Docker:
The Ultimate Beginners Guide To Starting With And Mastering Docker Fast

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/]

Table of Contents

Introduction: All About Space

There is a long history of the use of containers when it comes to computers. As any good computer programmer knows, containers run off of an operating system kernel rather than using physical hardware. Using multiple isolated instances of user space, the container methodology makes use of just one host. It is precisely because of their role as mere visitors to the operating system that containers are often viewed as less reliable and less secure than a medium that is physically bolted onto your hard drive.

Despite these reservations, however, containers have been found to be incredibly useful in specific test cases. When it comes to hyper-scale deployments and multi-tenant services, for example, they are found to work without a hitch. But no matter how well they may serve a purpose in certain scenarios containers is ultimately rather complex to deal with and keep track of.

This is exactly where Docker comes in. Because by being a completely open source engine that can fully automate the placement of apps into their containers Docker stands head and shoulders above the rest. Docker was created to enable an incredibly lightweight, fast-paced environment to issue code and a well-balanced workflow that enables your code to migrate from the test environment to your PC. It’s as basic as it gets. Really all you need is a compatible Linux kernel and you're ready to go, docking your apps in a matter of mere minutes.

Chapter 1: Docker Installation

The installation process of Docker couldn't be easier. Working on a great swath of platforms, it can be used through RHEL, as well as Ubuntu and its derivatives. And within the virtual domain, you can easily run and install Docker on your Windows or OS X operating systems.

There is even a tiny virtual manifestation that can guide you along by the name of "Boot2Docker". This Docker client can be placed on any platform that has a connection to the Docker daemon.

Before installation, you just need to make sure that you can meet a few requirements. First of all, you need to have at least a 64-bit architecture in place, because 32 bit and lower are not supported. You also have to have a kernel that is Linux 3.8 or higher. You may need to carry out a few diagnostic tests to make sure that your computer is up to par. After you have done this you can then add Docker to your host.

Start off by asking for your curl command within the programming shell. Type in: $ do sh – c "echo deb https://get.docker.io/ibunto docker main>. This will let you know if you need to install your curl command or if it is already in place. Once this is established you then need to insert your Docker containers "GPG" key by typing: $ studio apt – get – y install curl. After this simply update your APT: $ curl – s https://get.docker.io/gpg.

The Docker package can now be installed by typing: $ studio apt – get install lxc – Docker. Just check up on your installation to make sure Docker is indeed up and running by punching in the following: $ sudo docker info Containers 0 Images: 0.

But if you happen to use a what is termed a "UFW"(uncomplicated Firewall), you may need to make some adjustments in order to use this setting inside Docker. This is due to the fact that Docker runs on a complex network of intertwined bridges that are used to manage the containers.

This is then forwarded by UFW to all nearby packets. So in order for this to work appropriately, you will have to enable forwarding. In order to issue this command type in: DEFAULT_FORWARD_POLICY= "Drop". The next thing that you need to do is make sure that your device mapper is properly configured.

This device-mapper can be placed on any on a Red Hat Enterprise Linux platform. The process for installation may differ somewhat between different variations. For the Red Hat variation for Linux for example, you will need to install "EPEL" in order to be able to add your RPM.

If this is the case, then type in the following: $sudo rpm – Uvh http://download.fedoraproject.org/pub/epel/6/i386/epel-release-6 -8.noarch.rpm. With this command, you should be able to install your Docker Package.

Another thing that you need to be aware of is some of the package name changes that can be used to install the “docker-io package”. With the installation of this package, you can then begin the process of your Docker daemon.

This can be done with Red Hat Enterprise Linux 6 and CentOS 6 you that you can use. In order for your Docker to start up during the booting process, you should also enable your “sudo service enable docker”. If however, you are trying to run Docker on an OS X platform, you can start up the Docker platform by enabling your Boot2Docker tool.

It is a small virtual application and has one supporting command line and can be used in your OS X installation. And since your Boot2Docker is a small virtual machine inside your local IP address. In order to find your address, you can enter the boot2docker IP command. When you do this an instance such as $ curl localhost: You should now be able to install Docker in a variety of forms and on a variety of platforms.

Chapter 2: Getting Started on the Dock!

The first thing you need to do after you have installed Docker is to make sure that it is working correctly. After you have done this you then need to try and gauge how well your workflow is running. Pay close attention to how your system is managing the creation of containers. You can then take a container on through its usual lifecycle in its managed condition and you can remove it. In doing this, first, go to your binary Docker and make sure that it is operational.

Once this is established you can then build your first container. To design your own container simply use the old Docker running command to create it. The Docker can then be accessed by issuing a run command. This command is normally issued under two flats, the “t” and the “I”. It is the “t” flag that informs Docker what kind of material to create. It is the “I” command however that keeps the STDIN open on the container.

This kind of input is only half of what is needed for an interactive shell. The “t” also is the standard on which input is based. Now we can program Docker to create a totally new container. Then simultaneously in the background, you will find that Docker is already checking locally for the right ubuntu image. If this image is not found on the local Docker host it will be found on the Docker Hub run by Docker. This image can then be used to design new containers within the file system.

The IP address then works as a bridge, talking directly to the local host involved. You may be able to continue tinkering around with your container as long as you want. Upon finishing just punch in the “exit” command. The container has now ceased to run. You can now rest assured in the knowledge that your container is being properly stored without running and eating up your background data.

By just running docker you can use the docker “ps –a command”. Next, you need to give your container a name. Just a random name so that you can keep track of it, you can do this by entering “name flag”. For example, if you want to name your container Betty, you would name it “Betty_the_container”. You will then have the ability to enter in a “stopped container” by punching in: $ sudo docker start betty_the_container.

You may also reference the container by its container ID if you so choose. And if your container is reset you will have the same options for launching your Docker run command. There is also an interactive session container running. You can then reattach the whole thing simply by punching in your Docker command. After doing this Docker should be ready to go. I hope this chapter has helped you get started!

Chapter 3: Exploring Docker Image

In this chapter, we are going to learn a bit more about a fundamental aspect of docker; the docker image. When we call something a “Docker Image” we are simply employing shorthand for a network of file systems which are arranged in layers.

And at the very bottom of this system is always going to be the “bootif” (boot file system). This boot file system tends to resemble a Linux boot file system in its makeup and if you are familiar with the system it is unmistakable. But in actuality—most of the time—a docker user will not interact with the boot file system at all.

In fact, if any given container has been booted, it will then be moved right into the memory base of the system. This allows you to free up RAM that is being used by your “initrd” disk image. For most this probably resembles the usual Linux virtualization stack. In fact, Docker will work to layer your entire root file system. You have rootfs, and you have a boot file system that can be used as more than one operating system.

The good news is that these rootfs can be used on more than one of your typical operating platforms. For example, in the more common Linux boot, you will find this kind of file system put on a read-only and then calibrated so that its integrity can be checked.

But when it comes to Docker, however, the root file system will firmly stay in a read-only format. Docker will take full advantage of a union mount however and will use it to add even more read files.

But just take a union mount and you will be able to pull up several of these read files with no problem. This union mount will then infiltrate all of your file systems and allow the union mount to allow the files and subdirectories to work in the underlying systems.

These are the file systems that Docker calls “images”. This may seem like a rather longwinded explanation, but it really is a quite simple concept. These images can be neatly stacked on top of each other.

One thing to remember is that the image below is always going to be the parent image while the final image you will see is known as the base image. Now once our container is set to an image the Docker will then launch into a read-write file system. This will allow our Docker container applications to work. I hope that didn’t sound too terribly confusing. If so—let me clarify—and break it all down for you very simply.

First you have a writeable container, this is then stacked on top of your image, you can then use applications such as “Add Apache” this is then followed by another layered image, you can follow this up then with another function such as “Add Emacs” or “Ubuntu” this then will be followed by a base image, followed by your “bootfs”. After these, you will bottom out this configuration with your device mapper, namespace, csgroups and finally your Kernel which holds everything in place.

Think of it like a sandwich with a bunch of various layers, except instead of Bacon, Lettuce, and Tomato, you have Add Apache, Add emacs, and “Ubuntu” all between your slices of bread (images).

This is how it works. Just be aware that when you first employ Docker you will no doubt see an empty read-write layer. As this happens just apply the next layer and add your file that you need to have copied.

It is the read-only file that will continue to exist hidden away underneath your cloned copy. But first to get started with your Docker images you need to take a look at what images are actually available on your Docker hosting service. This is easily done by punching in your “docker images” code. You should see something like this appear:

$ sudo docker images

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
Ubuntu	latest	c4ff7513909d	6 days ago	225.4 MB

As you can see you have an image for a repository under the heading of “ubuntu”. From whence does this image arise? For this exercise, it is the Ubuntu image. Just think about how that works for a moment, this one image is able to control and collaborate with all of the files in your system in order to transition a smooth workflow.

This is a great way to organize your database. Adding to the convenience is the fact that Docker code is open source, meaning you can just dive right in without any fear of interference.

This is great for your “Docker Hub”; the place where your images are deposited. Inside this hub is a wide range of data such as your layers, images, and other metadata associated with your images.

You may notice that the top tier images are the only ones attached to ubuntu. These are managed by Docker and their vendors to provide a diverse blend of images that you can use for your basic specifications.

These areas of expertise make clear their solidarity with vendors and Docker inc. They are layered, contained and well constructed, with the correct date superimposed on the file.

In addition to this you should also be aware that the containers that you have docked will activate even if the image you have processed isn't available locally, if you need it, it can be instantly downloaded from the Hub.

And even if you don't specify exactly what tag you are looking for Docker will then happily supply you with the latest tag all of its own accord. Another good method for this is to use the "docker pull" command that will manage to pull images down preemptively. This saves a considerable amount of time when you are trying to launch a container with another image.

In addition to all these another focal point of interest for your Docker Hub. It is then possible to build a completely new image by simply building a brand new container. Now that we have learned how to manage our images lets focus on building some of our own from scratch. There are essentially two valid ways in which you can create a docker image. You can do it through the "docker commit" command" or you could do it with the "docker build" command by using a "docker file".

Using the "doccer commit" command is the most direct and the easiest to implement. It is a much more eclectic and versatile means of production. If you do not wish to create an image in this fashion you can do so by using a "docker file" to issue the docker build command. To get started you are going to need a "Docker Hub" account. To sign into this account all you have to do is use your Docker login command.

By doing this you can log into your Docker Hub and load up your credentials so that it will be there when you need it. After reading this chapter you should now know how Docker images can interact with each other and have a basic understanding of how you can modify, update and upload the metadata within the Docker Index.

You should also understand how to create your own images through your Docker files with relative comfort and ease. In the end, you should be well aware of how you can run and manipulate your Docker System in order to get the best benefits and the best means in which to use them.

Chapter 4: Making Tests with Docker

Testing! Testing! This is a test of the Emergency Docker System! Well—actually it is not quite that dramatic—but it is technical. All testing processes need to be thorough, so in this chapter, we are going to try and highlight all of the most important aspects in full detail. There are three main uses for Docker in these scenarios. Number one, a Docker test is most commonly used in order to test out a static website of some sort.

Number two, Docker platforms are built up to test out various internet apps. And finally, the third most common reason that function tests are run using Docker would be due to the fact that Docker tests are essential in order to have continuous integration of all of your systems and platforms. Web developers use Docker all the time and love its ability to manipulate the environment.

It can manipulate the immediate environment and allow you to begin installation of your Nginx container so that you can run simplistic web pages. To see this in action let's try our luck with bare bones, basic Dockerfile. Now let’s design a directory that will hold this Dockerfile inside it. It should look a little something like this: $ mkdir ngi $ cd $ touch Dockerfile.

By writing this out you have managed to install Nginx, create your own directory for a web page, add special configurations for Nginx to your local files for your image as well as exposing port 80 of the image. The two Nxix fies you painstakingly configured are then going to be needed to run the test for your webpage. As soon as you do this your nginx/gobal.conf file wi be copied right into your chosen directory through the ADD command.

This is where your port 80 will come in setting the stage of your root web server, moving it to /var/www/html/website. Nginx can then be used in a non-daemonized in order to get everything to work from within the parameters of your chosen Docker container. These are things that can be easily tested out within the confines of Docker. With this configuration file, you can take the daemon option completely off.

This allows you to keep from running in the background, bringing it right up to the forefront. And take it from someone who knows, because believe me, this will save you a lot of time, energy, and frustration in the long run.

Docker containers work on the premise that the running process inside of them will remain active. Because it is the natural default position of Nginx for it to demonize as soon as the program starts (and no this demon doesn’t need an exorcist!) the daemon will continue to run until it is launched.

Now you can take out your Docker file and be confident in you or knowledge of building bigger and better images on command. First, build y tour name and then you can design a vast litany of new images that can be used to build up everything for immediate execution.

After we have done this, let us take the time to examine the unique history that all of these efforts seem to espouse. Your Docker history begins with your final layer and then works backward to the very first copy or as it is sometimes referred to, the “parent image”.

No, you may notice that we have utilized a Docker run command in order to create an airtight container for your domain or web page. You most likely will know the most current options that will be available to you will be in regard to volume and the so-called “v option”.

There will then be a brief digression in volumes and you will find that the most important and necessary data will be conveniently placed on the shelves of your Docker.

The volume itself is a uniquely designed directory that is able to bypass a layered Union File System. Any changes to your volume are then made by directly bypassing your image and then with the inclusion of your built image. In the meantime, most of the testing can be done rather simultaneously.

It can change quite frequently and the image can be rebuilt during the process of development. Just make sure that your code is evenly distributed in between all of the various containers.

This option can work as the main directory that mounts a local host that is held in separation from the source of destination and then specify the read/write status of the whole deal. For example, the Nginx container, you can then mount it onto the web page that you are creating. Nginx is the best way to configure the directory for your server of choice.

The easiest way to do this is to use a framework such as Ruby on Rails, and PHP. The next thing that you should test out is how you can use Docker in order to test and build your very own web apps. The next application then should take these testing parameters to their fruition and logical conclusion. The Dockerfile can build up a basic image that develops the application. In the end, it should look a little something like this:

Web App Dockerfile Test
Maintainer Lucy Lee “lucy@notreal.com”
ENV REFRESHE_AT 2014-01
RUN apt-get update
RUN apt-get –y install ruby ruby-dev build-essential redis-tools
RUN gem install—no-rdoc—no-ri Sinatra json redis
RUN mkdir –p /opt/webapp
EXPOSE 4567
CMD

With this formula, you should be able to create your own image completely based on the Ubuntu. Now you can work to install Ruby Gems and Ruby into the mainframe of your gem binary.

You can also create a directory that will be able to hold a new web application that can be used as the default port in your container. This, my friends, is what Docker is all about, and the sooner you learn it the sooner that you will have a high-functioning operation! This is precisely why testing is so important.

Chapter 5: How to Use Docker API

Now that we have tested out our applications with Docker, we are now going to discover how we can build a workflow with Docker API. This way you can bind the Docker daemon to any networked port that you may have. When it comes to the Docker system there are three API's that you should be aware of. There is the remote Docker API, and there is the Docker Hub, and then finally the API registry.

These are all mainly "restful" API's. First off, let's check out the capabilities of our Docker remote API. These Docker daemons will bind to your socket and then turn on the host unit that is running. This then allows your daemon to run off of its root privilege so it will be able to utilize the right resources and manage itself accordingly.

Your Docker will then use its ownership for the group. In the end, you should have it configured to where any user, regardless of their root privileges, will be able to have access and run Docker.

This is, after all, an equal opportunity software. And the first thing that you will find is that whenever you make a query, there will simultaneously be another API running across the networks of your interface. A feat which can be accomplished by tweaking the "-H flag" sent right to your daemon Docker. This is easily achieved through a simple matter of adjusting the configuration files of your daemon and editing the final proof of the material.

Start out by taking control of and biding your daemon directly to your interface by using the port 2375. Just as shown here: ExecStart=/usr/bin/docker –d –H tcp://0 . 0 . 0 . 0 . 0 . 0 . 0: 2375. With your daemon bound, you are not free to test the working mechanisms of your Docker's client binary.

This involves the use of the –H flag in conjunction with your Docker's host. But even better than this, is being able to hook up with our daemon through a remotely. You are going to use your H flag to determine our host within the Docker host platform. It should look similar to this table:

Hook Up with your Daemon Remotely
$ $ sudo docker –H docker . example . com 0 0 0 0 0 0 0: 2375 info
Containers: 0 0 0 0
Images: 0 0 0 0
Driver: device mapper
Pool Name: docker – 252: 0 - 1 3 3 3 4 4 - pool
Data file: / var / lib / docker / device mapper / device mapper / data
Metadata file: / var / lib / docker / device mapper / device mapper / meta data

You should now be able to hook up with your daemon directly to your remote host. You should also be happy to hear that this API will be able to utilize all container operations that are a part of the command line. These can all be easily listed by using your containers termination point just like you would with a mainline ps docker command code.

You need to then move forward with the configuration of your container design by increasing the pair value of your hash. Just remember that a container must always be activated by its termination point. What you have seen so far in this chapter in regard to daemon's and API's has been universal all access ports. If however, you wish to have some security in your system you will need to add an authentication mechanism.

These authentication mechanisms use TLS/SS certification to ensure that all connections to your API are merited. Just be sure to configure your daemon to the key and certificates that you use for authentication. When using TLS in authentication it is using "tlsvverfigy flag" as its indicator. By the certificates used you should also be able to tell what the location and other options are.

With your server TLS ready you can then work to design your own personal credentials that you can give as a key, and distribute it to everyone that is working in your network. With this system, you can now secure your Docker API client library and you will also be able to rewrite the TProv application that is used directly; this is how to use Docker API.

Chapter 6: Most Common Problems

Now that Docker is installed and you've already gotten started on the basics. Let's delve into some of the most complex problems that are most often faced. One of the first things that many encounter is what happens when the run "docker-compose" this will start up the old release and then run the new on in its place.

The main problem with this is the downtime that you will experience. Because right in the middle of putting a stop to your old release and starting up your new one, there will inevitably be a lot of downtimes.

This downtime is occurring whether you are operating under one millisecond or a whole minute. As soon as a new container starts up the new service is initiated. But don't fret my friends because this problem can easily be alleviated. All you have to do is set up a proxy that has its own individual health check. Don't be surprised however that this will require you to run a multiple instances of the service. This way you can then stop one instance and then bring the other into place.

That way when the original instance is running you can then replace it with your other instance. You have to be careful, however because this process is known for being rather complicated and able to keep you from repeating the process in other situations and may even interfere with your "Compose Scale" command. Another solution to this issue is to simply deploy a new release by utilizing your blue-green deployment process.

This way you can deeply the proxy by simply requesting it's old form of release. With this deployment finished, the proxy should be then reconstituted and sent back with the other requests so that this service can be used to its full extent until the previous application is stopped. If you use Docker for any meaningful length of time, sooner or later you will encounter issues with your proxy reconfiguration.

This is due to the sheer immutability achieved from the cluster orchestrators such as Swarm and result in major monolithic applications. Because even if you do not impose direct and immediate deployment, your microservices will then start being used up much more often. This can be a major drawback and cause problems later on.

So whether your use tally's up to many times a day, once a week, or just once a day, you will need to keep track of it and configure our proxy accordingly during deployment. This is where the swarm is highly useful because it will proxy its needs on demand. These are just some of the most common problems that you will face as you got to know your Docker.

Conclusion: The Ease of Docker

There are a lot of reasons to like Docker. It is intuitive, it is easy to use, and it will streamline your whole database. But what I like best about Docker is the way that you can customize how and who accesses the system.

Because as it is, this is an uncertain world, and you can only be sure your work stays secure if you have a platform like Docker keeping track of all of your containers and who is accessing them.

And as highlighted in this book, once you know the basics, the fully automated platform of Docker practically docks all by itself.