Build and Configure Custom Docker Images with Dockerfile

**Pre-Requisities**

* Install Docker and Learn Docker Container Manipulation
* Deploy and Run Applications under Docker Containers

1. **What is Dockerfile?**

**A Dockerfile is a text file that defines a Docker image. You’ll use a Dockerfile to create your own custom Docker image, in other words to define your custom environment to be used in a Docker container.**

You’ll want to create your own Dockerfile when existing images don’t satisfy your project needs. This will actually happen most of the time, which means that learning about the Dockerfile is a pretty essential part of working with Docker.

All you need to do is to create a text file named Dockerfile (with no extension) and define your image.

This file content is divided into 2 parts, **Instruction** & **Argument.**

**List of instructions:**

|  |  |
| --- | --- |
| **From** | Sets the Base Image for subsequent instructions |
| **Add** | copies new files, directories or remote file to container. Invalidates caches. Avoid ADD and use COPY instead |
| **Copy** | copies new files or directories to container. By default, this copies as root regardless of the user/work directory |
| **Label** | apply key/value metadata to your images, containers, or daemons. [ eg. LAVEL email="santanudas4u@gmail.com" |
| **Run** | execute any commands in a new layer on top of the current image and commit the results |
| **Cmd** | For executing any command |
| **Maintainer** | Set the Author field of the generated images, i.e the developer name |
| **Expose** | informs Docker that the container listens on the specified network ports at runtime. NOTE: does not actually make ports accessible. |
| **Env** | sets environment variable |
| **Entrypoint** | configures a container that will run as an executable |
| **Volume** | creates a mount point for externally mounted volumes or other containers. |
| **User** | sets the user name for following RUN / CMD / ENTRYPOINT commands. |
| **Workdir** | sets the working directory. |
| **Arg** | defines a build-time variable. |
| **Onbuild** | adds a trigger instruction when the image is used as the base for another build. |
| **Stopsignal** | sets the system call signal that will be sent to the container to exit. |

### COPY vs ADD

Both ADD and COPY are designed to add directories and files to your Docker image in the form of ADD <src>... <dest> or COPY <src>... <dest>. Most resources, including myself, suggest to use COPY.

The reason behind this is that ADD has extra features compared to COPY that make ADD more unpredictable and a bit over-designed. ADD can pull files from url sources, which COPY cannot. ADD can also extract compressed files assuming it can recognize and handle the format. You cannot extract archives with COPY.

The ADD instruction was added to Docker first, and COPY was added later to provide a straightforward, rock solid solution for copying files and directories into your container’s file system.

If you want to pull files from the web into your image I would suggest to use RUN and curl and uncompress your files with RUN and commands you would use on the command line.

1. **Creating the same automation using Docker FIle:**

cat >Docerfile

FROM ubuntu

RUN apt-get update

RUN apt-get install –y python python-pip

RUN pip install flask

COPY app.py /opt/app.py

ENTRYPOINT FLASK\_APP=/opt/app.py flask run --host=0.0.0.0

**Note:** here FROM, RUN, COPY, ENTRYPOINT are INSTRUCTIONS & the other parts are ARGUMENTS

1. **Building Docker image:**

docker build . –t my-simple-web-app/webapplication:1.0

or

docker build . –t santanudas/my-simple-web-app

**This command is structured as follows:**

* docker build is the command to build a Docker image from a Dockerfile
* -t my-simple-web-app defines the tag (hence -t) of the image, which will be basically the name of the image. As the first part I put my own name my-simple-web-app, because I’m the maintainer of the image, then I gave it a human readable name **santanudas** and provided a version number 1.0.
* please note the . (dot) at the end of the line. You need to specify the directory where docker build should be looking for a Dockerfile. Therefore, tells docker build to look for the file in the current directory.

**Understand image layering**

 Docker build provided the build output in 4 steps, namely Step 1/4, Step 2/4, Step 3/4 and Step 4/4.

At the headline of each step you can see the corresponding line in your Dockerfile. This is because docker build executes the lines in the Dockerfile one at a time.

What is more important that with every step in the build process Docker will create an intermediary image for the specific step. This means that Docker will take the base image (alpine:3.4), then execute RUN apk update and then Docker will add the resulting files from that step as another layer on top of the base image.

----> these lines denote the image ids of intermediary images.

This means that the final Docker image consist of 4 layers and the intermediary layers are also available on your system as standalone images. This is useful because Docker will use the intermediary images as image cache, which means your future builds will be much faster for those Dockerfile steps that you do not modify.

Please issue the command docker images -a in terminal.

We used -a to list all images on your computer including intermediary images. Please note how the image ids are the same as the ones you see during the build process.

Only RUN, COPY and ADD instructions create layers to improve build performance.

The main advantage of image layering lies in image caching.

If you build your Dockerfile again now with the same command docker build -t takacsmark/alpine-smarter:1.0 ., you’ll notice that the build was almost instantaneous and the output for every step says that the build was done from cache.

**Note:** **docker images and docker images -a display the size of the image including the size of parent images.**

#### **Dangling images**

Images are hanging around and it does not have a proper tag or name right now. (You can check the image ids to see that this is the same image we built previously).

Docker calls such images dangling images.

You can use the following command to list dangling images:

docker images --filter "dangling=true"

I personally don’t like it when images are just hanging around without a purpose, so here is how to remove them:

docker rmi $(docker images -q --filter "dangling=true").

## **Dockerfile best practices**

## **Minimize the number of steps in the Dockerfile**

## Minimizing the number of steps in your image may improve build and pull performance. Therefore it’s a cool best practice to combine several steps into one line, so that they’ll create only one intermediary image. Keep in mind that only RUN, COPY and ADD instructions create layers.

## **Start your Dockerfile with the steps that are least likely to change**

The best practice is to structure your Dockerfile according to the following:

1. Install tools that are needed to build your application.
2. Install dependencies, libraries and packages.
3. Build your application.

### Clean up your Dockerfile

Always review your steps in the Dockerfile and only keep the minimum set of steps that are needed by your application. Always remove unnecessary components.

### Use a .dockerignore file

**The directory where you issue the docker build command is called the build context.** Docker will send all of the files and directories in your build directory to the Docker daemon as part of the build context. If you have stuff in your file that is not needed by your build, you’ll have an unnecessarily larger build context that results in a larger image size.

You can remedy this situation by adding a .dockerignore file that works similarly to .gitignore. You can specify the list of folders and files that should be ignored in the build context.

### Containers should be ephemeral

It is your best interest to design and build Docker images that can be destroyed and recreated/replaced automatically or with minimal configuration.

Which means that you should create Dockerfiles that define stateless images. Any state, should be kept outside of your containers.

### One container should have one concern

Design your application in a way that your web server, database, in-memory cache and other components have their own dedicated containers.

You’ll see the benefits of such a design when scaling your app horizontally. We’ll look into interoperability of containers and container networking in a future tutorial.

## **Dockerfile building workflow**

Fairly straightforward 4 step approach to build my Dockerfiles in an iterative manner. It is the following:

1. **Pick the right base image** - in this step I experiment with the base images available on-line for the technology in question. I usually check out different flavors, like an image based on Debian Jessie and another on based on Alpine.

I also check out the images made by others for a specific technology. For node.js, for example, I’m not building my own images from a Linux base, I usually use the offical node images as a base image.

When working with php, I usually start from php with the Apache web server included and add my stuff myself.

1. **Go to shell and build your environment** - as a next step I go with a try-and-fail approach. I pull my chosen images to my computer and start a container in interactive mode with a shell.

I start manually executing the steps in the container and see how things work out. Once a step seems to be OK, I add it to my Dockerfile.

If something goes wrong, I change the course, and I update the Dockerfile immediately.

1. **Add the steps to your Dockerfile and build your image** - I keep adding steps continuously as I make progress with my setup in the container.

Every now and then I stop and build my image from the Dockerfile to make sure that it produces the same results every time.

Then I use the newly built image to start a container with a shell and go on with my installation and set-up steps.

1. **Repeat steps 2 and 3** - I keep repeating steps 2 and 3 until I reach the stage that I like.

**More examples**:

MAINTAINER your\_name <user@domain.tld>

RUN apt-get -y install apache2

RUN echo “Hello Apache server on Ubuntu Docker” > /var/www/html/index.html

EXPOSE 80

CMD /usr/sbin/apache2ctl -D FOREGROUND

[](https://www.tecmint.com/wp-content/uploads/2016/02/Dockerfile-Repository.png)