Docker - File

Docker also gives you the capability to create your own Docker images, and it can be done with the help of Docker Files. A Dockerfile is a text file that Docker reads in from top to bottom. It contains a bunch of instructions which informs Docker HOW the Docker image should get built.

You can relate it to cooking. In cooking you have recipes. A recipe lets you know all of the steps you must take in order to produce whatever you’re trying to cook.

The act of cooking is building the recipe.

A Dockerfile is a recipe (or blueprint if that helps) for building Docker images, and the act of running a separate build command produces the Docker image from that recipe.

So at the end of the day, we can sum things up as:

A Dockerfile is a recipe for creating Docker images

A Docker image gets built by running a Docker command (which uses that Dockerfile)

A Docker container is a running instance of a Docker image

Dockerfile Basics

Before we construct our Dockerfile, you need to understand what makes up the file. This will be a text file, named *Dockerfile*, that includes specific keywords that dictate how to build a specific image. The specific keywords you can use in a file are:

ADD copies the files from a source on the host into the container’s own filesystem at the set destination.

CMD can be used for executing a specific command within the container.

ENTRYPOINT sets a default application to be used every time a container is created with the image.

ENV sets environment variables.

EXPOSE associates a specific port to enable networking between the container and the outside world.

FROM defines the base image used to start the build process.

MAINTAINER defines a full name and email address of the image creator.

RUN is the central executing directive for Dockerfiles.

USER sets the UID (or username) which is to run the container.

VOLUME is used to enable access from the container to a directory on the host machine.

WORKDIR sets the path where the command, defined with CMD, is to be executed.

LABEL allows you to add a label to your docker image.

Detailed explanation  
<https://kapeli.com/cheat_sheets/Dockerfile.docset/Contents/Resources/Documents/index>

Not all keywords are required for a Dockerfile to function. Case in point, our example will only make use of FROM, MAINTAINER, and RUN

1. Create the Dockerfile

Create an empty directory for this task and create an empty file in that directory with the name Dockerfile. You can do this easily by issuing the command touch Dockerfile in your empty directory.

Congratulations, you just created your first Dockerfile! Let’s open the file in your favorite text editor!

The Alpine image does not have git, vim and curl by default, as you can see in the video. So let’s create a custom image from Alpine that has git, vim and curl included. This will be your first custom Docker image.

2. Define the base image with FROM

Every Dockerfile must start with the FROM instruction. The idea behind is that you need a starting point to build your image. You can start FROM scratch, scratch is an explicitly empty image on the Docker store that is used to build base images like Alpine, Debian and so on.

I start my images mostly from other images. You can start you Docker images from any valid image that you pull from public registries. The image you start from is called the base image. In our case let’s add FROM alpine:3.4 to the Dockerfile.

Right now your Dockerfile should look like this:

FROM alpine:3.4

3. Add the lines to install packages

Please add the lines to install vim and curl like this:

**FROM** alpine:3.4

**RUN** apk update

**RUN** apk add vim

**RUN** apk add curl

This is not best practice, these are just a few lines to get started. Don’t worry, you’ll learn the best practices in this article.

4. Build your image

Please run the following in terminal: docker build -t takacsmark/alpine-smarter:1.0 .

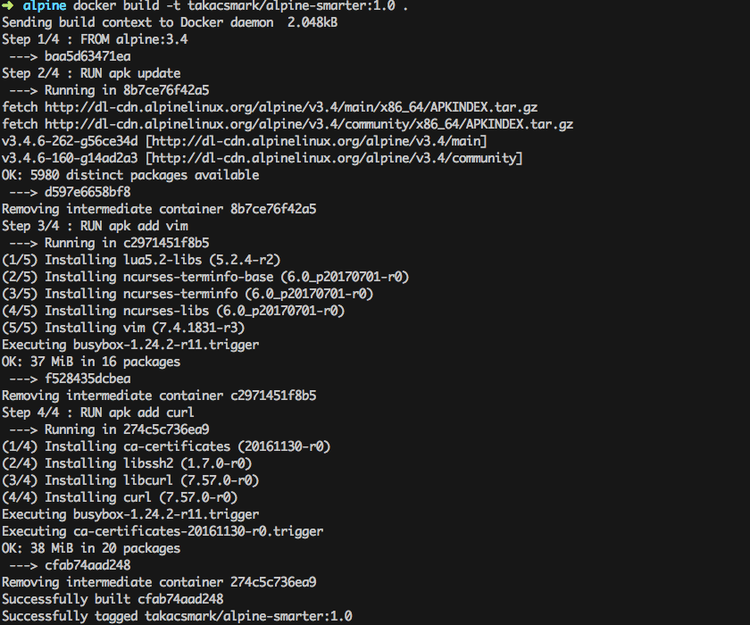
This command is structured as follows:

docker build is the command to build a Docker image from a Dockerfile

-t takacsmark/alpine-smarter:1.0 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 takacsmark, because I’m the maintainer of the image, then I gave it a human readable name alpine-smarter 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.

You should see a similar output in terminal now:



5. Enjoy the results

Docker created an image from your Dockerfile. You should see a new image in your image list issuing docker images again.

Let’s check what’s inside our new image, let’s run the following command and check out vim and curl: docker run --rm -ti takacsmark/alpine-smarter:1.0 /bin/sh

Right now you should be in the shell of your running container, so let issue the following commands: vim --v and curl --version. You should be seeing the version of vim and curl in your terminal.

We have successfully added two packages to the Alpine base image. Let’s not stop here, there is more!

Understand image layering

If you look at the above screen shot again you can notice that docker build provided the build output in 4 steps, namely Step 1/4, Step 2/4, Step 3/4 and Step 4/4.

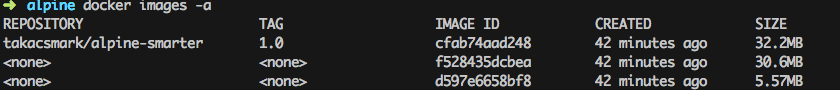
At the headline of each step you can see the corresponding line in your Dockerfile. This is because docker build executes the lines in the Dockerfile one at a time.

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.

You can follow the concept by following the line in the output that start with ----> 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.

Let’s first see all the images that were created. Please issue the command docker images -a in terminal.

You should see something like this: 

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.

This behavior makes our lives a lot easier. Since image layers are built on top of each other Docker will use images cache during the build process up to the line where the first change occurs in your Dockerfile. Every later step will be re-built.

**Please note that each layer only stores the differences compared to the underlying layer. The video may be misleading from this perspective, because I interperet the sizes in docker images -a differently. The right interpretation is that docker images and docker images -a display the size of the image including the size of parent images.**

Image cache example

Let’s play with the cache a little bit. Let’s change our Dockerfile to see the behavior. Let’s change the list line from adding curl to adding git. This is the resulting file:

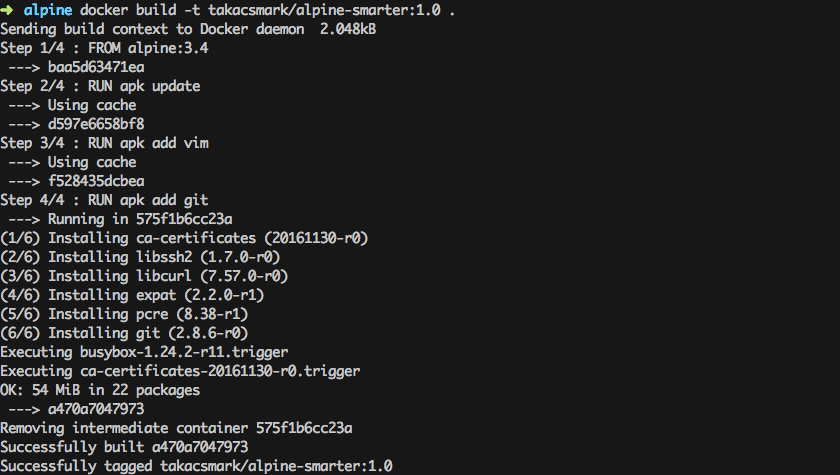
**FROM** alpine:3.4

**RUN** apk update

**RUN** apk add vim

**RUN** apk add git

Let’s issue our build command again: docker build -t takacsmark/alpine-smarter:1.0 ..

You’ll see that the first 3 steps run using cache and only the last step will be re-run, as shown in the picture. 

Please note that if you change an early step in the Dockerfile, for example you add one line after apk update like this:

**FROM** alpine:3.4

**RUN** apk update

**RUN** apk add curl

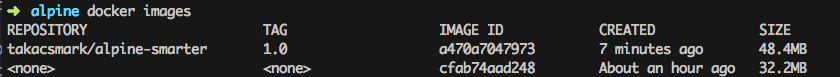
**RUN** apk add vim

**RUN** apk add git

In this case every step after the change will be re-built. Which means that the steps to install curl, vim and git will be run from scratch, no caching will be available beyond the point where the change occured.

Dangling images

If you execute docker images now in terminal, you’ll see something nasty.



Our newly built image is ready to use, but the previous image that we built with curl is still 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.

Check out the top Docker courses on Udemy

sponsored link

We can reformulate our Dockerfile like this:

FROM alpine:3.4

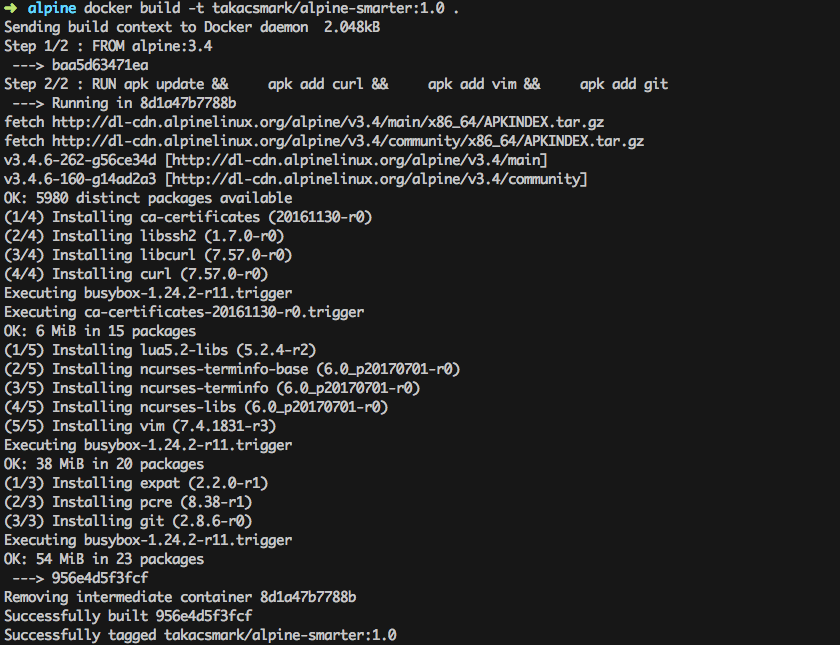
RUN apk update && \

apk add curl && \

apk add vim && \

apk add git

After building this Dockerfile the usual way you’ll find that this time it has only taken 2 steps instead of 4, which will result in 1 new image, instead of 3 images.



Keep in mind that only RUN, COPY and ADD instructions create layers.

Sort multi-line instructions

It’s a good idea to sort multiline instructions in a human readable manner. My example above is not optimal, because I’m installing packages in no order at all. I should write a file like this instead, where I order packages in alphabetical order. This is very useful when you work with a long list.

FROM alpine:3.4

RUN apk update && \

apk add curl && \

apk add git && \

apk add vim

(Yes you can remove apk add from the last 3 lines like this):

FROM alpine:3.4

RUN apk update && apk add \

curl \

git \

vim

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

This is easier said than done. Anyway, your image will stabilize after a while and changes will be less likely. The best practice is to structure your Dockerfile according to the following:

Install tools that are needed to build your application.

Install dependencies, libraries and packages.

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 directory 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.

If you want to have a look at the size of your build context, just check out the first line of your docker build output. My alpine build output for example says: Sending build context to Docker daemon 2.048kB.

Containers should be ephemeral

This would belong to generic Docker guidelines, but it’s never enough to stress this point. 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

Think of containers as entities that take responsibility for one aspect of your project. So 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.

It’s a good idea to check out [the official Dockerfile best practices page](https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/) for more info.

Dockerfile key instructions best practices

The official Docker documentation is usually very easy to follow and easy to understand. What I learnt from your comments is, that the stuff that most people need is some insight into others’ experience to see the big picture.

The reason why I’ll give you some insight into the key instructions is to share my experience and give you hints how to use them. I’ll not give you the specs, for exact specs please check the [Dockerfile reference page](https://docs.docker.com/engine/reference/builder/).

We’ll cover the following basic instructions to get you started:

FROM - every Dockerfile starts with FROM, with the introduction of multi-stage builds as of version 17.05, you can have more than one FROM instruction in one Dockerfile.

COPY vs ADD - these two are often confused, so I’ll explain the difference.

ENV - well, setting environment variables is pretty important.

RUN - let’s run commands.

VOLUME - another source of confusion, what’s the difference between Dockerfile VOLUME and container volumes?

USER - when root is too mainstream.

WORKDIR - set the working directory.

EXPOSE - get your ports right.

ONBUILD - give more flexibility to your team and clients.

FROM

For beginners it’s enough to understand that every Dockerfile must start with the FROM instruction in the form of FROM <image>[:tag]. This will set the base image for your Dockerfile, which means that subsequent instructions will be applied to this base image.

The tag value is optional, if you don’t specify the tag Docker will use the tag latest and will try and use or pull the latest version of the base image during build.

On the little bit more advanced side, let’s note the following:

There is one instruction that you can put before FROM into your Dockerfile. This instruction is ARG. ARG is used to specify arguments for the docker build command with the --build-arg <varname>=<value> flag.

You can have more than one FROM instructions in your Dockerfile. You will want to use this feature, for example, when you use one base image to build your app and another base image to run it.

It’s called a multi-stage build and you can read about it [here](https://docs.docker.com/engine/userguide/eng-image/multistage-build/).

This is why every section that starts with FROM in your Dockerfile is called a build stage (even in the simple case of having only one FROM instruction). You can specify the name of the build stage in the form FROM <image>[:tag] [AS <name>].

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.

ENV

ENV is used to define environment variables. The interesting thing about ENV is that it does two things:

You can use it to define environment variables that will be available in your container. So when you build an image and start up a container with that image you’ll find that the environment variable is available and is set to the value you specified in the Dockerfile.

You can use the variables that you specify by ENV in the Dockerfile itself. So in subsequent instructions the environment variable will be available.

RUN

RUN will execute commands, so it’s one of the most-used instructions. I would like to highlight two points:

You’ll use a lot of apt-get type of commands to add new packages to your image. It’s always advisable to put apt-get update and apt-get install commands on the same line. This is important because of layer caching. Having these on two separate lines would mean that if you add a new package to your install list, the layer with apt-get update will not be invalidated in the layer cache and you might end up in a mess. [Read more here](https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#run).

RUN has two forms; RUN <command> (called shell form) and RUN ["executable", "param1", "param2"] called exec form. Please note that RUN <command> will invoke a shell automatically (/bin/sh -c by default), while the exec form will not invoke a command shell. [If you want to tackle a problem around this read here](https://docs.docker.com/engine/reference/builder/#run).

VOLUME

This is where I found Docker documentation not so easy to follow. So let me put it in plain English.

You can use the VOLUME instruction in a Dockerfile to tell Docker that the stuff you store in that specific directory should be stored on the host file system not in the container file system. This implies that stuff stored in the volume will persist and be available also after you destroy the container.

In other words it is best practice to crate a volume for your data files, database files, or any file or directory that your users will change when they use your application.

The data stored in the volume will remain on the host machine even if you stop the container and remove the container with docker rm. (The volume will be removed on exit if you start the container with docker run --rm, though.)

You can also share these volumes between containers with docker run --volumes-from.

You can inspect your volumes with the docker volume ls and docker volume inspect commands.

You can also have a look inside your volumes by navigating to Docker volumes in your file system. On Linux you can go to /var/lib/docker/volumes pick the id of the volume and list it as a directory. You can find out the id of the container and thus the volume by running docker inspect on your container.

On Mac, you’ll not be able to access /var/lib/docker/volumes so easily. If you run screen ~/Library/Containers/com.docker.docker/Data/com.docker.driver.amd64-linux/tty on your Mac, you get a terminal where you can navigate as if you were using a Linux machine.

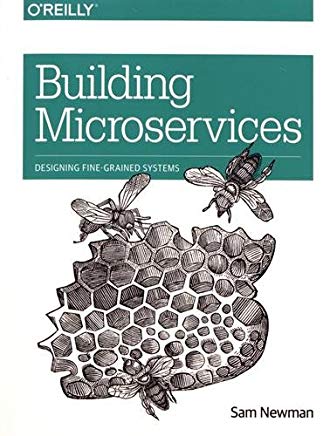
The difference between the VOLUME instruction in Dockerfile and starting your container with docker run -v ... is this: VOLUME in Dockerfile will create a new empty directory for your files under the standard Docker structure, i.e. /var/lib/docker/volumes. docker run -v ... can do more, you can mount existing directories from your host file system into your container and you can also specify the path of the directory on the host.

Now you may think that docker run -v ... is the better option, but keep in mind that mounting an existing directory assumes that the directory exists on the host, which may give you portability issues.

This topic is a good candidate for another detailed post and video, so let’s move on for now.

USER

Don’t run your stuff as root, be humble, use the USER instruction to specify the user. This user will be used to run any subsequent RUN, CMD AND ENDPOINT instructions in your Dockerfile.



Great reading - Building Microservices - Designing Fine Grained Systems

sponsored link

WORKDIR

A very convenient way to define the working directory, it will be used with subsequent RUN, CMD, ENTRYPOINT, COPY and ADD instructions. You can specify WORKDIR multiple times in a Dockerfile.

If the directory does not exists, Docker will create it for you.

EXPOSE

An important instruction to inform your users about the ports your application is listening on. EXPOSE will not publish the port, you need to use docker run -p... to do that when you start the container.

CMD and ENTRYPOINT

CMD is the instruction to specify what component is to be run by your image with arguments in the following form: CMD [“executable”, “param1”, “param2”…].

You can override CMD when you’re starting up your container by specifying your command after the image name like this: $ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...].

You can only specify one CMD in a Dockerfile (OK, physically you can specify more than one, but only the last one will be used).

It is good practice to specify a CMD even if you are developing a generic container, in this case an interactive shell is a good CMD entry. So you do CMD ["python"] or CMD [“php”, “-a”] to give your users something to work with.

So what’s the deal with ENTRYPOINT? When you specify an entry point, your image will work a bit differently. You use ENTRYPOINT as the main executable of your image. In this case whatever you specify in CMD will be added to ENTRYPOINT as parameters.

**ENTRYPOINT** ["git"]

**CMD** ["--help"]

This way you can build Docker images that mimic the behavior of the main executable you specify in ENTRYPOINT.

ONBUILD

This is so nice. You can specify instructions with ONBUILD that will be executed when your image is used as the base image of another Dockerfile. :)

This is useful when you want to create a generic base image to be used in different variations by many Dockerfiles, or in many projects or by many parties.

So you do not need to add the specific stuff immediately, like you don’t need to copy the source code or config files in the base image. How could you even do that, when these things will be available only later?

So what you do instead is to add ONBUILD instructions. So you can do something like this:

**ONBUILD** **COPY** . /usr/src/app

**ONBUILD** **RUN** /usr/src/app/mybuild.sh

ONBUILD instructions will be executed right after the FROM instruction in the downstram Dockerfile.

The following steps explain how you should go about creating a Docker File.

Step 1 − Create a file called Docker File and edit it using vim. Please note that the name of the file has to be "Dockerfile" with "D" as capital.



Step 2 − Build your Docker File using the following instructions.

#This is a sample Image

FROM ubuntu

MAINTAINER demousr@gmail.com

RUN apt-get update

RUN apt-get install –y nginx

CMD [“echo”,”Image created”]

The following points need to be noted about the above file −

The first line "#This is a sample Image" is a comment. You can add comments to the Docker File with the help of the # command

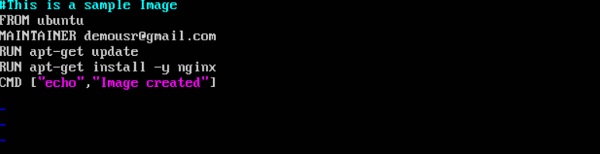
The next line has to start with the FROM keyword. It tells docker, from which base image you want to base your image from. In our example, we are creating an image from the ubuntu image.

The next command is the person who is going to maintain this image. Here you specify the MAINTAINER keyword and just mention the email ID.

The RUN command is used to run instructions against the image. In our case, we first update our Ubuntu system and then install the nginx server on our ubuntu image.

The last command is used to display a message to the user.

Step 3 − Save the file. In the next chapter, we will discuss how to build the image.



Build

Let's build a Docker image that's based on a simple node application. Execute the following command to create and switch into a folder named test.

mkdir test && cd test

Create a Dockerfile:

cat > Dockerfile <<EOF

# Use an official Node runtime as the parent image

FROM node:6

# Set the working directory in the container to /app

WORKDIR /app

# Copy the current directory contents into the container at /app

ADD . /app

# Make the container's port 80 available to the outside world

EXPOSE 80

# Run app.js using node when the container launches

CMD ["node", "app.js"]

EOF

This file instructs the Docker daemon on how to build your image.

The initial line specifies the base parent image, which in this case is the official Docker image for node version 6.

In the second, we set the working (current) directory of the container.

In the third, we add the current directory's contents (indicated by the "." ) into the container.

Then we expose the container's port so it can accept connections on that port and finally run the node command to start the application.

Now you'll write the node application, and after that you'll build the image.

Run the following to create the node application:

cat > app.js <<EOF

const http = require('http');

const hostname = '0.0.0.0';

const port = 80;

const server = http.createServer((req, res) => {

res.statusCode = 200;

res.setHeader('Content-Type', 'text/plain');

res.end('Hello World\n');

});

server.listen(port, hostname, () => {

console.log('Server running at http://%s:%s/', hostname, port);

});

process.on('SIGINT', function() {

console.log('Caught interrupt signal and will exit');

process.exit();

});

EOF

This is a simple HTTP server that listens on port 80 and returns "Hello World."

Now let's build the image.

Note again the ".", which means current directory so you need to run this command from within the directory that has the Dockerfile:

docker build -t node-app:0.1 .

It might take a couple of minutes for this command to finish executing. When it does, your output should resemble the following:

Sending build context to Docker daemon 3.072 kB

Step 1 : FROM node:6

6: Pulling from library/node

...

...

...

Step 5 : CMD node app.js

---> Running in b677acd1edd9

---> f166cd2a9f10

Removing intermediate container b677acd1edd9

Successfully built f166cd2a9f10

The -t is to name and tag an image with the name:tag syntax. The name of the image is node-app and the tag is 0.1. The tag is highly recommended when building Docker images. If you don't specify a tag, the tag will default to latestand it becomes more difficult to distinguish newer images from older ones. Also notice how each line in the Dockerfile above results in intermediate container layers as the image is built.

Now, run the following command to look at the images you built:

docker images

Your output should resemble the following:

REPOSITORY TAG IMAGE ID CREATED SIZE

node-app 0.1 f166cd2a9f10 25 seconds ago 656.2 MB

node 6 5a767079e3df 15 hours ago 656.2 MB

hello-world latest 1815c82652c0 6 days ago 1.84 kB

Notice node is the base image and node-app is the image you built. You can't remove node without removing node-app first. The size of the image is relatively small compared to VMs. Other versions of the node image such as node:slimand node:alpine can give you even smaller images for easier portability. The topic of slimming down container sizes is further explored in Advanced Topics. You can view all versions in the official repository [here](https://hub.docker.com/_/node).

Run

In this module, use this code to run containers based on the image you built:

docker run -p 4000:80 --name my-app node-app:0.1

(Command Output)

Server running at http://0.0.0.0:80/

The --name flag allows you to name the container if you like. The -p instructs Docker to map the host's port 4000 to the container's port 80. Now you can reach the server at http://localhost:4000. Without port mapping, you would not be able to reach the container at localhost.

Open another terminal (in Cloud Shell, click the + icon), and test the server:

curl http://localhost:4000

(Command Output)

Hello World

The container will run as long as the initial terminal is running. If you want the container to run in the background (not tied to the terminal's session), you need to specify the -d flag.

Close the initial terminal and then run the following command to stop and remove the container:

docker stop my-app && docker rm my-app

Now run the following command to start the container in the background:

docker run -p 4000:80 --name my-app -d node-app:0.1

docker ps

(Command Output)

CONTAINER ID IMAGE COMMAND CREATED ... NAMES

xxxxxxxxxxxx node-app:0.1 "node app.js" 16 seconds ago ... my-app

Notice the container is running in the output of docker ps. You can look at the logs by executing docker logs [container\_id].

Tip: You don't have to write the entire container ID, as long as the initial characters uniquely identify the container. For example, you can execute docker logs 17b if the container ID is 17bcaca6f....

docker logs [container\_id]

(Command Output)

Server running at http://0.0.0.0:80/

Let's modify the application. In your Cloud Shell, open the test directory you created earlier in the lab:

cd test

Edit app.js with a text editor of your choice (for example nano or vim) and replace "Hello World" with another string:

....

const server = http.createServer((req, res) => {

res.statusCode = 200;

res.setHeader('Content-Type', 'text/plain');

res.end('Welcome to Cloud\n');

});

....

Build this new image and tag it with 0.2:

docker build -t node-app:0.2 .

(Command Output)

Step 1/5 : FROM node:6

---> 67ed1f028e71

Step 2/5 : WORKDIR /app

---> Using cache

---> a39c2d73c807

Step 3/5 : ADD . /app

---> a7087887091f

Removing intermediate container 99bc0526ebb0

Step 4/5 : EXPOSE 80

---> Running in 7882a1e84596

---> 80f5220880d9

Removing intermediate container 7882a1e84596

Step 5/5 : CMD node app.js

---> Running in f2646b475210

---> 5c3edbac6421

Removing intermediate container f2646b475210

Successfully built 5c3edbac6421

Successfully tagged node-app:0.2

Notice in Step 2 we are using an existing cache layer. From Step 3 and on, the layers are modified because we made a change in app.js.

Run another container with the new image version. Notice how we map the host's port 8080 instead of 80. We can't use host port 4000 because it's already in use.

docker run -p 8080:80 --name my-app-2 -d node-app:0.2

docker ps

(Command Output)

CONTAINER ID IMAGE COMMAND CREATED

xxxxxxxxxxxx node-app:0.2 "node app.js" 53 seconds ago ...

xxxxxxxxxxxx node-app:0.1 "node app.js" About an hour ago ...

Test the containers:

curl http://localhost:8080

(Command Output)

Welcome to Cloud

And now test the first container you made:

curl http://localhost:4000

(Command Output)

Hello World

Debug

Now that we're familiar with building and running containers, let's go over some debugging practices.

You can look at the logs of a container using docker logs [container\_id]. If you want to follow the log's output as the container is running, use the -f option.

docker logs -f [container\_id]

(Command Output)

Server running at http://0.0.0.0:80/

Sometimes you will want to start an interactive Bash session inside the running container. You can use docker exec to do this. Open another terminal (in Cloud Shell, click the + icon) and enter the following command:

docker exec -it [container\_id] bash

The -it flags let you interact with a container by allocating a pseudo-tty and keeping stdin open. Notice bash ran in the WORKDIR directory (/app) specified in the Dockerfile. From here, you have an interactive shell session inside the container to debug.

(Command Output)

root@xxxxxxxxxxxx:/app*#*

Look at the directory

ls

(Command Output)

Dockerfile app.js

root@xxxxxxxxxxxx:/app*#*

Exit the Bash session. In the new terminal, enter:

exit

You can examine a container's metadata in Docker by using Docker inspect:

docker inspect [container\_id]

(Command Output)

[

{

"Id": "xxxxxxxxxxxx....",

"Created": "2017-08-07T22:57:49.261726726Z",

"Path": "node",

"Args": [

"app.js"

],

...

Use --format to inspect specific fields from the returned JSON. For example:

docker inspect --format='{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' [container\_id]

(Example Output)

192.168.9.3

Be sure to check out the following resources for more information on debugging: