# Understanding Dockerfiles
# Instructions & options of Docker

For creating an image using existing containers, the general Docker workflow was:

- **start a container based on an image in a known state**
- **add things to the filesystem, such as packages, codebases, libraries, files, or anything else**
- **commit the changes as layers to make a new image**

In the walkthrough, we will deep dive into creating docker images using a docker file and we will discuss all possible options and **instructions** which can be used to create a docker image.

In this Lab, we'll look at a more robust way to build an image. Rather than just running commands and adding files with tools like *wget*, we'll put our instructions in a special file called the ***Dockerfile***. A Dockerfile is similar in concept to the recipes and manifests found in infrastructure automation (IA) tools like Chef or Puppet.

Overall, a Dockerfile is much more stripped down than the IA tools, consisting of a single file with a DSL that has a handful of instructions. The format looks like this:

```
# Comment  INSTRUCTION arguments
```

The following table summarizes the instructions; many of these options map directly to the  option in the “docker run” command:

| Command | Description |
|---|---|
| ADD | Copies a file from the host system onto the container |
| CMD | The command that runs when the container starts |
| ENTRYPOINT | |
| ENV | Sets an environment variable in the new container |
| EXPOSE | Opens a port for linked containers |
| FROM | The base image to use in the build. This is mandatory and must be the first command in the file. |
| MAINTAINER | An optional value for the maintainer of the script |
| ONBUILD | A command that is triggered when the image in the Dcokerfile is used as a base for another image |
| RUN | Executes a command and saves the result as a new layer |
| USER | Sets the default user within the container |
| VOLUME | Creates a shared volume that can be shared among containers or by the host machine |
| WORKDIR | Set the default working directory for the container |

Once you’ve created a Dockerfile and added all your instructions, you can use it to build an image using the docker build command. The format for this command is:

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

The build command results in a new image that you can start using docker run, just like any other image. **Each line in the Dockerfile will correspond to a layer in the images commit history.**

# Example of building an image from a Dockerfile

Perhaps the best way to understand a Dockerfile is to dive into an example. Let's take a look at the example we went through in our overview chapter and condense it into a Dockerfile:

```
#
# Simple example of a Dockerfile
#
FROM ubuntu:latest
MAINTAINER Vishnu "devops@vishnu.com"

RUN apt-get update
RUN apt-get install -y python python-pip wget
RUN pip install Flask

ADD hello.py /home/hello.py

WORKDIR /home
```

As you can see, it's pretty straightforward: we start from "**ubuntu:latest**," install dependencies with the **RUN** command, add our code file with the **ADD** command, and then set the **default directory** for when the container starts. Once we have a Dockerfile itself, we can build an image using docker build, like this:

```
$ docker build -t "simple_flask:dockerfile" .
```

The **"-t"** flag adds a tag to the image so that it gets a nice repository name and tag. Also not the final **"."**, which tells Docker to use the Dockerfile in the current directory. Once you start the build, you'll see it churn away for a while installing things, and when it completes, you'll have a brand new image. Running docker history will show you the effect each command has on the overall size of the file:

```
$ docker history simple_flask:dockerfile
IMAGE               CREATED             CREATED BY                                      SIZE
9ada423c0a60        3 days ago          /bin/sh -c #(nop) WORKDIR /home                 0 B
5c3625267cd9        3 days ago          /bin/sh -c #(nop) ADD file:96e699cd177f1a3f3c   163 B
9c20a6548fbe        3 days ago          /bin/sh -c pip install Flask                    4.959 MB
7195370ae6e1        3 days ago          /bin/sh -c apt-get install -y python python-p   136.1 MB
761bf82875cc        3 days ago          /bin/sh -c apt-get update                       19.94 MB
40b29df1d2c2        3 days ago          /bin/sh -c #(nop) MAINTAINER Andrew Odewahn "   0 B
c4ff7513909d        9 days ago          /bin/sh -c #(nop) CMD [/bin/bash]               0 B
cc58e55aa5a5        9 days ago          /bin/sh -c apt-get update && apt-get dist-upg   32.67 MB
0ea0d582fd90        9 days ago          /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/   1.895 kB
d92c3c92fa73        9 days ago          /bin/sh -c rm -rf /var/lib/apt/lists/*          0 B
9942dd43ff21        9 days ago          /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic   194.5 kB
1c9383292a8f        9 days ago          /bin/sh -c #(nop) ADD file:c1472c26527df28498   192.5 MB
511136ea3c5a        14 months ago                                                       0 B
```

**Finally, you can start the container itself with the following command:**

```
$ docker run -p 5000:5000 simple_flask:dockerfile python
hello.py
```

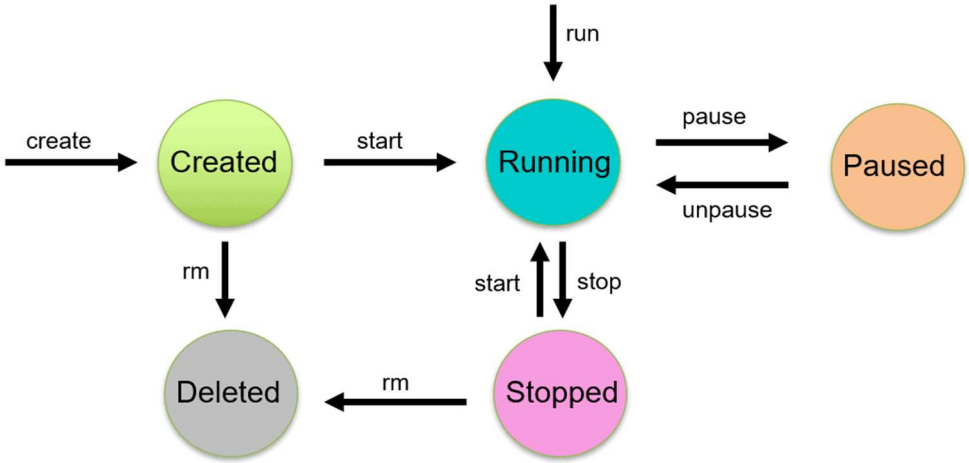
Notice that in this example we're running the Flask app directly when we start the container, rather than just running the bash shell and starting it as we've done in other examples.

# What is the Docker container lifecycle?

The complete **lifecycle of a docker container** revolves around five phases:

1. Create phase
2. Running phase
3. Paused phase/unpause phase
4. Stopped phase
5. Killed phase

Refer to the image below for the complete lifecycle of a Docker container.



## Create phase

In the **create phase**, a docker container is created from a docker image.

```
docker create --name <name-of-container> <docker-image-name>
```

## Running phase

In the **running phase**, the docker container starts executing the commands mentioned in the image. To run a docker container, use the `docker run` **command**.

```
docker run <container-id>
```

OR

```
docker run <container-name>
```

The `docker run` command creates a container if it is not present. In this case, the creation of the container can be skipped.

## Paused phase

In the **paused phase**, the current executing command in the docker container is paused. Use the `docker pause` **command** to pause a running container.

```
docker pause container <container-id or container-name>
```

Note: `docker pause` pauses all the processes in the container. It sends the `SIGSTOP` signal to pause the processes in the container.

## Unpause phase

In the **unpause phase**, the paused container resumes executing the commands once it is unpaused.

Use the `docker unpause` **command** to resume a paused container.

Then, Docker sends the `SIGCONT` signal to resume the process.

```
docker unpause <container-id or container-name>
```

## Stop phase

In the **stop phase**, the container's main process is shutdown gracefully. Docker sends `SIGTERM` for graceful shutdown, and if needed, `SIGKILL`, to kill the container's main process.

Use the `docker stop` **command** to stop a container.

```
docker stop <container-id or container-name>
```

Restarting a docker container would translate to `docker stop`, then `docker run`, i.e., stop and run phases.

## Kill phase

In the **kill phase**, the container's main processes are shutdown abruptly. Docker sends a `SIGKILL` signal to kill the container's main process.

```
docker kill <container-id or container-name>
```