Docker Explained: Using Dockerfiles to Automate Building of Images

Posted December 13, 2013 419.3k DOCKER MONGODB NGINX

Introduction

Docker containers are created by using [base] images. An image can be basic, with nothing but the operating-system fundamentals, or it can consist of a sophisticated pre-built application stack ready for launch.

When building your images with docker, each action taken (i.e. a command executed such as apt-get install) forms a new layer on top of the previous one. These base images then can be used to create new containers.

In this DigitalOcean article, we will see about automating this process as much as possible, as well as demonstrate the best practices and methods to make most of docker and containers via *Dockerfiles*: scripts to build containers, step-by-step, layer-by-layer, automatically from a source (base) image.

Glossary

1. Docker in Brief

2. Dockerfiles

3. Dockerfile Syntax

1. What is Syntax?
2. Dockerfile Syntax Example

4. Dockerfile Commands

1. ADD
2. CMD
3. ENTRYPOINT
4. ENV
5. EXPOSE
6. FROM
7. MAINTAINER
8. RUN
9. USER
10. VOLUME
11.

5.

6.

1. Creating the Empty Dockerfile
2. Defining Our File and Its Purpose

3. Setting The Base Image to Use
4. Defining The Maintainer (Author)
5. Updating The Application Repository List
6. Setting Arguments and Commands for Downloading MongoDB
7. Setting The Default Port For MongoDB
8. Saving The Dockerfile
9. Building Our First Image
10. Running A MongoDB Instance

Docker in Brief

The **docker project** offers higher-level tools which work together, built on top of some Linux kernel features. The goal is to help developers and system administrators port applications. - with all of their dependencies conjointly - and get them running across systems and machines *headache free*.

Docker achieves this by creating safe, LXC (i.e. Linux Containers) based environments for applications called “docker containers”. These containers are created using *docker images*, which can be built either by executing commands manually or automatically through **Dockerfiles**.

Note: To learn more about docker and its parts (e.g. docker daemon, CLI, images etc.), check out our introductory article to the project: Docker Explained: Getting Started.

Dockerfiles

Each Dockerfile is a script, composed of various commands (instructions) and arguments listed successively to automatically perform actions on a base image in order to create (or form) a new one. They are used for organizing things and greatly help with deployments by simplifying the process start-to-finish.

Dockerfiles begin with defining an image FROM which the *build process* starts. Followed by various other methods, commands and arguments (or conditions), in return, provide a new image which is to be used for creating docker containers.

They can be used by providing a Dockerfile's content - in various ways - to the **docker daemon** to build an image (as explained in the "How To Use" section).

Dockerfile Syntax

Before we begin talking about Dockerfile, let's quickly go over its syntax and what that actually means.

What is Syntax?

Very simply, syntax in programming means a structure to order commands, arguments, and everything else that is required to program an application to perform a procedure (i.e. a function / collection of instructions).

These structures are based on rules, clearly and explicitly defined, and they are to be followed by the programmer to interface with whichever computer application (e.g. interpreters, daemons etc.) uses or expects them. If a script (i.e. a file containing series of tasks to be performed) is not correctly structured (i.e. wrong syntax), the computer program will not be able to parse it. Parsing roughly can be understood as going over an input with the end goal of understanding what is meant.

Do
es

D

Dockerfile syntax consists of two kind of main line blocks: comments and commands + arguments.

```
# Line blocks used for commenting
command argument argument ..
```

> A Simple Example:

```
# Print "Hello docker!"
RUN echo "Hello docker!"
```

Dockerfile Commands (Instructions)

Currently there are about a dozen different set of commands which Dockerfiles can contain to have docker build an image. In this section, we will go over all of them, individually, before working on a Dockerfile example.

Note: As explained in the previous section (Dockerfile Syntax), all these commands are to be listed (i.e. written) successively, inside a single plain text file (i.e. Dockerfile), in the order you would like them performed (i.e. executed) by the docker daemon to build an image. However, some of these commands (e.g. MAINTAINER) can be placed anywhere you seem fit (but always after FROM command), as they do not constitute of any execution but rather *value of a definition* (i.e. just some additional information).

ADD

The ADD command gets two arguments: a source and a destination. It basically copies the files from the source on the host into the container's own filesystem at the set destination. If, however, the source is a URL (e.g. http://github.com/user/file/), then the contents of the URL are downloaded and placed at the destination.

Example:

```
# Usage: ADD [source directory or URL] [destination directory]
ADD /my_app_folder /my_app_folder
```

CMD

The command CMD, similarly to RUN, can be used for executing a specific command. However, unlike RUN it is not executed during build, but when a container is instantiated using the image being built. Therefore, it should be considered as an initial, default command that gets executed (i.e. run) with the creation of containers based on the image.

To clarify: an example for CMD would be running an application upon creation of a container which is already installed using RUN (e.g. RUN apt-get install ...) inside the image. This default application execution command that is set with CMD becomes the default and replaces any command which is passed during the creation.

Example:

```
# Usage 1: CMD application "argument", "argument", ..
CMD "echo" "Hello docker!"
```

ENTRYPOINT

EN
ha
an

If y
ENTRYPOINT.

Example:

```
# Usage: ENTRYPOINT application "argument", "argument", ..
# Remember: arguments are optional. They can be provided by CMD
#           or during the creation of a container.
ENTRYPOINT echo

# Usage example with CMD:
# Arguments set with CMD can be overridden during *run*
CMD "Hello docker!"
ENTRYPOINT echo
```

ENV

The ENV command is used to set the environment variables (one or more). These variables consist of “key = value” pairs which can be accessed within the container by scripts and applications alike. This functionality of docker offers an enormous amount of flexibility for running programs.

Example:

```
# Usage: ENV key value
ENV SERVER_WORKS 4
```

EXPOSE

The EXPOSE command is used to associate a specified port to enable networking between the running process inside the container and the outside world (i.e. the host).

Example:

```
# Usage: EXPOSE [port]
EXPOSE 8080
```

> To learn about ports, check out this document on port redirection.

FROM

FROM directive is probably the most crucial amongst all others for Dockerfiles. It defines the base image to use to start the build process. It can be any image, including the ones you have created previously. If a FROM image is not found on the host, docker will try to find it (and download) from the **docker image index**. It needs to be the first command declared inside a Dockerfile.

Example:

```
# Usage: FROM [image name]
FROM ubuntu
```

MAINTAINER

Or
ex

Ex

```
# Usage: MAINTAINER [name]
MAINTAINER authors_name
```

RUN

The RUN command is the central executing directive for Dockerfiles. It takes a command as its argument and runs it to form the image. Unlike CMD, it actually **is** used to build the image (forming another layer on top of the previous one which is committed).

Example:

```
# Usage: RUN [command]
RUN aptitude install -y riak
```

USER

The USER directive is used to set the UID (or username) which is to run the container based on the image being built.

Example:

```
# Usage: USER [UID]
USER 751
```

VOLUME

The VOLUME command is used to enable access from your container to a directory on the host machine (i.e. mounting it).

Example:

```
# Usage: VOLUME ["/dir_1", "/dir_2" ..]
VOLUME ["/my_files"]
```

WORKDIR

The WORKDIR directive is used to set where the command defined with CMD is to be executed.

Example:

```
# Usage: WORKDIR /path
WORKDIR ~/
```

How to Use Dockerfiles

Using the Dockerfiles is as simple as having the docker daemon run one. The output after executing the script will be the ID of the new docker image.

Usage:

```
# Build an image using the Dockerfile at current location
# Example: sudo docker build -t [name] .
su
```


D

In

which can be used to create a docker image to run MongoDB containers.

Note: After starting to edit the Dockerfile, all the content and arguments from the sections below are to be written (appended) inside of it successively, following our example and explanations from the **Docker Syntax** section. You can see what the end result will look like at the latest section of this walkthrough.

Creating the Empty Dockerfile

Using the nano text editor, let's start editing our Dockerfile.

```
sudo nano Dockerfile
```

Defining Our File and Its Purpose

Albeit optional, it is always a good practice to let yourself and everybody figure out (when necessary) what this file is and what it is intended to do. For this, we will begin our Dockerfile with fancy comments (i#) to describe it - *and have it like cool kids*.

```
############################################################
# Dockerfile to build MongoDB container images
# Based on Ubuntu
############################################################
```

Setting The Base Image to Use

```
# Set the base image to Ubuntu
FROM ubuntu
```

Defining The Maintainer (Author)

```
# File Author / Maintainer
MAINTAINER Example McAuthor
```

Updating The Application Repository List

Note: This step is not necessary, given that we are not using the repository right afterwards. However, it can be considered good practice.

```
# Update the repository sources list
RUN apt-get update
```

Setting Arguments and Commands for Downloading MongoDB

```
################## BEGIN INSTALLATION ######################
# Install MongoDB Following the Instructions at MongoDB Docs
# Ref: http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/

# Add the package verification key
RU
#
RU                                                                .lis
#
RU

# Install MongoDB package (.deb)
RUN apt-get install -y mongodb-10gen
```

```
# Create the default data directory
RUN mkdir -p /data/db

##################### INSTALLATION END #####################
```

Setting The Default Port For MongoDB

```
# Expose the default port
EXPOSE 27017

# Default port to execute the entrypoint (MongoDB)
CMD ["--port 27017"]

# Set default container command
ENTRYPOINT usr/bin/mongod
```

Saving The Dockerfile

After you have appended everything to the file, it is time to save and exit. Press CTRL+X and then Y to confirm and save the Dockerfile.

> This is what the final file should look like:

```
###############################################################
# Dockerfile to build MongoDB container images
# Based on Ubuntu
###############################################################

# Set the base image to Ubuntu
FROM ubuntu

# File Author / Maintainer
MAINTAINER Example McAuthor

# Update the repository sources list
RUN apt-get update

################## BEGIN INSTALLATION ######################
# Install MongoDB Following the Instructions at MongoDB Docs
# Ref: http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/

# Add the package verification key
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10

# Add MongoDB to the repository sources list
RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | tee /etc/apt/sources.list.d/mongodb.list

# Update the repository sources list once more
RUN apt-get update

# Install MongoDB package (.deb)
RU
#
RU
##

# Expose the default port
EXPOSE 27017
```

```
# Default port to execute the entrypoint (MongoDB)
CMD ["--port 27017"]

# Set default container command
ENTRYPOINT usr/bin/mongod
```

Building Our First Image

Using the explanations from before, we are ready to create our first MongoDB image with docker!

```
sudo docker build -t my_mongodb .
```

Note: The **-t [name]** flag here is used to tag the image. To learn more about what else you can do during build, run `sudo docker build --help`.

Running A MongoDB Instance

Using the image we have build, we can now proceed to the final step: creating a container running a MongoDB instance inside, using a name of our choice (if desired with **-name [name]**).

```
sudo docker run -name my_first_mdb_instance -i -t my_mongodb
```

Note: If a name is not set, we will need to deal with complex, alphanumeric IDs which can be obtained by listing all the containers using `sudo docker ps -l`.

Note: To detach yourself from the container, use the escape sequence CTRL+P followed by CTRL+Q.

Enjoy!

Submitted by: O.S. Tezer

The Docker Ecosystem: Scheduling and Orchestration

The Docker Ecosystem: An Overview of Containerization

SCROLL TO TOP

13 Comments

Leave a comment...

Log In to Comment

scott373738 *January 10, 2014*
0 The syntax for setting an ENV variable is:

Usage: ENV key value
Not
Usage: ENV key = value

kamaln7 MOD *January 10, 2014*
0 Nice catch, scott. I'll update the article. :]

randeep0146 *May 16, 2014*
0 Thanks for the info, I would like to add that DockerFile is a script that houses instructions needed to create an image. The image can then be created based on the instructions in the DockerFile using the Docker build command. It simplifies deployment of an image by easing the entire image and container creation process.

http://flux7.com/blogs/docker/docker-tutorial-series-part-3-automation-is-the-word-using-dockerfile/

mitchellsimoens *February 17, 2015*
0 One thing to note, each time you use the RUN command, it makes a layer in your image which may bloat your total image tree (if you run docker images --tree in the terminal to view the tree structure).

If you don't want all those layers, use a single RUN command. A great example of this is actually the Dockerfile for the ubuntu image.

tetractys *June 29, 2016*
0 This should be mentioned in the section on the RUN command, since inadvertently adding layers is not best practice or a good habit for a beginner to build. Nice info.

4

0 This guide is out of date and stale.

SCROLL TO TOP

tetractys *June 29, 2016*

0 Seems that all of these big shot sites and service providers love having nice tutorials but can't be bothered actually keeping them useful or relevant. Like the software they explain, the guides and tutorials MUST evolve and conform to changes in the respective software, else they become useless disk space on their server and make newbs pull their hair out. One major thing I've noticed in my two years as a budding developer. No one can be bothered to actually revise or tweak any of their articles to keep them in any way correct or accurate.

What else in here is out of date and wrong? I'm making notes as I go and I'm trying to make sure they're accurate for 2016/Docker ~v1.11.

---

igorg *April 12, 2016*

1

```
sudo docker run -name my_first_mdb_instance -i -t my_mongodb
```

wont' work.
Should be:

```
sudo docker run --name=my_first_mdb_instance -i -t my_mongodb
```

---

tetractys *June 29, 2016*

0 I noticed that you haven't updated the ENV bit completely. I'm making notes and trying to learn and this almost messed me up save that I read the comments. You still have "key = value" as the explanation but your example is correct. This should be corrected in both places so as not to cause further confusion.

Also, your link to documentation on port forwarding under the bit on EXPOSE is a 404. Regarding the actual content of that section, you should explain how Docker knows what port will be mapped to 8080 or explain what default ports will be mapped to ones declared by EXPOSE. Reading this as a newb, I see I'm exposing 8080 but what port is that mapped to? 80? 443? 3306? Some more clarification on how that works would be good for newbs like me since that and the ENV leave much to be presumed, which is exactly the opposite of what a beginner tutorial should do.

---

oquidave *October 5, 2016*

0 I get the following error when I try to start the image >

[initandlisten] error couldn't remove journal file during shutdown Couldn't fsync directory '/data/db/journal': errno:30 Read-only file system

---

anhducbkhn *November 14, 2016*

0 Hi,
This is my dockerfile

FROM phusion/baseimage:0.9.19
MAINTAINER anhducbkhn
RUN apt-get update
RUN apt-get install -y memcached
RUN memcached -d -u nobody -m 1024 -l 127.0.0.1 -p 11211
EXPOSE 11211
ENTRYPOINT usr/bin/memcached

I build successfully image, but cannot start.
Each time, I run command:

sudo docker run -i -t my_memcached /bin/bash

---

jkana *January 30, 2017*

0 Thanks for this great tutorial! I wanted to ask a docker question and got your tutorial, I think all my questions have been answered for now...