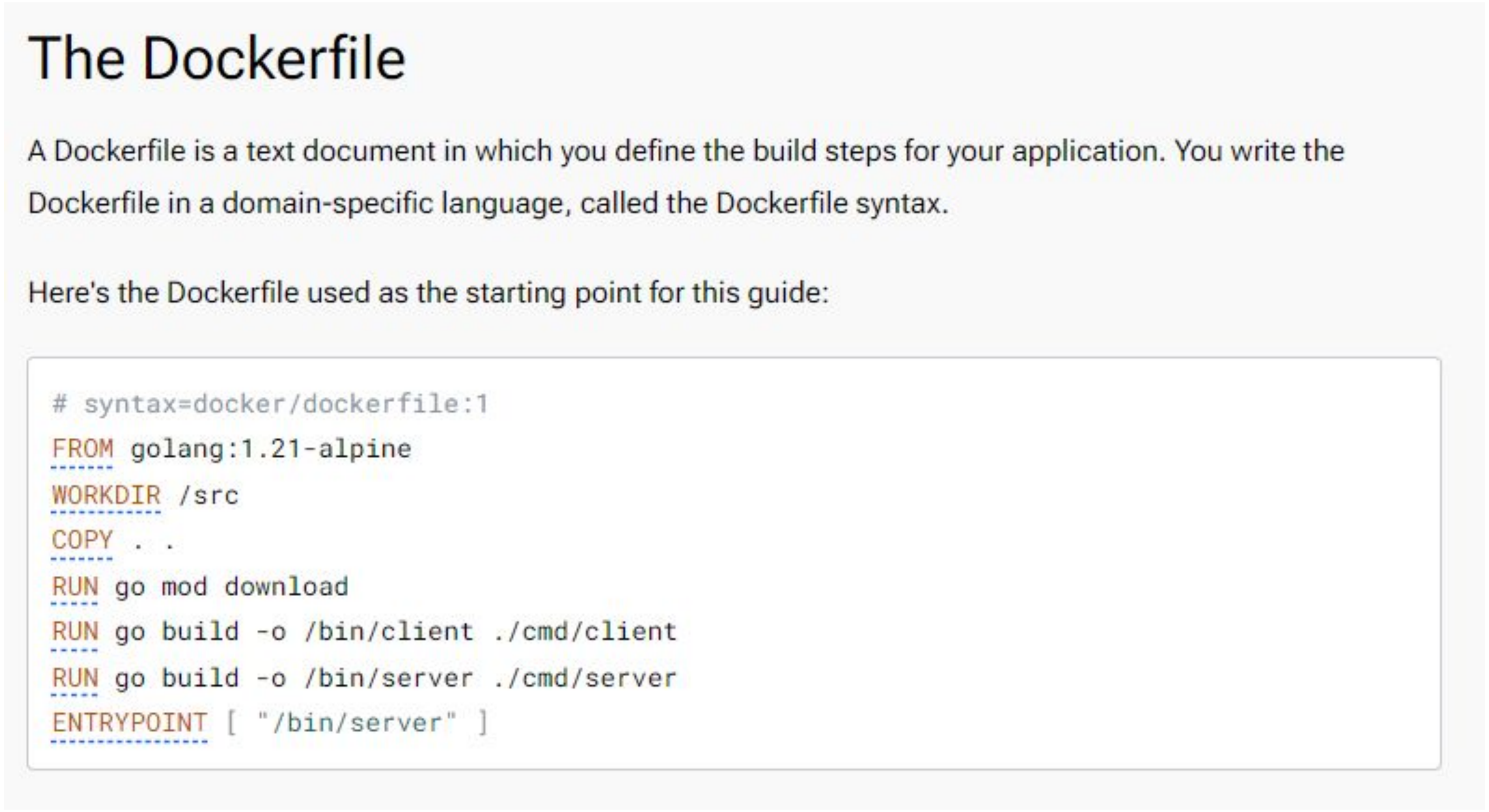
The application

The example project for this guide is a client-server application for translating messages to a fictional language.

Here's an overview of the files included in the project:

```
.
├── Dockerfile
├── cmd
│   ├── client
│   │   ├── main.go
│   │   ├── request.go
│   │   └── ui.go
│   └── server
│       ├── main.go
│       └── translate.go
├── go.mod
└── go.sum
```

The `cmd/` directory contains the code for the two application components: client and server. The client is a user interface for writing, sending, and receiving messages. The server receives messages from clients, translates them, and sends them back to the client.

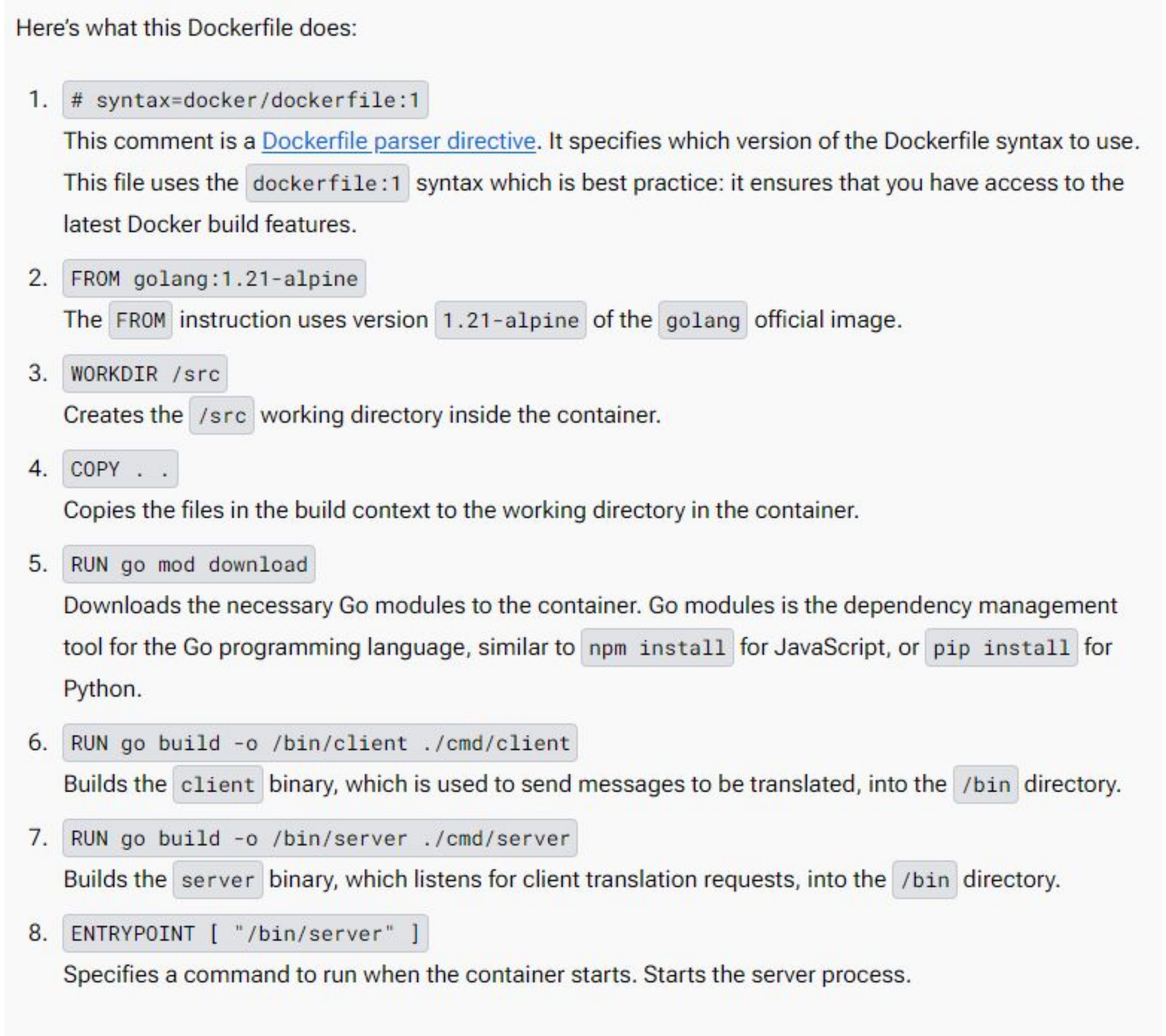


The Dockerfile

A Dockerfile is a text document in which you define the build steps for your application. You write the Dockerfile in a domain-specific language, called the Dockerfile syntax.

Here's the Dockerfile used as the starting point for this guide:

```dockerfile
# syntax=docker/dockerfile:1
FROM golang:1.21-alpine
WORKDIR /src
COPY . .
RUN go mod download
RUN go build -o /bin/client ./cmd/client
RUN go build -o /bin/server ./cmd/server
ENTRYPOINT [ "/bin/server" ]
```



Here's what this Dockerfile does:

1. `# syntax=docker/dockerfile:1`

   This comment is a Dockerfile parser directive. It specifies which version of the Dockerfile syntax to use. This file uses the `dockerfile:1` syntax which is best practice: it ensures that you have access to the latest Docker build features.

2. `FROM golang:1.21-alpine`

   The `FROM` instruction uses version `1.21-alpine` of the `golang` official image.

3. `WORKDIR /src`

   Creates the `/src` working directory inside the container.

4. `COPY . .`

   Copies the files in the build context to the working directory in the container.

5. `RUN go mod download`

   Downloads the necessary Go modules to the container. Go modules is the dependency management tool for the Go programming language, similar to `npm install` for JavaScript, or `pip install` for Python.

6. `RUN go build -o /bin/client ./cmd/client`

   Builds the `client` binary, which is used to send messages to be translated, into the `/bin` directory.

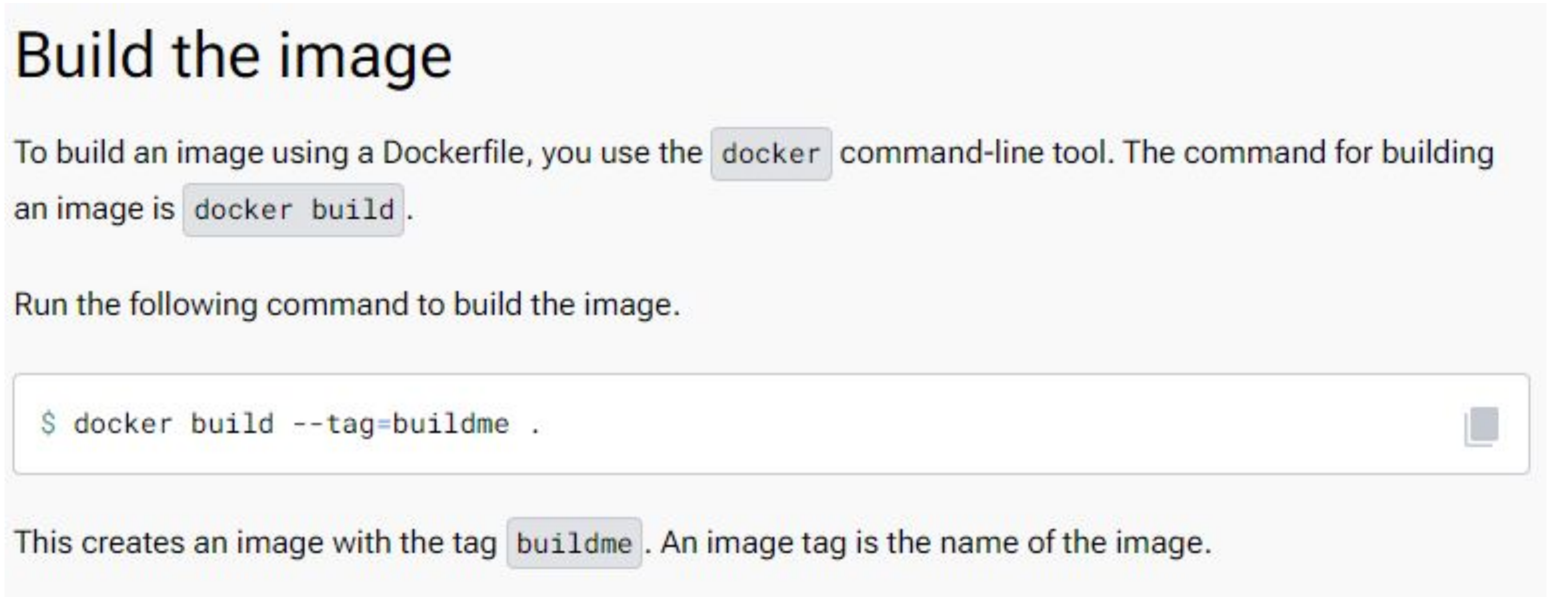
7. `RUN go build -o /bin/server ./cmd/server`

   Builds the `server` binary, which listens for client translation requests, into the `/bin` directory.

8. `ENTRYPOINT [ "/bin/server" ]`

   Specifies a command to run when the container starts. Starts the server process.

Building image



Build the image

To build an image using a Dockerfile, you use the `docker` command-line tool. The command for building an image is `docker build`.

Run the following command to build the image.

```
$ docker build --tag=buildme .
```

This creates an image with the tag `buildme`. An image tag is the name of the image.

Run the container

The image you just built contains two binaries, one for the server and one for the client. To see the translation service in action, run a container that hosts the server component, and then run another container that invokes the client.

To run a container, you use the `docker run` command.

1. Run a container from the image in detached mode.

   ```
   $ docker run --name=buildme --rm --detach buildme
   ```

   This starts a container named `buildme`.

2. Run a new command in the `buildme` container that invokes the client binary.

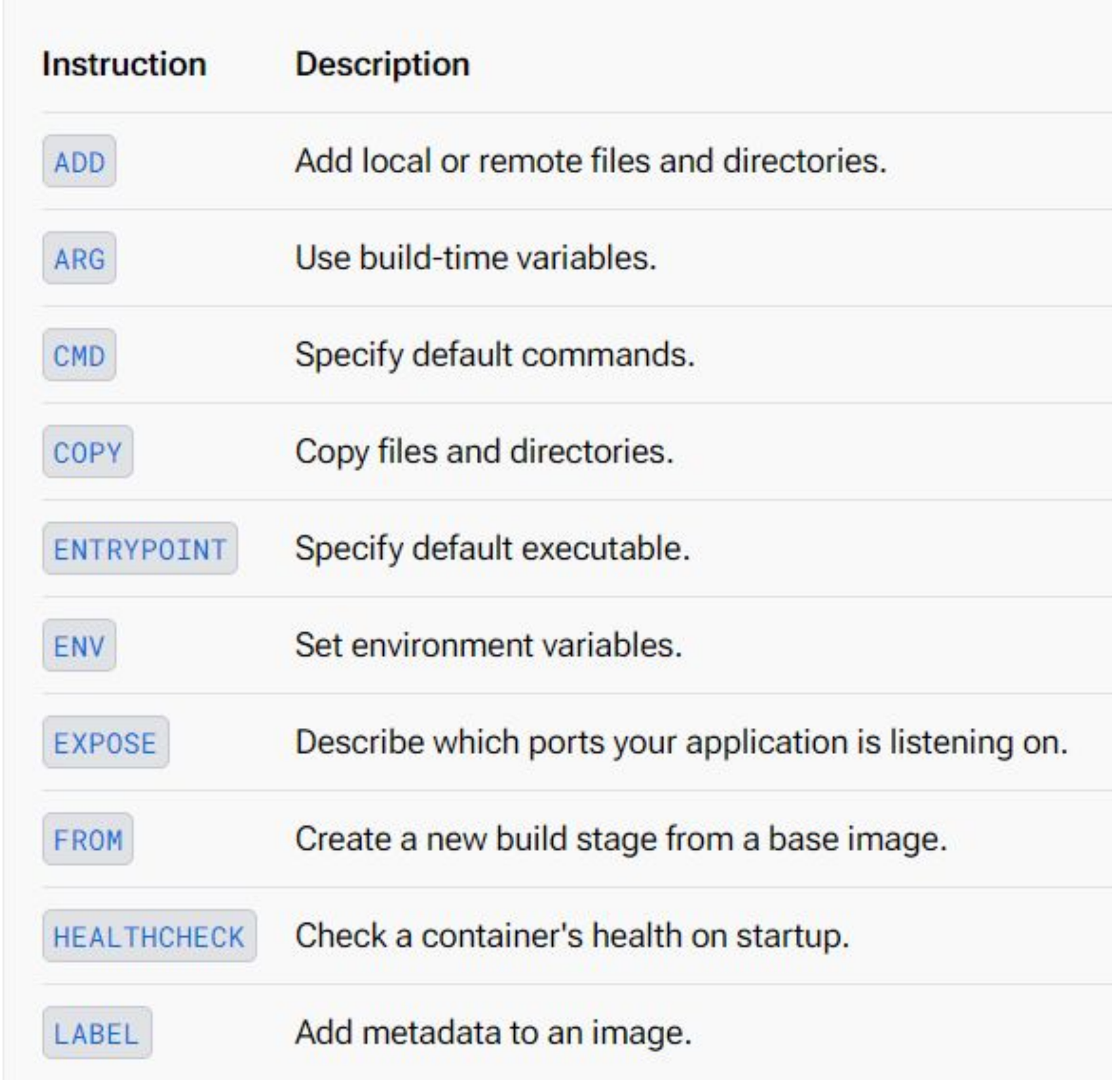
   ```
   $ docker exec -it buildme /bin/client
   ```

The `docker exec` command opens a terminal user interface where you can submit messages for the backend (server) process to translate.

When you're done testing, you can stop the container:

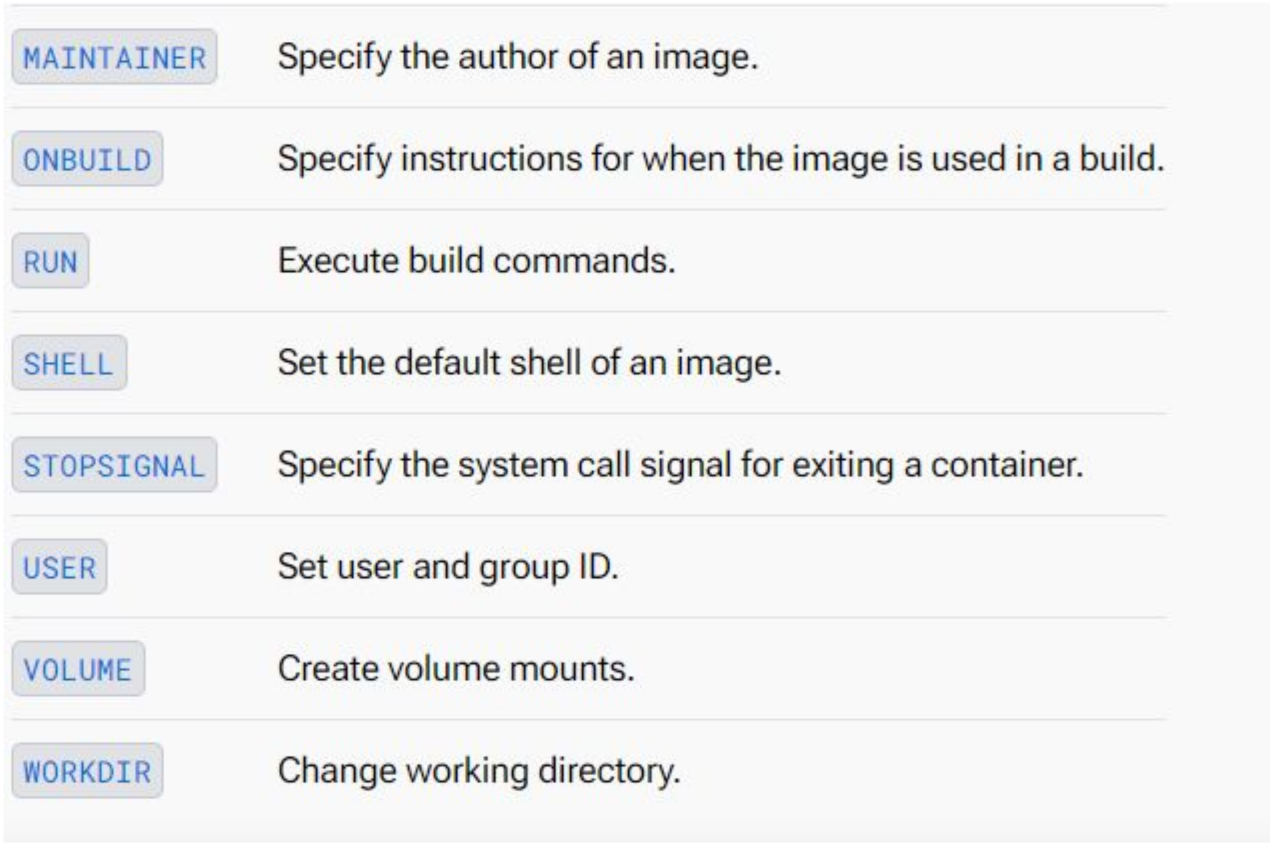
```
$ docker stop buildme
```

Dockerfile Instructions

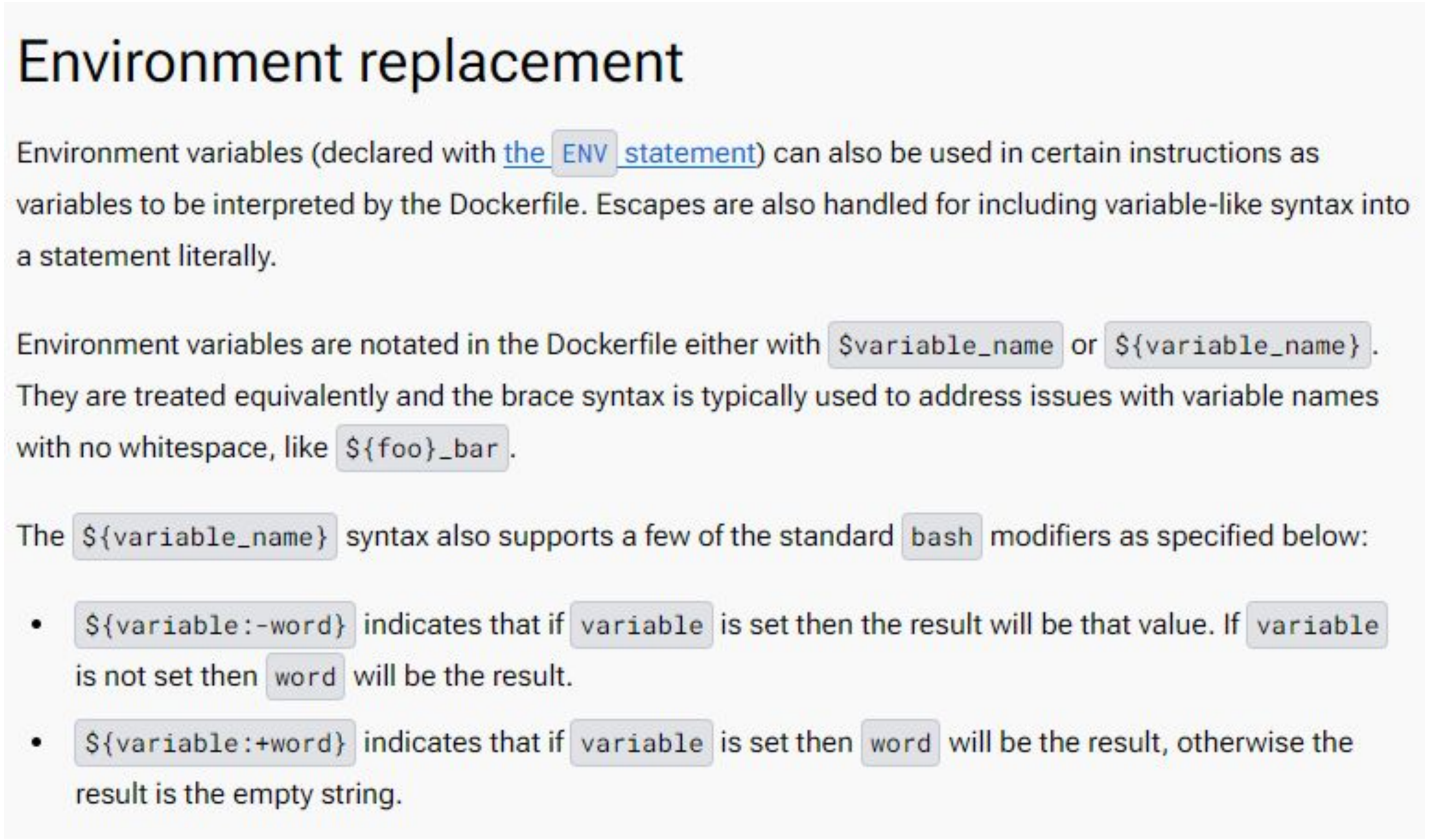


Instruction	Description
`ADD`	Add local or remote files and directories.
`ARG`	Use build-time variables.
`CMD`	Specify default commands.
`COPY`	Copy files and directories.
`ENTRYPOINT`	Specify default executable.
`ENV`	Set environment variables.
`EXPOSE`	Describe which ports your application is listening on.
`FROM`	Create a new build stage from a base image.
`HEALTHCHECK`	Check a container's health on startup.
`LABEL`	Add metadata to an image.

Cont.



`MAINTAINER`	Specify the author of an image.
`ONBUILD`	Specify instructions for when the image is used in a build.
`RUN`	Execute build commands.
`SHELL`	Set the default shell of an image.
`STOPSIGNAL`	Specify the system call signal for exiting a container.
`USER`	Set user and group ID.
`VOLUME`	Create volume mounts.
`WORKDIR`	Change working directory.



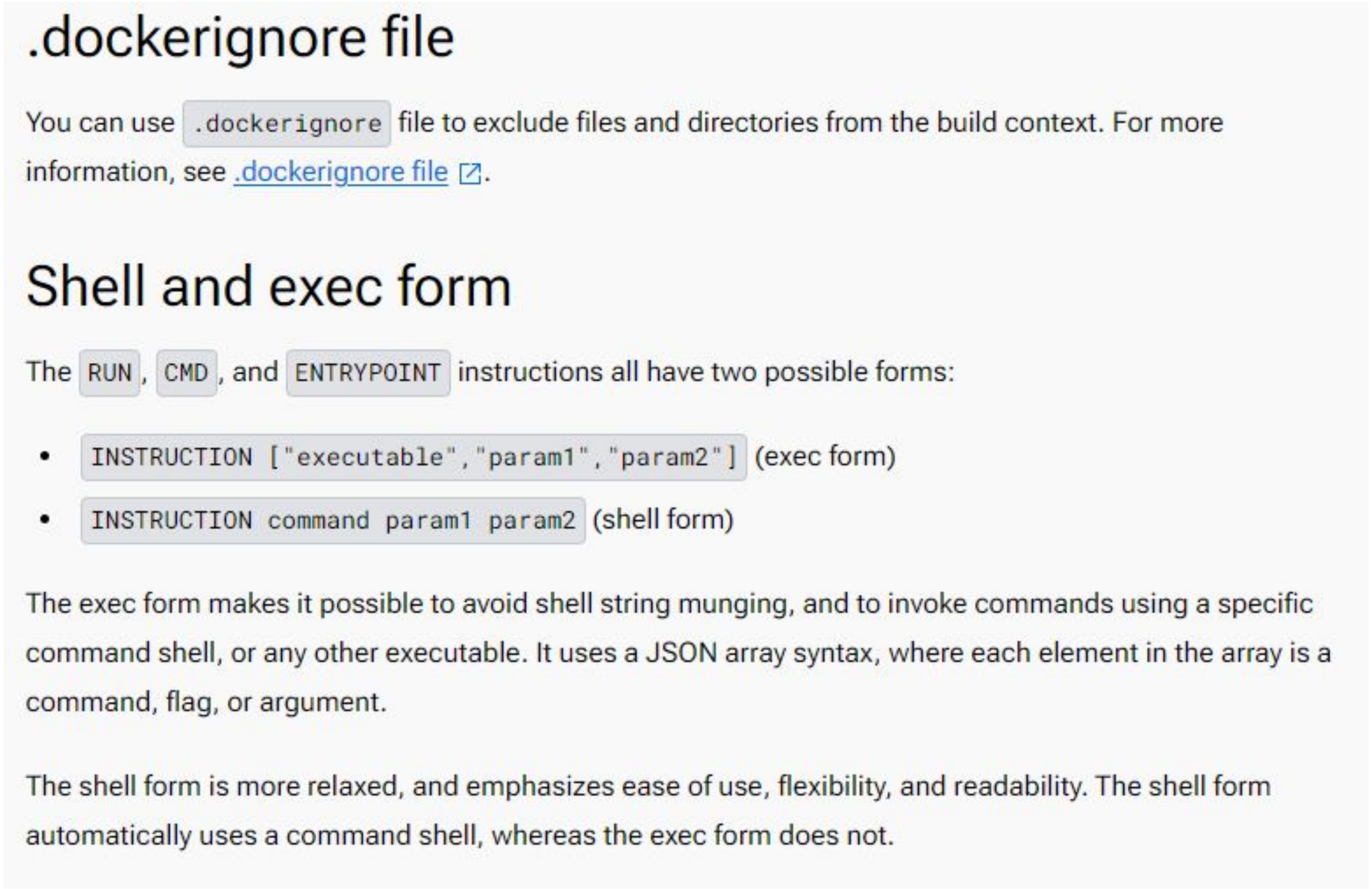
Environment replacement

Environment variables (declared with the `ENV` statement) can also be used in certain instructions as variables to be interpreted by the Dockerfile. Escapes are also handled for including variable-like syntax into a statement literally.

Environment variables are notated in the Dockerfile either with `$variable_name` or `${variable_name}`. They are treated equivalently and the brace syntax is typically used to address issues with variable names with no whitespace, like `${foo}_bar`.

The `${variable_name}` syntax also supports a few of the standard `bash` modifiers as specified below:

- `${variable:-word}` indicates that if `variable` is set then the result will be that value. If `variable` is not set then `word` will be the result.
- `${variable:+word}` indicates that if `variable` is set then `word` will be the result, otherwise the result is the empty string.



.dockerignore file

You can use `.dockerignore` file to exclude files and directories from the build context. For more information, see .dockerignore file.

Shell and exec form

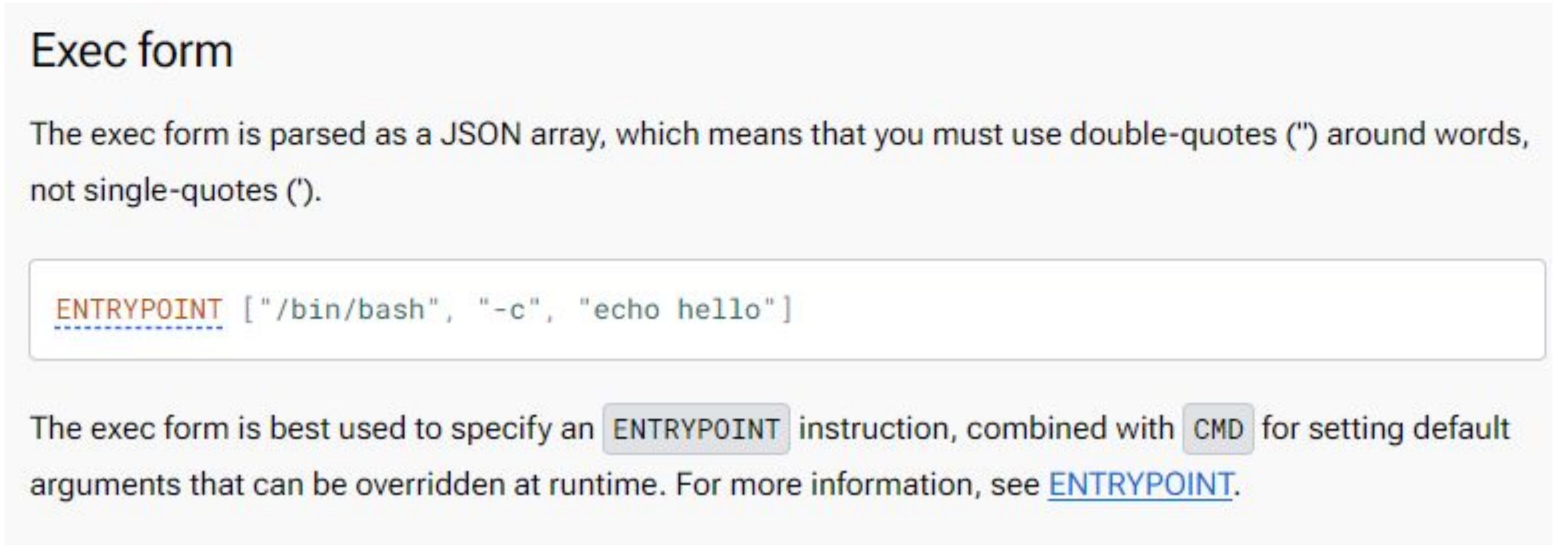
The `RUN`, `CMD`, and `ENTRYPOINT` instructions all have two possible forms:

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

The exec form makes it possible to avoid shell string munging, and 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.

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.

Exec form



Exec form

The exec form is parsed as a JSON array, which means that you must use double-quotes (") around words, not single-quotes (').

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

The exec form is best used to specify an `ENTRYPOINT` instruction, combined with `CMD` for setting default arguments that can be overridden at runtime. For more information, see ENTRYPOINT.

Understand how ARG and FROM interact

`FROM` instructions support variables that are declared by any `ARG` instructions that occur before the first `FROM`.

```
ARG  CODE_VERSION=latest
FROM base:${CODE_VERSION}
CMD  /code/run-app

FROM extras:${CODE_VERSION}
CMD  /code/run-extras
```

An `ARG` declared before a `FROM` is outside of a build stage, so it can't be used in any instruction after a `FROM`. To use the default value of an `ARG` declared before the first `FROM` use an `ARG` instruction without a value inside of a build stage:

```
ARG VERSION=latest
FROM busybox:$VERSION
ARG VERSION
RUN echo $VERSION > image_version
```

RUN

The `RUN` instruction will execute any commands to create a new layer on top of the current image. The added layer is used in the next step in the Dockerfile. `RUN` has two forms:

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

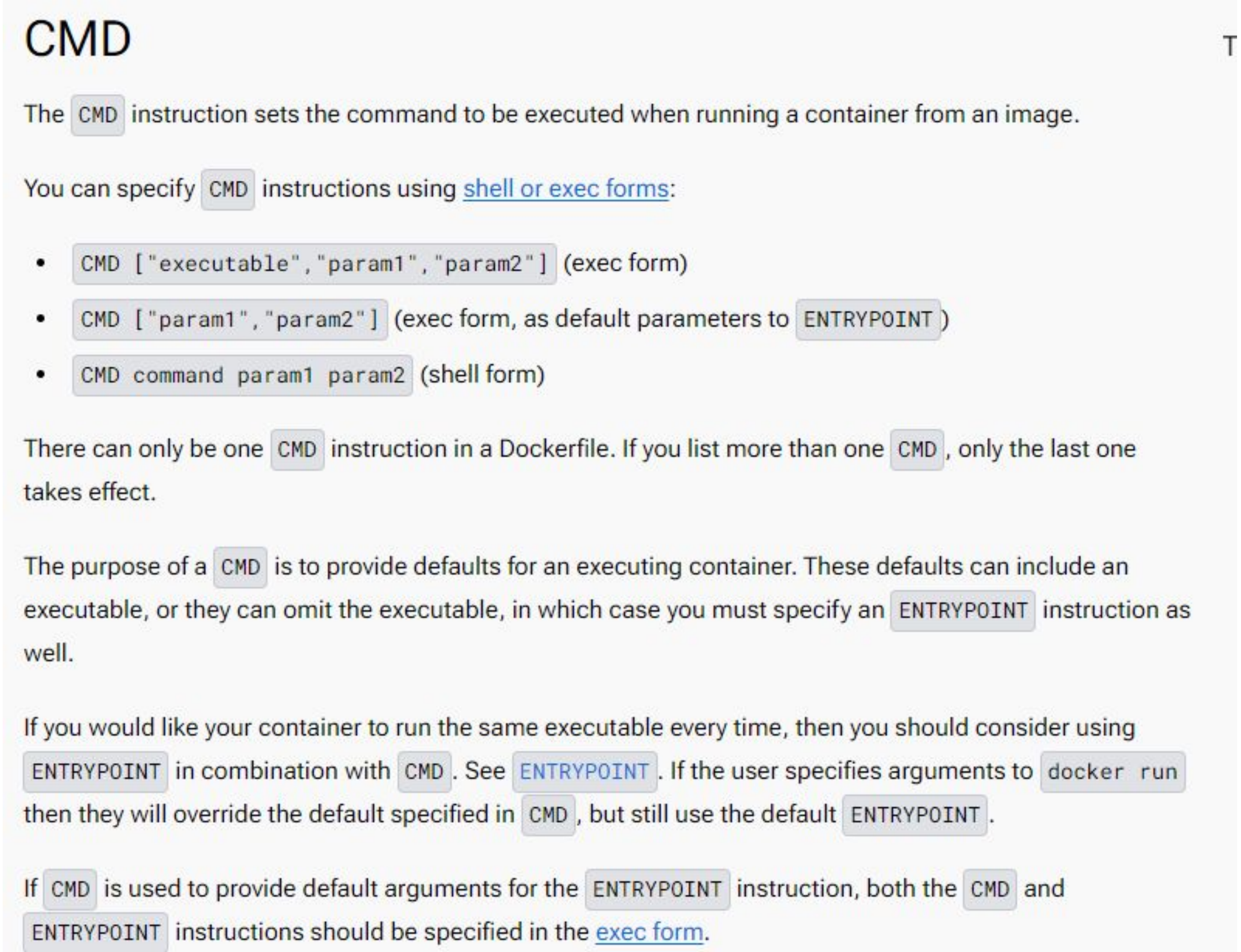
For more information about the differences between these two forms, see shell or exec forms.

The shell form is most commonly used, and lets you break up longer instructions into multiple lines, either using newline escapes, or with heredocs:

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

The available `[OPTIONS]` for the `RUN` instruction are:

- `--mount`
- `--network`
- `--security`



CMD

The `CMD` instruction sets the command to be executed when running a container from an image.

You can specify `CMD` instructions using shell or exec forms:

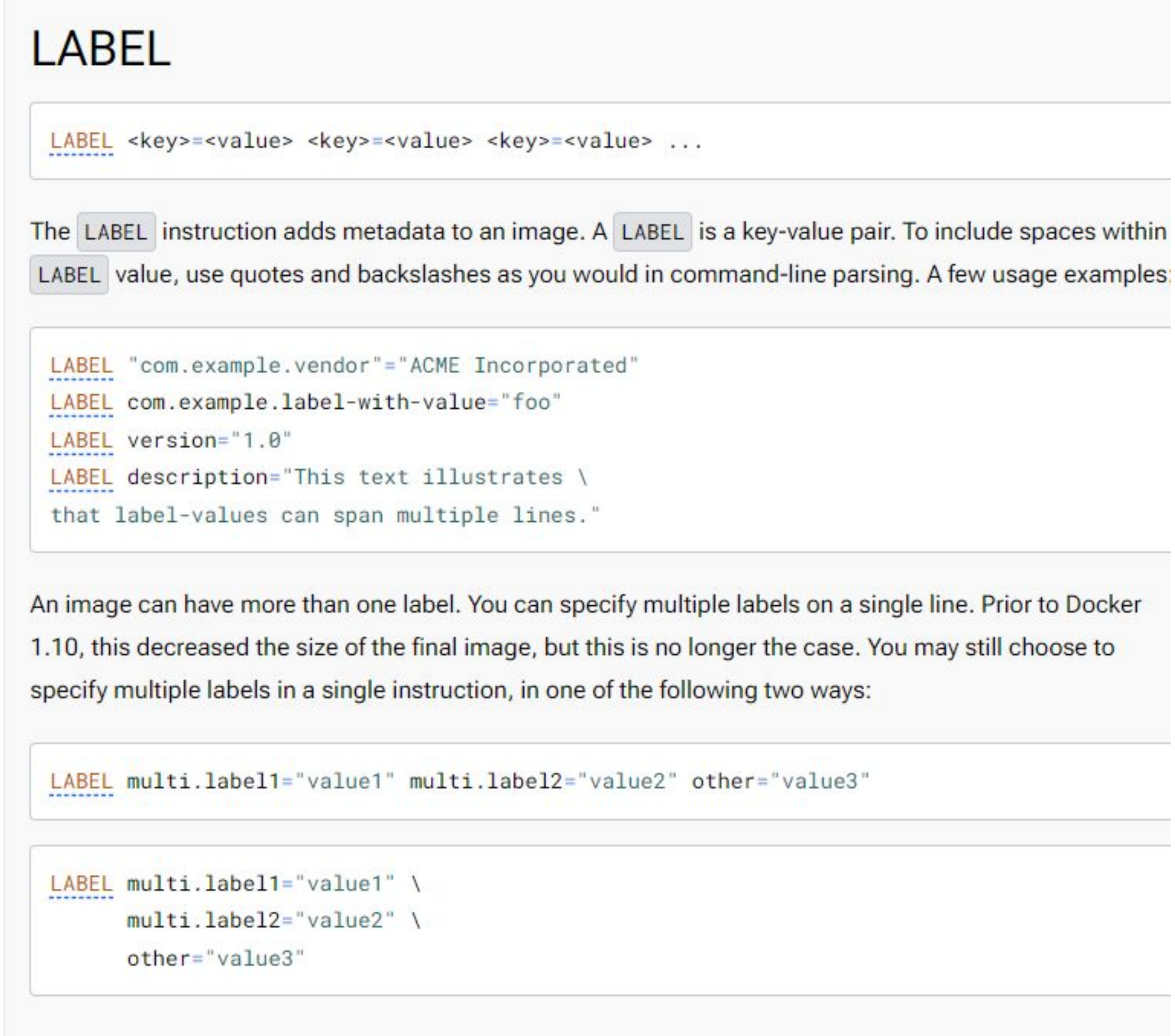
- `CMD ["executable","param1","param2"]` (exec form)
- `CMD ["param1","param2"]` (exec form, as default parameters to `ENTRYPOINT`)
- `CMD command param1 param2` (shell form)

There can only be one `CMD` instruction in a Dockerfile. If you list more than one `CMD`, only the last one takes effect.

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 you would like your container to run the same executable every time, then you should consider using `ENTRYPOINT` in combination with `CMD`. See `ENTRYPOINT`. If the user specifies arguments to `docker run` then they will override the default specified in `CMD`, but still use the default `ENTRYPOINT`.

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.



LABEL

```
LABEL <key>=<value> <key>=<value> <key>=<value> ...
```

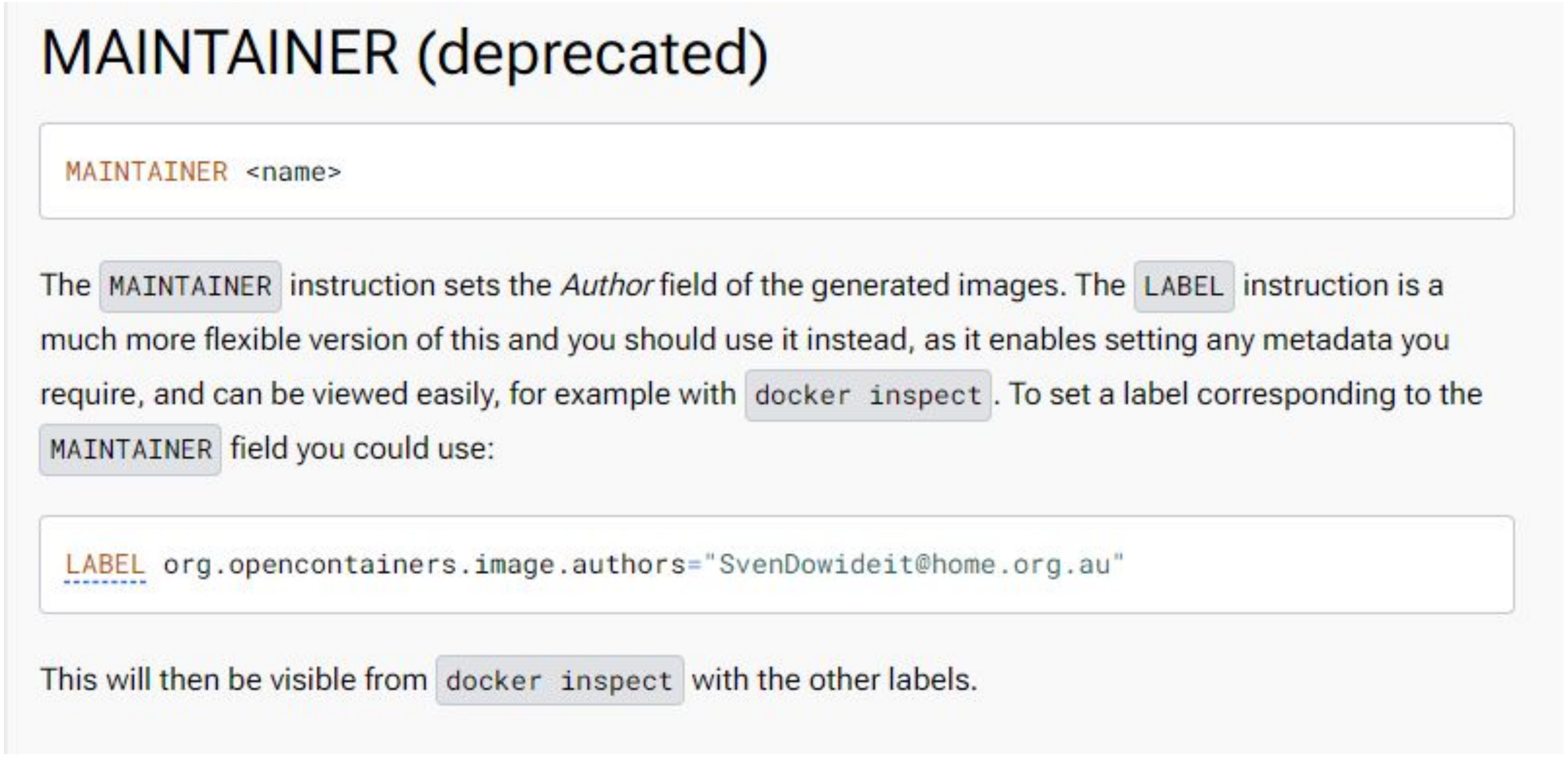
The `LABEL` instruction adds metadata to an image. A `LABEL` is a key-value pair. To include spaces within `LABEL` value, use quotes and backslashes as you would in command-line parsing. A few usage examples

```
LABEL "com.example.vendor"="ACME Incorporated"
LABEL com.example.label-with-value="foo"
LABEL version="1.0"
LABEL description="This text illustrates \
that label-values can span multiple lines."
```

An image can have more than one label. You can specify multiple labels on a single line. Prior to Docker 1.10, this decreased the size of the final image, but this is no longer the case. You may still choose to specify multiple labels in a single instruction, in one of the following two ways:

```
LABEL multi.label1="value1" multi.label2="value2" other="value3"
```

```
LABEL multi.label1="value1" \
      multi.label2="value2" \
      other="value3"
```



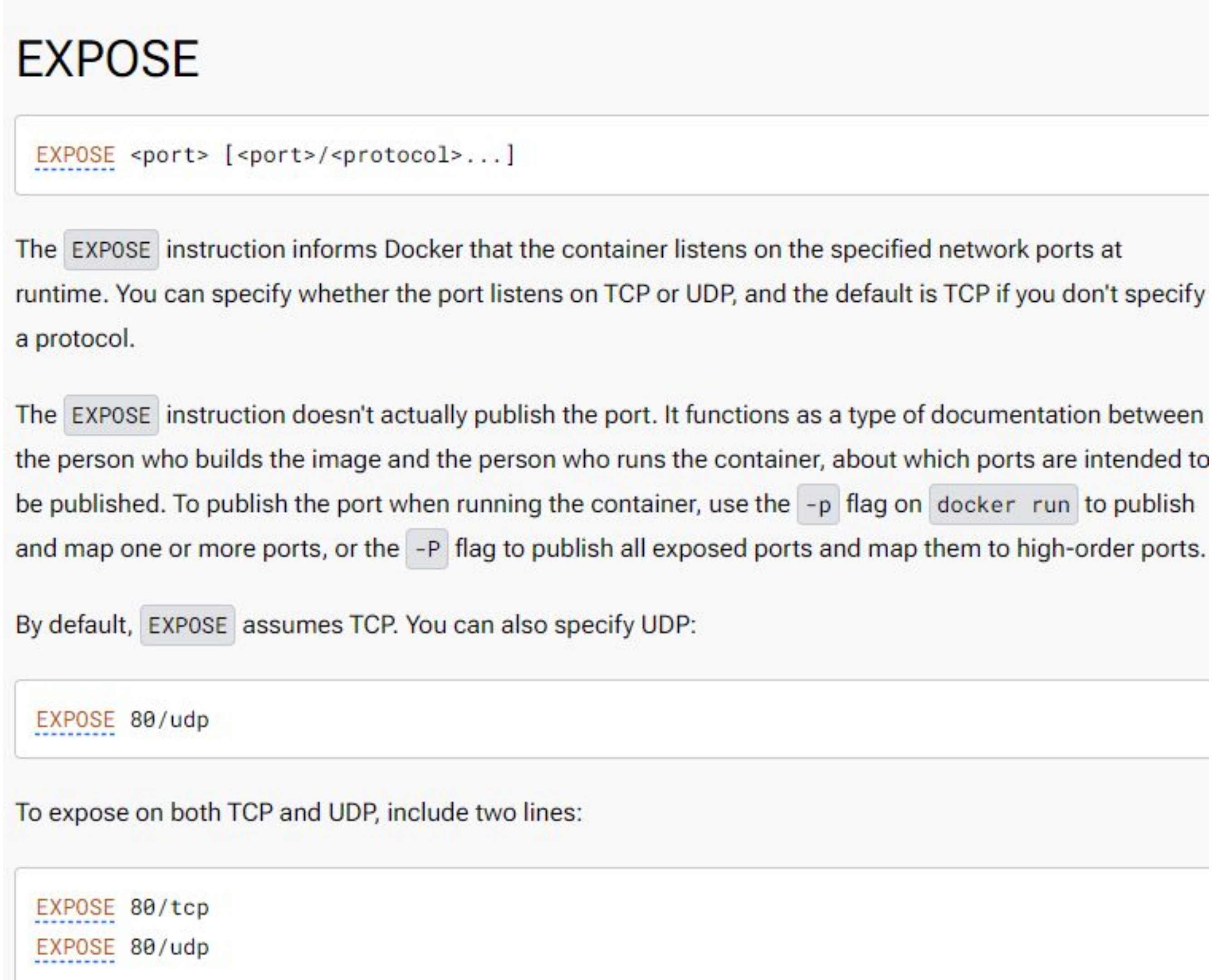
MAINTAINER (deprecated)

```
MAINTAINER <name>
```

The `MAINTAINER` instruction sets the *Author* field of the generated images. The `LABEL` instruction is a much more flexible version of this and you should use it instead, as it enables setting any metadata you require, and can be viewed easily, for example with `docker inspect`. To set a label corresponding to the `MAINTAINER` field you could use:

```
LABEL org.opencontainers.image.authors="SvenDowideit@home.org.au"
```

This will then be visible from `docker inspect` with the other labels.



EXPOSE

```
EXPOSE <port> [<port>/<protocol>...]
```

The `EXPOSE` instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the default is TCP if you don't specify a protocol.

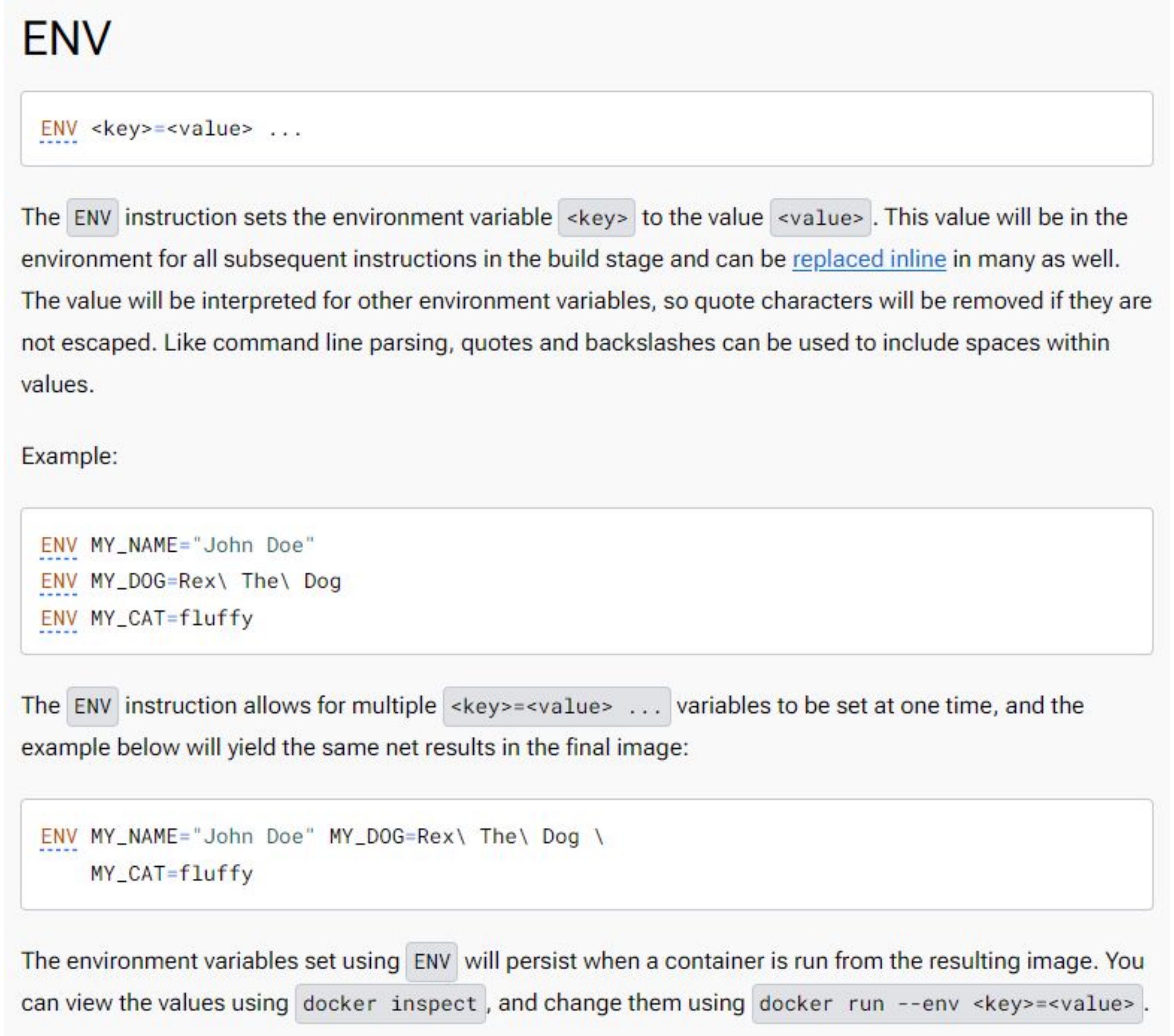
The `EXPOSE` instruction doesn't actually publish the port. It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published. To publish the port when running the container, use the `-p` flag on `docker run` to publish and map one or more ports, or the `-P` flag to publish all exposed ports and map them to high-order ports.

By default, `EXPOSE` assumes TCP. You can also specify UDP:

```
EXPOSE 80/udp
```

To expose on both TCP and UDP, include two lines:

```
EXPOSE 80/tcp
EXPOSE 80/udp
```



ENV

```
ENV <key>=<value> ...
```

The `ENV` instruction sets the environment variable `<key>` to the value `<value>`. This value will be in the environment for all subsequent instructions in the build stage and can be replaced inline in many as well. The value will be interpreted for other environment variables, so quote characters will be removed if they are not escaped. Like command line parsing, quotes and backslashes can be used to include spaces within values.

Example:

```
ENV MY_NAME="John Doe"
ENV MY_DOG=Rex\ The\ Dog
ENV MY_CAT=fluffy
```

The `ENV` instruction allows for multiple `<key>=<value> ...` variables to be set at one time, and the example below will yield the same net results in the final image:

```
ENV MY_NAME="John Doe" MY_DOG=Rex\ The\ Dog \
    MY_CAT=fluffy
```

The environment variables set using `ENV` will persist when a container is run from the resulting image. You can view the values using `docker inspect`, and change them using `docker run --env <key>=<value>`.

ADD

ADD has two forms. The latter form is required for paths containing whitespace.

```
ADD [OPTIONS] <src> ... <dest>
ADD [OPTIONS] ["<src>", ... "<dest>"]
```

The available `[OPTIONS]` are:

- `--keep-git-dir`
- `--checksum`
- `--chown`
- `--chmod`
- `--link`
- `--exclude`

The `ADD` instruction copies new files, directories or remote file URLs from `<src>` and adds them to the filesystem of the image at the path `<dest>`.

Each `<src>` may contain wildcards and matching will be done using Go's filepath.Match rules. For example:

To add all files in the root of the build context starting with "hom":

```
ADD hom* /mydir/
```

In the following example, `?` is a single-character wildcard, matching e.g. "home.txt".

```
ADD hom?.txt /mydir/
```

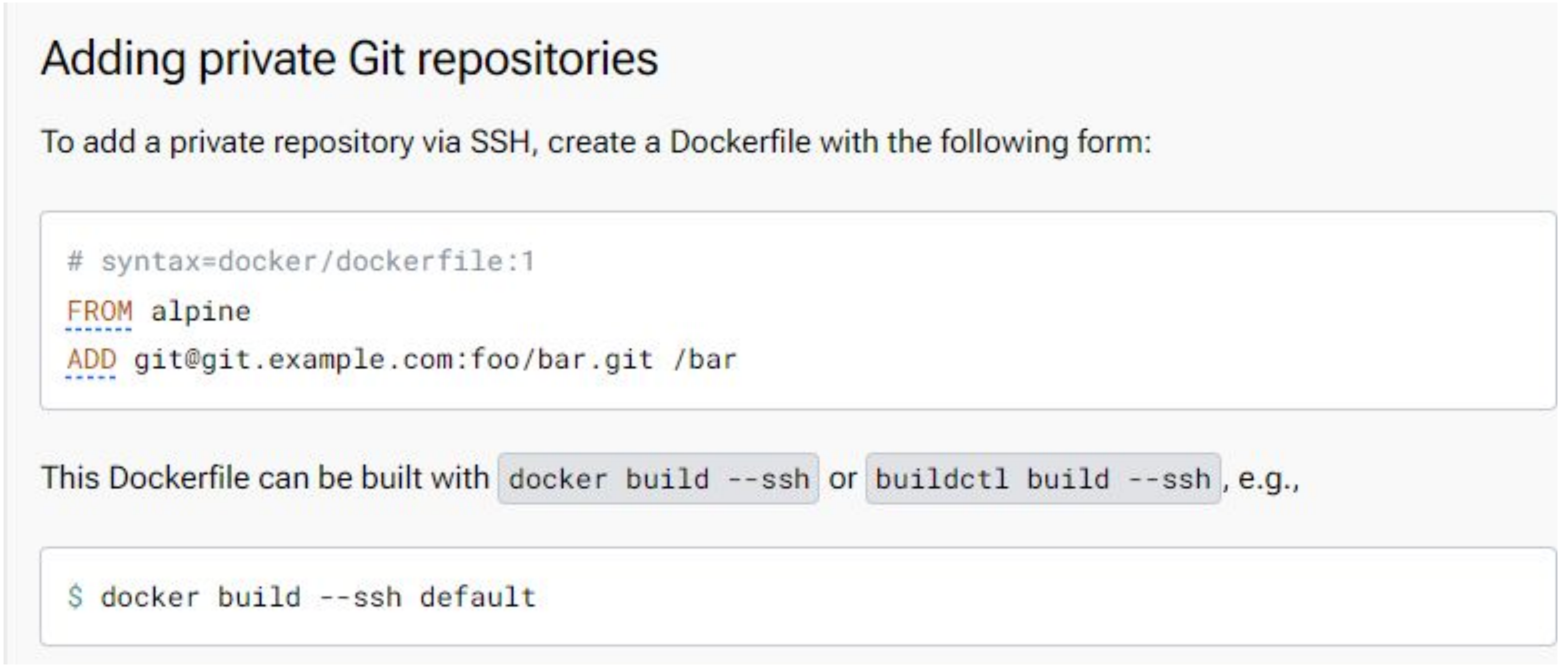
The `<dest>` is an absolute path, or a path relative to `WORKDIR`, into which the source will be copied inside the destination container.

The example below uses a relative path, and adds "test.txt" to `<WORKDIR>/relativeDir/`:

```
ADD test.txt relativeDir/
```

Whereas this example uses an absolute path, and adds "test.txt" to `/absoluteDir/`

```
ADD test.txt /absoluteDir/
```



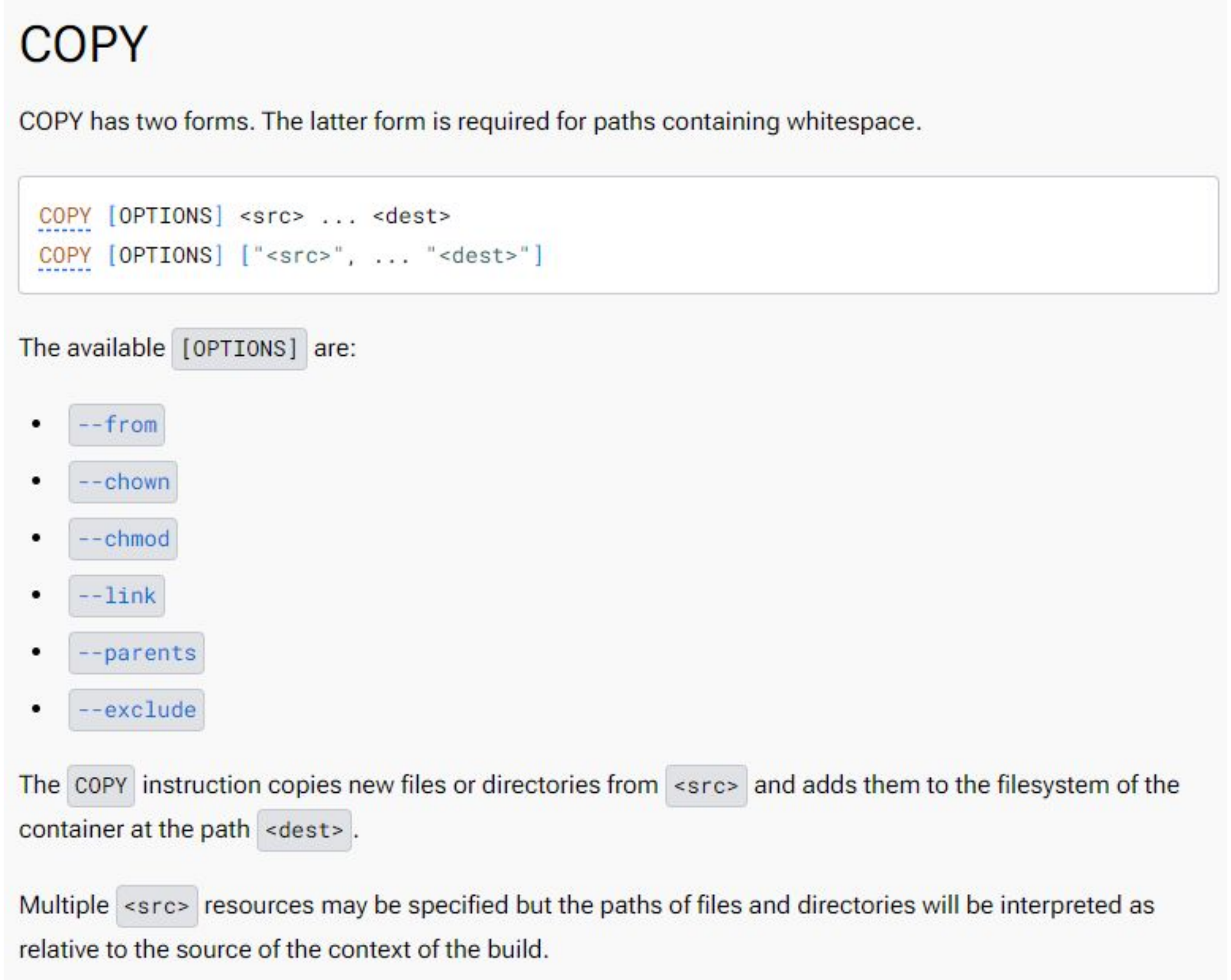
Adding private Git repositories

To add a private repository via SSH, create a Dockerfile with the following form:

```
# syntax=docker/dockerfile:1
FROM alpine
ADD git@git.example.com:foo/bar.git /bar
```

This Dockerfile can be built with `docker build --ssh` or `buildctl build --ssh`, e.g.,

```
$ docker build --ssh default
```



COPY

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

```
COPY [OPTIONS] <src> ... <dest>
COPY [OPTIONS] ["<src>", ... "<dest>"]
```

The available `[OPTIONS]` are:

- `--from`
- `--chown`
- `--chmod`
- `--link`
- `--parents`
- `--exclude`

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.

COPY --from

By default, the `COPY` instruction copies files from the build context. The `COPY --from` flag lets you copy files from an image, a build stage, or a named context instead.

```
COPY [--from=<image|stage|context>] <src> ... <dest>
```

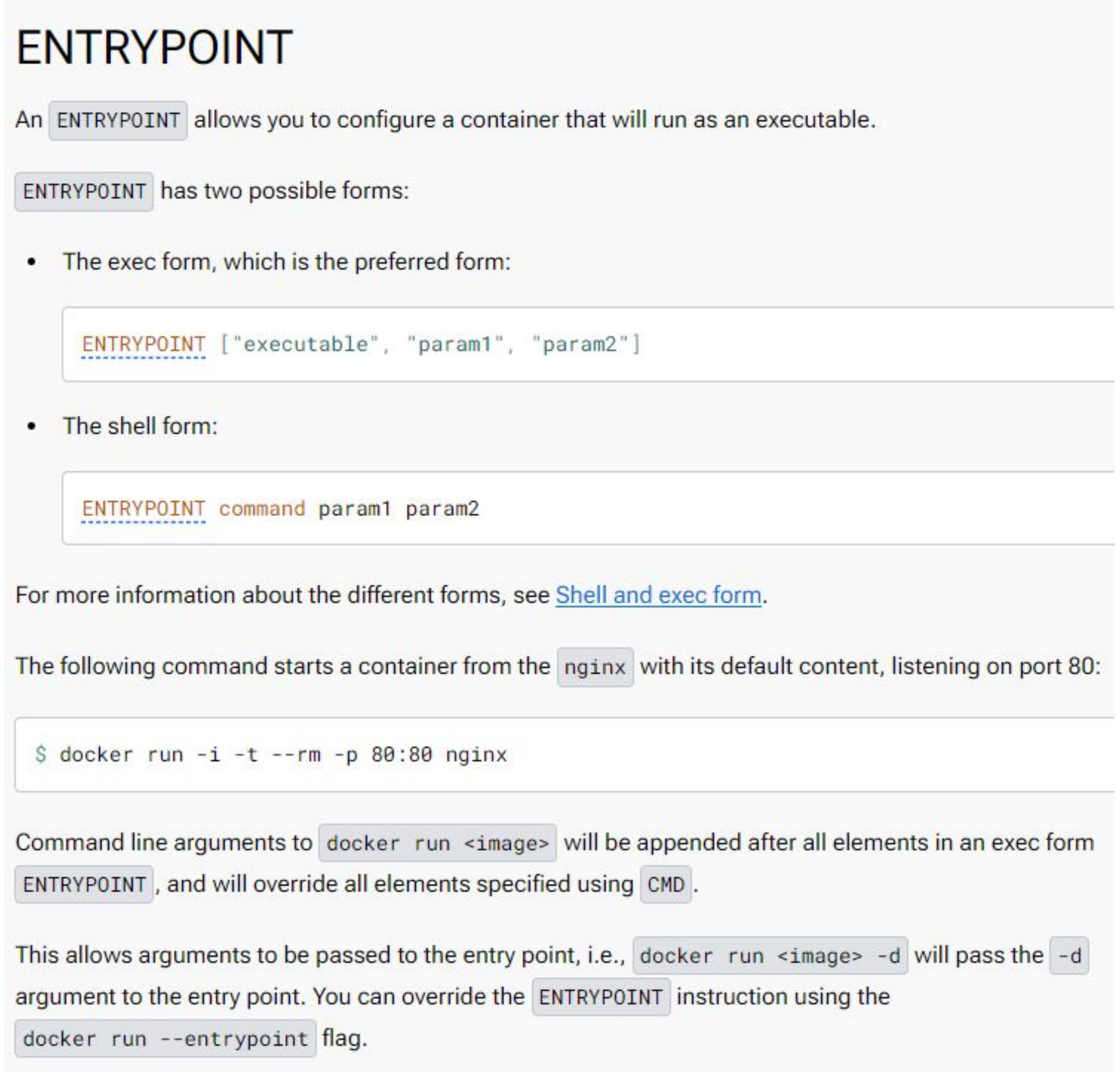
To copy from a build stage in a multi-stage build, specify the name of the stage you want to copy from. You specify stage names using the `AS` keyword with the `FROM` instruction.

```
# syntax=docker/dockerfile:1
FROM alpine AS build
COPY . .
RUN apk add clang
RUN clang -o /hello hello.c

FROM scratch
COPY --from=build /hello /
```

You can also copy files directly from other images. The following example copies an `nginx.conf` file from the official Nginx image.

```
COPY --from=nginx:latest /etc/nginx/nginx.conf /nginx.conf
```



ENTRYPOINT

An `ENTRYPOINT` allows you to configure a container that will run as an executable.

`ENTRYPOINT` has two possible forms:

- The exec form, which is the preferred form:

  ```
  ENTRYPOINT ["executable", "param1", "param2"]
  ```

- The shell form:

  ```
  ENTRYPOINT command param1 param2
  ```

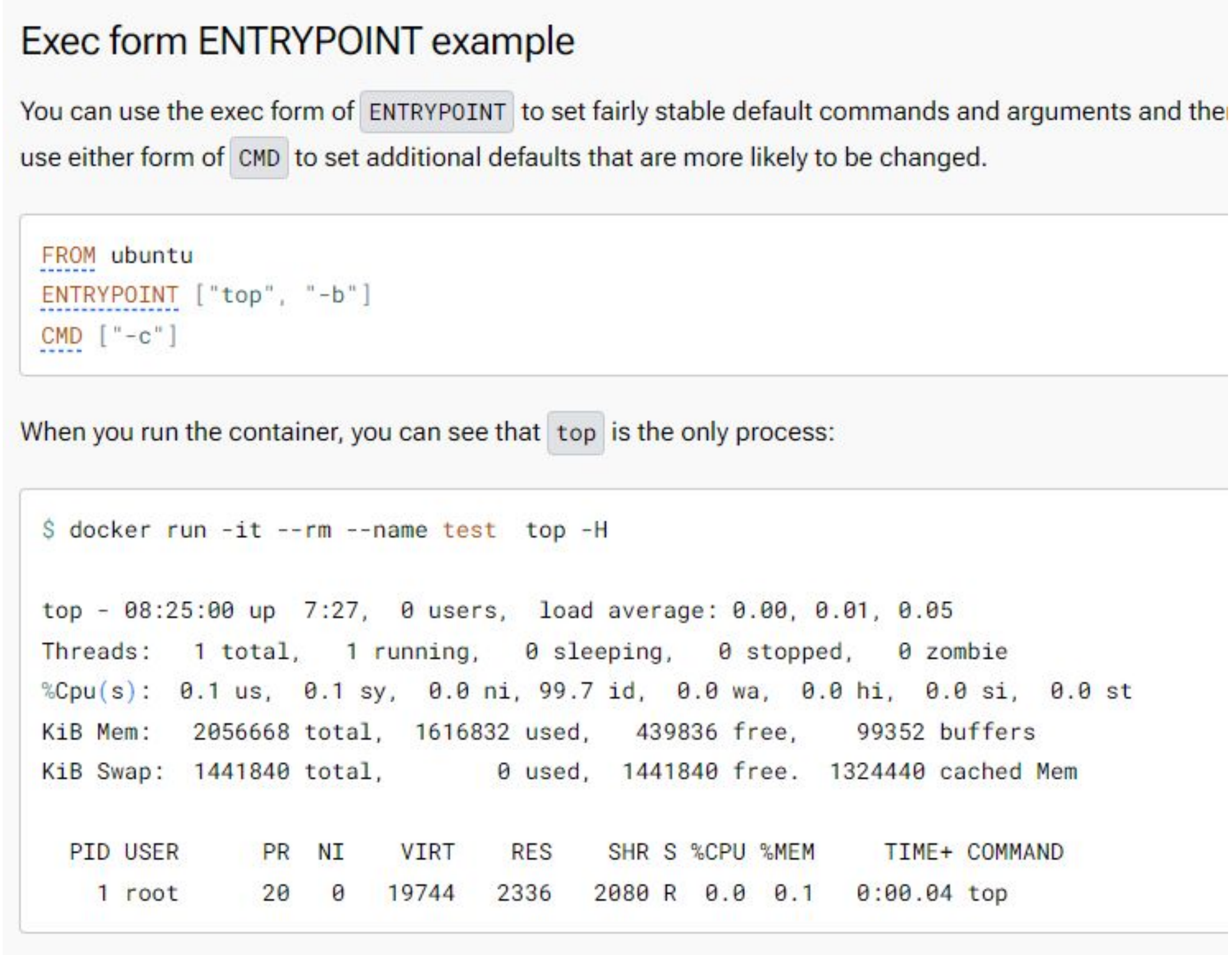
For more information about the different forms, see Shell and exec form.

The following command starts a container from the `nginx` with its default content, listening on port 80:

```
$ docker run -i -t --rm -p 80:80 nginx
```

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`.

This allows arguments to be passed to the entry point, i.e., `docker run <image> -d` will pass the `-d` argument to the entry point. You can override the `ENTRYPOINT` instruction using the `docker run --entrypoint` flag.



Exec form ENTRYPOINT example

You can use the exec form of `ENTRYPOINT` to set fairly stable default commands and arguments and the
use either form of `CMD` to set additional defaults that are more likely to be changed.

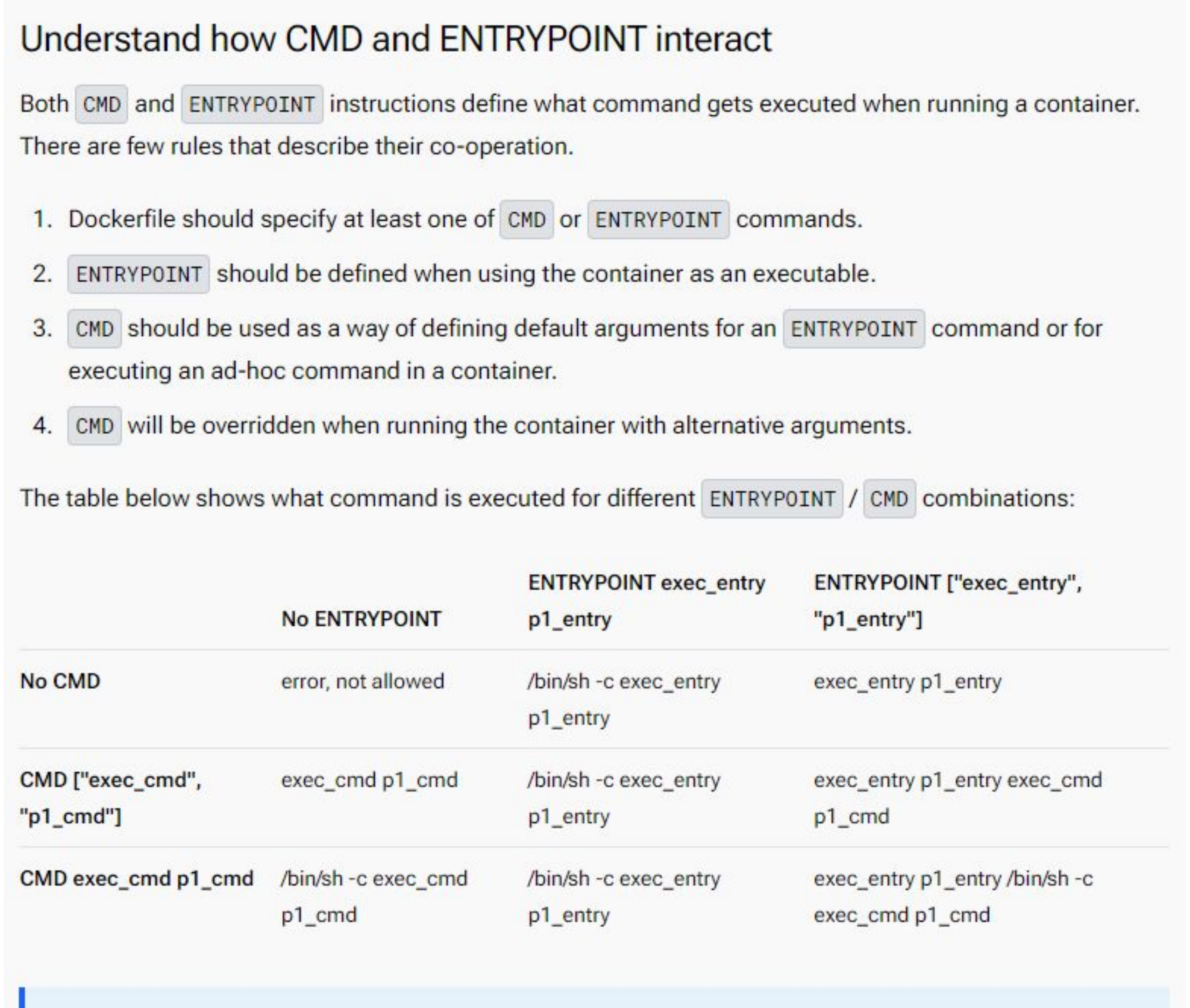
```
FROM ubuntu
ENTRYPOINT ["top", "-b"]
CMD ["-c"]
```

When you run the container, you can see that `top` is the only process:

```
$ docker run -it --rm --name test  top -H

top - 08:25:00 up  7:27,  0 users,  load average: 0.00, 0.01, 0.05
Threads:   1 total,   1 running,   0 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.1 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   2056668 total,  1616832 used,   439836 free,    99352 buffers
KiB Swap:  1441840 total,        0 used,  1441840 free.  1324440 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
    1 root      20   0   19744   2336   2080 R  0.0  0.1   0:00.04 top
```



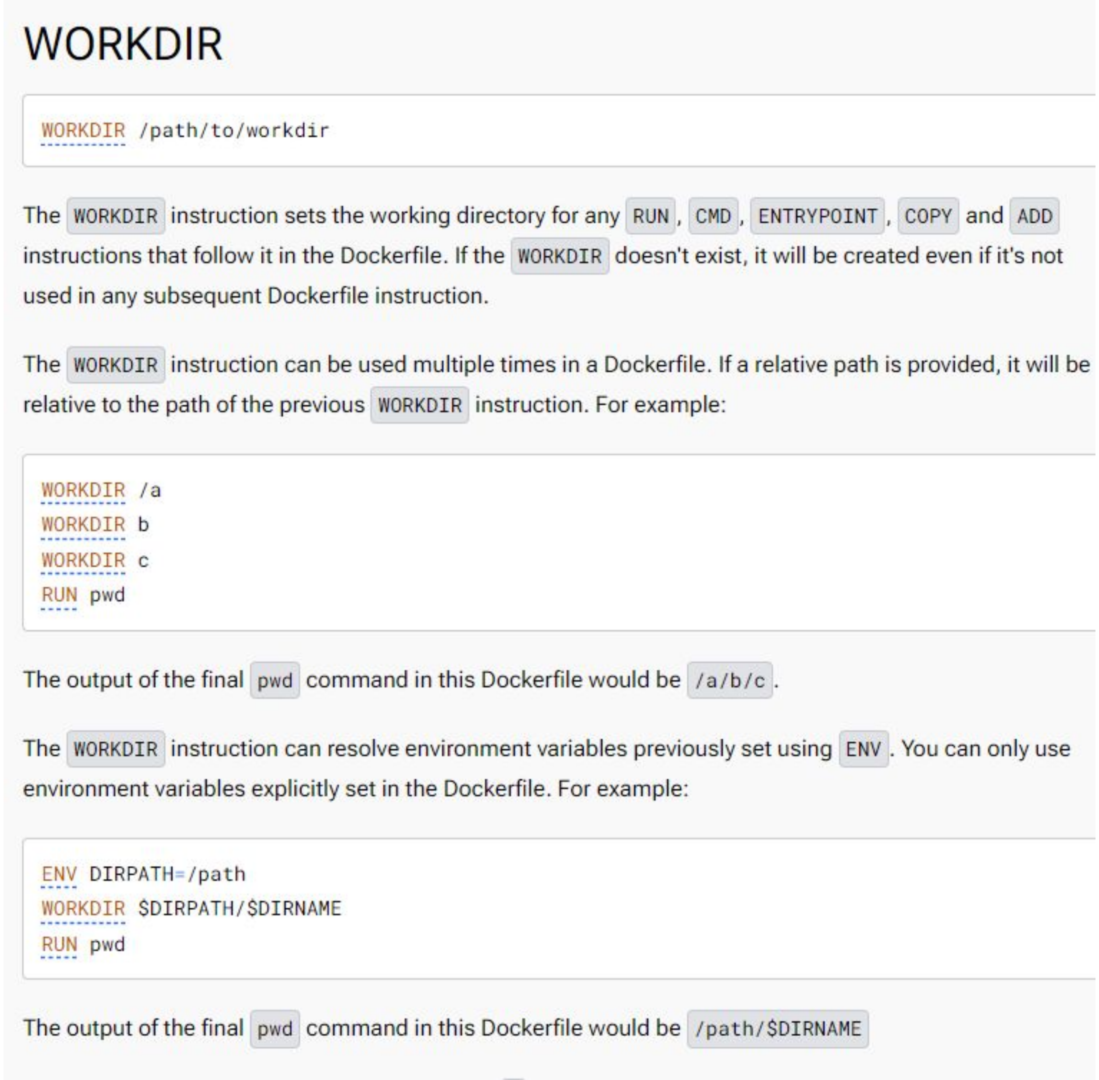
Understand how CMD and ENTRYPOINT interact

Both `CMD` and `ENTRYPOINT` instructions define what command gets executed when running a container. There are few rules that describe their co-operation.

1. Dockerfile should specify at least one of `CMD` or `ENTRYPOINT` commands.
2. `ENTRYPOINT` should be defined when using the container as an executable.
3. `CMD` should be used as a way of defining default arguments for an `ENTRYPOINT` command or for executing an ad-hoc command in a container.
4. `CMD` will be overridden when running the container with alternative arguments.

The table below shows what command is executed for different `ENTRYPOINT` / `CMD` combinations:

	No ENTRYPOINT	**ENTRYPOINT exec_entry p1_entry**	**ENTRYPOINT ["exec_entry", "p1_entry"]**
No CMD	error, not allowed	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry
CMD ["exec_cmd", "p1_cmd"]	exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry exec_cmd p1_cmd
CMD exec_cmd p1_cmd	/bin/sh -c exec_cmd p1_cmd	/bin/sh -c exec_entry p1_entry	exec_entry p1_entry /bin/sh -c exec_cmd p1_cmd



WORKDIR

```
WORKDIR /path/to/workdir
```

The `WORKDIR` instruction sets the working directory for any `RUN`, `CMD`, `ENTRYPOINT`, `COPY` and `ADD` instructions that follow it in the Dockerfile. If the `WORKDIR` doesn't exist, it will be created even if it's not used in any subsequent Dockerfile instruction.

The `WORKDIR` instruction can be used multiple times in a Dockerfile. If a relative path is provided, it will be relative to the path of the previous `WORKDIR` instruction. For example:

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

The output of the final `pwd` command in this Dockerfile would be `/a/b/c`.

The `WORKDIR` instruction can resolve environment variables previously set using `ENV`. You can only use environment variables explicitly set in the Dockerfile. For example:

```
ENV DIRPATH=/path
WORKDIR $DIRPATH/$DIRNAME
RUN pwd
```

The output of the final `pwd` command in this Dockerfile would be `/path/$DIRNAME`

References

1. https://docs.docker.com/guides/docker-overview/