NEW FOR 2016!

Docker

Creating Structured Containers

CURATED COURSE

[PACKT]

Docker

Creating Structured Containers

A course in five modules

Rethink what's possible with Docker and optimize its power
with your Course Guide Ankita Thakur

Learn to use the next-platform Docker from start to finish!

To contact your Course Guide
Email: ankitat@packtpub.com

[PACKT]

BIRMINGHAM - MUMBAI

Meet Your Course Guide

Hello and welcome to this Docker course! You now have a clear pathway to become proficient in Docker.

This course has been planned and created for you by me Ankita Thakur – I am your Course Guide, and I am here to help you have a great journey along the pathways of learning that I have planned for you.

I've developed and created this course for you and you'll be seeing me through the whole journey, offering you my thoughts and ideas behind what you're going to learn next and why I recommend each step. I'll provide tests and quizzes to help you reflect on your learning, and code challenges that will be pitched just right for you through the course.

If you have any questions along the way, you can reach out to me over e-mail or telephone and I'll make sure you get everything from the course that we've planned. Details of how to contact me are included on the first page of this course.

What's so cool about Docker?

So hot off the presses, the latest buzz that has been on the tip of everyone's tongues and the topic of almost any conversation that includes containers these days is Docker! With this course, you will go from just being the person in the office who hears that buzz to the one who is tooting it around every day. Your fellow office workers will be flocking to you for anything related to Docker and shower you with gifts—well, maybe not gifts, but definitely tapping your brain for knowledge!

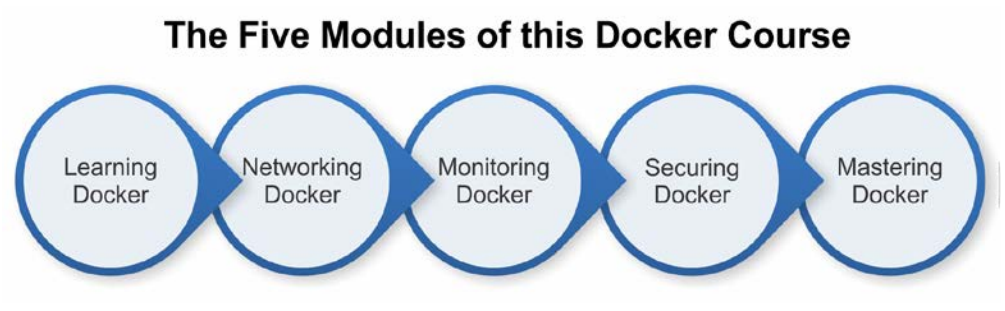
The popular Docker containerization platform has come up with an enabling engine to simplify and accelerate the life cycle management of containers. There are industry-strength and openly automated tools made freely available to facilitate the needs of container networking and orchestration. Therefore, producing and sustaining business-critical distributed applications is becoming easy. Business workloads are methodically containerized to be easily taken to cloud environments, and they are exposed for container crafters and composers to bring forth cloud-based software solutions and services. Precisely speaking, containers are turning out to be the most featured, favored, and fine-tuned runtime environment for IT and business services.

What's in it for me – Course Structure

Docker has been a game-changer when it comes to virtualization. It has now grown to become a key driver of innovation beyond system administration. It is now having an impact on the world of web development and beyond. But how can you make sure you're keeping up with the innovations that it's driving? How can you be sure you're using it to its full potential? This course is meticulously designed and developed in order to empower developers, cloud architects, sysadmins, business managers, and strategists, with all the right and relevant information on the Docker platform and its capacity to power up mission-critical, composite, and distributed applications across industry verticals.

However, I want to highlight that the road ahead may be bumpy on occasions, and some topics may be more challenging than others, but I hope that you will embrace this opportunity and focus on the reward. Remember that we are on this journey together, and throughout this course, we will add many powerful techniques to your arsenal that will help us solve the problems.

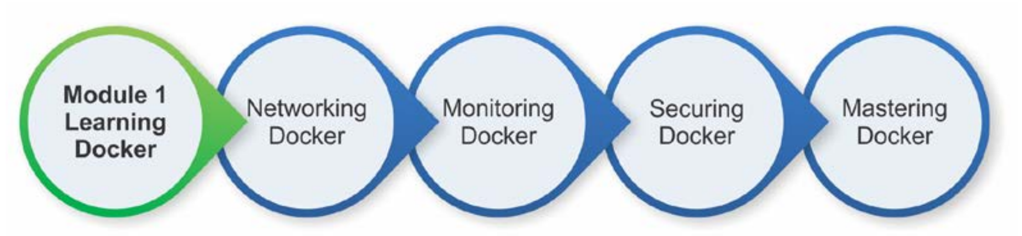
I've created this learning path for you that consists of five models. Each of these modules is a mini-course in their own way, and as you complete each one, you'll have gained key skills and be ready for the material in the next module.



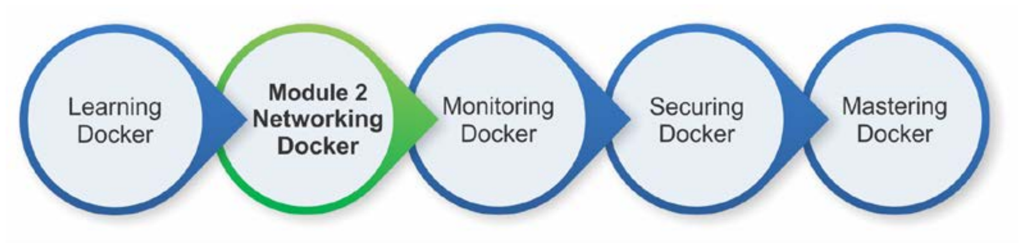
So let's now look at the pathway these modules create—basically all the topics that will be exploring in this learning journey.

Course Journey

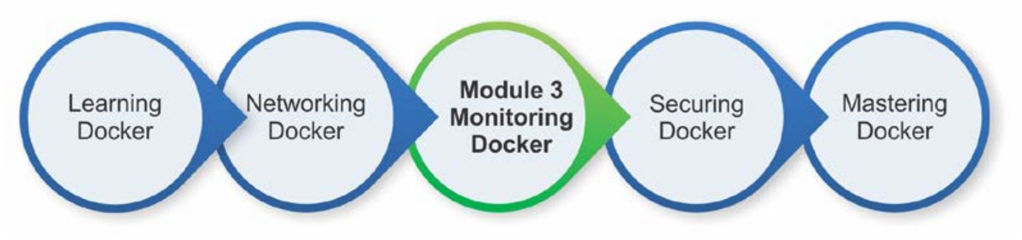
We start the course with our very first module, *Learning Docker*, to help you get familiar with Docker. This module is a step-by-step guide that will walk you through the various features of Docker from Docker software installation to knowing Docker in detail. It will cover best practices to make sure you're confident with the basics, such as building, managing, and storing containers, before diving deeper into advanced topics of Docker.



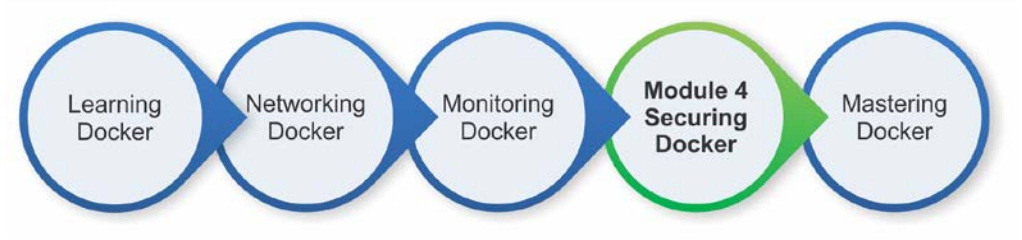
Docker provides the networking primitives that allow administrators to specify how different containers network with each application and connect each of its components, then distribute them across a large number of servers and ensure coordination between them irrespective of the host or VM they are running in. The second module, *Networking Docker*, will show you how to create, deploy, and manage a virtual network for connecting containers spanning single or multiple hosts.



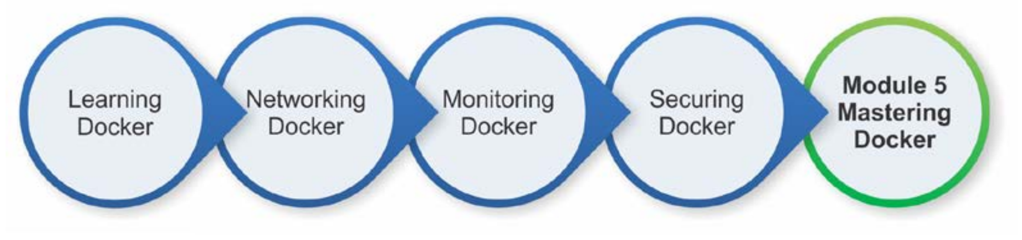
Next, we come to our third module, *Monitoring Docker*. This module will show you how monitoring containers and keeping a keen eye on the working of applications helps improve the overall performance of the applications that run on Docker. This module will cover monitoring containers using Docker's native monitoring functions, various plugins, as well as third-party tools that help in monitoring.



With the rising integration and adoption of Docker containers, there is a growing need to ensure their security. The purpose of our fourth module, *Securing Docker*, is to provide techniques and enhance your skills to secure Docker containers easily and efficiently. It will share the techniques to configure Docker components securely and explore the different security measures/methods one can use to secure the kernel. Furthermore, it will cover the best practices of reporting Docker security findings and will show you how you can safely report any security findings you come across.



Finally, the last module—*Mastering Docker*! Now that you've learned the nitty-gritty of Docker, it's time to take a step ahead and learn some advanced topics. This module will help you deploy Docker in production. You also learn three interesting GUI applications: Shipyard, Panamax, and Tutum.



Ankita Thakur

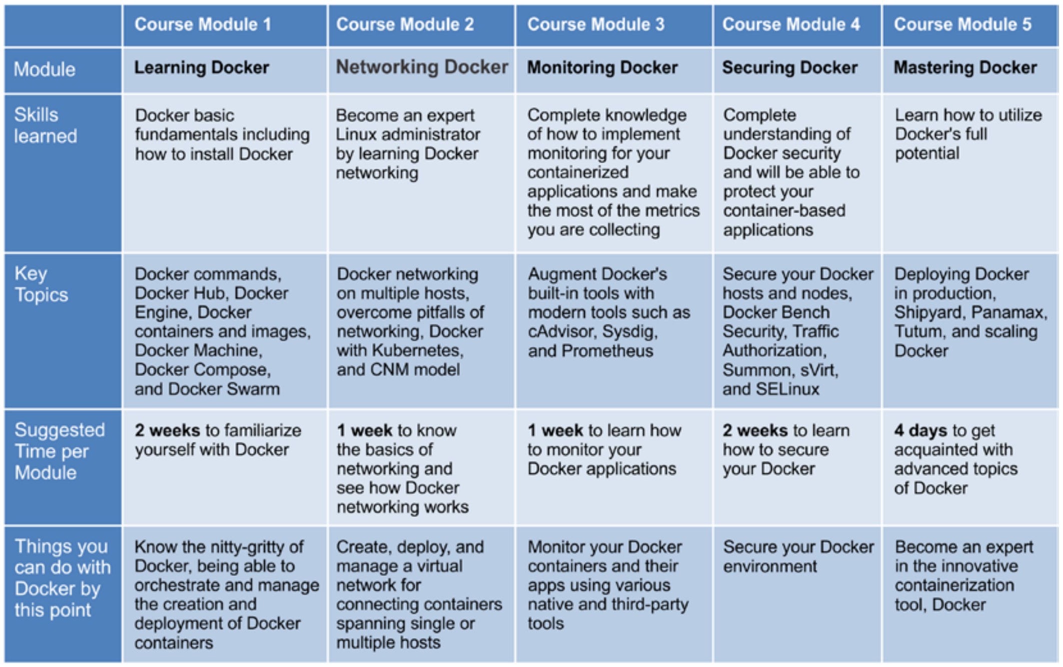
Your Course Guide

Did you know that average salary for Docker jobs is about $28, 000?

Docker is an emerging field and there is a high demand of people who knows this booming technology.

The Course Roadmap and Timeline

Here's a view of the entire course plan before we begin. This grid gives you a topic overview of the whole course and its modules, so you can see how we will move through particular phases of learning to use Docker, what skills you'll be learning along the way, and what you can do with those skills at each point. I also offer you an estimate of the time you might want to take for each module, although a lot depends on your learning style how much you're able to give the course each week!



	Course Module 1	Course Module 2	Course Module 3	Course Module 4	Course Module 5
Module	**Learning Docker**	**Networking Docker**	**Monitoring Docker**	**Securing Docker**	**Mastering Docker**
Skills learned	Docker basic fundamentals including how to install Docker	Become an expert Linux administrator by learning Docker networking	Complete knowledge of how to implement monitoring for your containerized applications and make the most of the metrics you are collecting	Complete understanding of Docker security and will be able to protect your container-based applications	Learn how to utilize Docker's full potential
Key Topics	Docker commands, Docker Hub, Docker Engine, Docker containers and images, Docker Machine, Docker Compose, and Docker Swarm	Docker networking on multiple hosts, overcome pitfalls of networking, Docker with Kubernetes, and CNM model	Augment Docker's built-in tools with modern tools such as cAdvisor, Sysdig, and Prometheus	Secure your Docker hosts and nodes, Docker Bench Security, Traffic Authorization, Summon, sVirt, and SELinux	Deploying Docker in production, Shipyard, Panamax, Tutum, and scaling Docker
Suggested Time per Module	**2 weeks** to familiarize yourself with Docker	**1 week** to know the basics of networking and see how Docker networking works	**1 week** to learn how to monitor your Docker applications	**2 weeks** to learn how to secure your Docker	**4 days** to get acquainted with advanced topics of Docker
Things you can do with Docker by this point	Know the nitty-gritty of Docker, being able to orchestrate and manage the creation and deployment of Docker containers	Create, deploy, and manage a virtual network for connecting containers spanning single or multiple hosts	Monitor your Docker containers and their apps using various native and third-party tools	Secure your Docker environment	Become an expert in the innovative containerization tool, Docker

Table of Contents

Course Module 4: Securing Docker

Course Module 1
Learning Docker

Ankita Thakur

Your Course Guide

Optimize the power of Docker to run your applications quickly and easily with Course Module 1, Learning Docker

Course Module 1
Learning Docker

Course Module 1

Module 1 Learning Docker
Networking Docker
Monitoring Docker
Securing Docker
Mastering Docker

Docker is a next-generation platform for simplifying application containerization life-cycle. Docker allows you to create a robust and resilient environment in which you can generate portable, composable, scalable, and stable application containers. Let's begin this learning journey with the first module of the course, *Learning Docker*, and get started with Docker—the Linux containerizing technology that has revolutionized application sandboxing.

Ankita Thakur

Your Course Guide

We'll start off by elucidating the installation procedure for Docker and a few troubleshooting techniques. You will be introduced to the process of downloading Docker images and running them as containers. You'll learn how to run **containers as a service (CaaS)** and also discover how to share data among containers. This module will teach you how to use Docker Machine to build new servers from scratch. You'll learn how to take greater control over your containers using some of Docker's most sophisticated and useful tools, such as Docker Compose and Docker Swarm. You'll explore how to establish the link between containers and orchestrate containers using Docker Compose. You will also come across relevant details about application testing inside a container. You will discover how to debug a container using the docker exec command and the nsenter tool. By the end of this module, I'm sure you'll become fluent with the basic components of Docker.

What are you looking for? Let's get started...

1
Getting Started with Docker

These days, Docker technology is gaining more market and more mind shares among information technology (IT) professionals across the globe. In this chapter, we would like to shed more light on Docker, and show why it is being touted as the next best thing for the impending cloud IT era. In order to make this book relevant to software engineers, we have listed the steps needed for crafting highly usable application-aware containers, registering them in a public registry repository, and then deploying them in multiple IT environments (on-premises as well as off-premises). In this book, we have clearly explained the prerequisites and the most important details of Docker, with the help of all the education and experiences that we could gain through a series of careful implementations of several useful Docker containers in different systems. For doing this, we used our own laptops as well as a few leading public **Cloud Service Providers** (**CSP**).

We would like to introduce you to the practical side of Docker for the game-changing Docker-inspired containerization movement.

In this chapter, we will cover the following topics:

- An introduction to Docker
- Docker on Linux
- Differentiating between containerization and virtualization
- Installing the Docker engine
- Understanding the Docker setup
- Downloading the first image
- Running the first container
- Running a Docker container on **Amazon Web Services** (**AWS**)
- Troubleshooting the Docker containers

An introduction to Docker

Due to its overwhelming usage across industry verticals, the IT domain has been stuffed with many new and path-breaking technologies used not only for bringing in more decisive automation but also for overcoming existing complexities.

Virtualization has set the goal of bringing forth IT infrastructure optimization and portability. However, virtualization technology has serious drawbacks, such as performance degradation due to the heavyweight nature of **virtual machines** (**VM**), the lack of application portability, slowness in provisioning of IT resources, and so on. Therefore, the IT industry has been steadily embarking on a Docker-inspired containerization journey. The Docker initiative has been specifically designed for making the containerization paradigm easier to grasp and use. Docker enables the containerization process to be accomplished in a risk-free and accelerated fashion.

Precisely speaking, **Docker** is an open source containerization engine, which automates the packaging, shipping, and deployment of any software applications that are presented as lightweight, portable, and self-sufficient containers, that will run virtually anywhere.

A Docker **container** is a software bucket comprising everything necessary to run the software independently. There can be multiple Docker containers in a single machine and containers are completely isolated from one another as well as from the host machine.

In other words, a Docker container includes a software component along with all of its dependencies (binaries, libraries, configuration files, scripts, jars, and so on). Therefore, the Docker containers could be fluently run on x64 Linux kernel supporting namespaces, control groups, and file systems, such as **Another Union File System** (**AUFS**). AUFS is a layered copy-on-write file system that shares common portions of the operating system between containers.

There have been many tools and technologies aimed at making distributed applications possible, even easy to set up, but none of them have as wide an appeal as Docker does, which is primarily because of its cross-platform nature and friendliness towards both system administrators and developers. It is possible to set up Docker in any OS, be it Windows, OS X, or Linux, and Docker containers work the same way everywhere. This is extremely powerful, as it enables a write-once-run anywhere workflow. Docker containers are guaranteed to run the same way, be it on your development desktop, a bare-metal server, virtual machine, data center, or cloud. No longer do you have the situation where a program runs on the developer's laptop but not on the server.

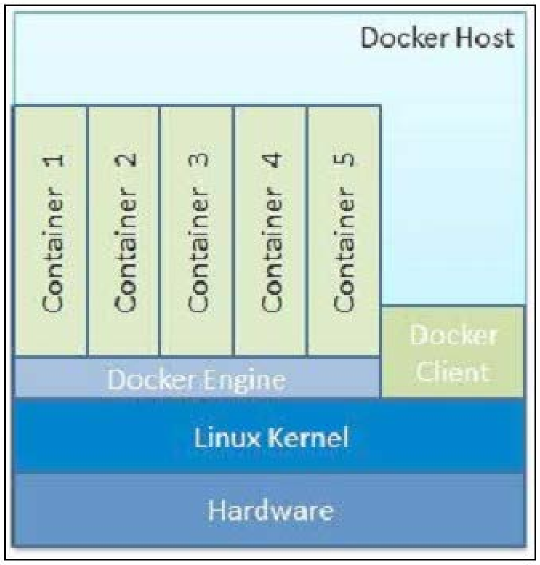
In a nutshell, the Docker solution lets us quickly assemble composite, enterprise-scale, and business-critical applications. For doing this, we can use different and distributed software components: Containers eliminate the friction that comes with shipping code to distant locations. Docker also lets us test the code and then deploy it in production as fast as possible. The Docker solution primarily consists of the following components:

- The Docker engine
- The Docker Hub

The Docker engine is for enabling the realization of purpose-specific as well as generic Docker containers. The Docker Hub is a fast-growing repository of the Docker images that can be combined in different ways for producing publicly findable, network-accessible, and widely usable containers.

Docker on Linux

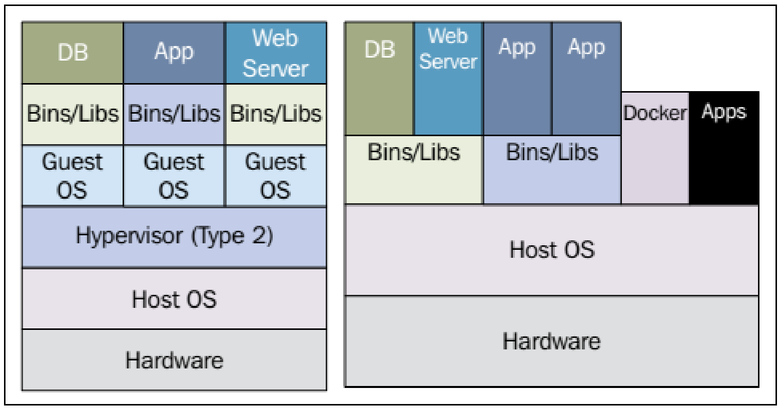
Supposethat we want to directly run the containers on a Linux machine. The Docker engine produces, monitors, and manages multiple containers as illustrated in the following diagram:



The preceding diagram vividly illustrates how future IT systems would have hundreds of application-aware containers, which would innately be capable of facilitating their seamless integration and orchestration for deriving modular applications (business, social, mobile, analytical, and embedded solutions). These contained applications could fluently run on converged, federated, virtualized, shared, dedicated, and automated infrastructures.

Differentiating between containerization and virtualization

It is pertinent, and paramount to extract and expound the game-changing advantages of the Docker-inspired containerization movement over the widely used and fully matured virtualization paradigm. In the containerization paradigm, strategically sound optimizations have been accomplished through a few crucial and well-defined rationalizations and the insightful sharing of the compute resources. Some of the innate and hitherto underutilized capabilities of the Linux kernel have been rediscovered. These capabilities have been rewarded for bringing in much-wanted automation and acceleration, which will enable the fledgling containerization idea to reach greater heights in the days ahead, especially those of the cloud era. The noteworthy business and technical advantages of these include the bare metal-scale performance, real-time scalability, higher availability, and so on. All the unwanted bulges and flab are being sagaciously eliminated to speed up the roll-out of hundreds of application containers in seconds and to reduce the time taken for marketing and valuing in a cost-effective fashion. The following diagram on the left-hand side depicts the virtualization aspect, whereas the diagram on the right-hand side vividly illustrates the simplifications that are being achieved in the containers:



The following table gives a direct comparison between virtual machines and containers:

Virtual Machines (VMs)	**Containers**
Represents hardware-level virtualization	Represents operating system virtualization
Heavyweight	Lightweight
Slow provisioning	Real-time provisioning and scalability
Limited performance	Native performance
Fully isolated and hence more secure	Process-level isolation and hence less secure

The convergence of containerization and virtualization

A hybrid model, having features from both the virtual machines and that of containers, is being developed. It is the emergence of system containers, as illustrated in the preceding right-hand-side diagram. Traditional hypervisors, which implicitly represent hardware virtualization, directly secure the environment with the help of the server hardware. That is, VMs are completely isolated from the other VMs as well as from the underlying system. But for containers, this isolation happens at the process level and hence, they are liable for any kind of security incursion. Furthermore, some vital features that are available in the VMs are not available in the containers. For instance, there is no support for system services like SSH. On the other hand, VMs are resource-hungry and hence, their performance gets substantially degraded. Indeed, in containerization parlance, the overhead of a classic hypervisor and a guest operating system will be eliminated to achieve bare metal performance. Therefore, a few VMs can be provisioned and made available to work on a single machine. Thus, on one hand, we have the fully isolated VMs with average performance and on the other side, we have the containers that lack some of the key features, but are blessed with high performance. Having understood the ensuing needs, product vendors are working on system containers. The objective of this new initiative is to provide full system containers with the performance that you would expect from bare metal servers, but with the experience of virtual machines. The system containers in the preceding right-hand-side diagram represent the convergence of two important concepts (virtualization and containerization) for smarter IT. We will hear and read more about this blending in the future.

Containerization technologies

Having recognized the role and the relevance of the containerization paradigm for IT infrastructure augmentation and acceleration, a few technologies that leverage the unique and decisive impacts of the containerization idea have come into existence and they have been enumerated as follows:

- **LXC(Linux Containers)**: Thisis the father of all kinds of containers and it represents an operating-system-level virtualization environment for running multiple isolated Linux systems (containers) on a single Linux machine. The article LXCon the Wikipedia website states that:

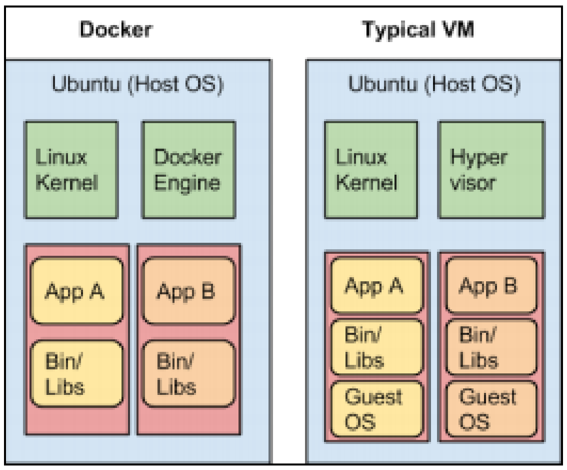
    "The Linux kernel provides the cgroups functionality that allows limitation and prioritization of resources (CPU, memory, block I/O, network, etc.) without the need for starting any virtual machines, and namespace isolation functionality that allows complete isolation of an applications' view of the operating environment, including process trees, networking, user IDs and mounted file systems."

    You can get more information from `http://en.wikipedia.org/wiki/LXC`

- **OpenVZ**: This is an OS-level virtualization technology based on the Linux kernel and the operating system. OpenVZ allows a physical server to run multiple isolated operating system instances, called containers, **virtual private servers** (**VPSs**), or **virtual environments** (**VEs**).
- **The FreeBSD jail**: This is a mechanism that implements an OS-level virtualization, which lets the administrators partition a FreeBSD-based computer system into several independent mini-systems called jails.
- **The AIX Workload partitions(WPARs)**: These are the software implementations of the OS-level virtualization technology, which provide application environment isolation and resource control.
- **Solaris Containers(including Solaris Zones)**: This is an implementation of the OS-level virtualization technology for the x86 and SPARC systems. A Solaris Container is a combination of the system resource controls and boundary separation provided by zones. Zones act as completely isolated virtual servers within a single operating system instance.

Docker networking/linking

Another important aspect that needs to be understood is how Docker containers are networked or linked together. The way they are networked or linked together highlights another important and large benefit of Docker. When a container is created, it creates a bridge network adapter for which it is assigns an address; it is through these network adapters that the communication flows when you link containers together. Docker doesn't have the need to expose ports to link containers. Let's take a look at it with the help of the following illustration:



In the preceding illustration, we can see that the typical VM has to expose ports for others to be able to communicate with each other. This can be dangerous if you don't set up your firewalls or, in this case with MySQL, your MySQL permissions correctly. This can also cause unwanted traffic to the open ports. In the case of Docker, you can link your containers together, so there is no need to expose the ports. This adds security to your setup, as there is now a secure connection between your containers.

Ankita Thakur

Your Course Guide

Reflect and Test Yourself!

Q1. Which of the following statement is incorrect about containers?

1. It represents operating system virtualization
2. It is lightweight
3. It gives limited performance
4. It is less secure

Installing Docker

The Docker engine is built on top of the Linux kernel and it extensively leverages its features. Therefore, at this point in time, the Docker engine can only be directly run on Linux OS distributions. Nonetheless, the Docker engine could be run on the Mac and Microsoft Windows operating systems by using the lightweight Linux VMs with the help of adapters, such as Boot2Docker. Due to the surging growing of Docker, it is now being packaged by all major Linux distributions so that they can retain their loyal users as well as attract new users. You can install the Docker engine by using the corresponding packaging tool of the Linux distribution; for example, by using the `apt-get` command for Debian and Ubuntu, and the `yum` command for Red Hat, Fedora, and CentOS. You can look up the instructions for your operating system at `https://docs.docker.com/installation/#installation`.

> Note that Docker is called `docker.io` here and just `docker` on other platforms since Ubuntu (and Debian) already has a package named `docker`. Therefore, all the files with the name `docker` are installed as `docker.io`.
>
> Examples are `/usr/bin/docker.io` and `/etc/bash_completion.d/docker.io`.

Installing Docker from the Ubuntu package repository

Docker is supported by Ubuntu from Ubuntu 12.04 onwards. Remember that you still need a 64-bit operating system to run Docker. This section explains the steps involved in installing the Docker engine from the Ubuntu package repository in detail. To install the Ubuntu packaged version, follow these steps:

1. The best practice for installing the Ubuntu packaged version is to begin the installation process by resynchronizing with the Ubuntu package repository. This step will essentially update the package repository to the latest published packages, thus we will ensure that we always get the latest published version by using the command shown here:

   ```
   $ sudo apt-get update
   ```

2. Kick-start the installation by using the following command. This setup will install the Docker engine along with a few more support files, and it will also start the `docker` service instantaneously:

   ```
   $ sudo apt-get install -y docker.io
   ```

That's it! You have now installed Docker onto your system. Remember that the command has been renamed `docker.io`, so you will have to run all Docker commands with `docker.io` instead of `docker`. However, for your convenience, you can create a soft link for `docker.io` called `docker`. This will enable you to execute Docker commands as `docker` instead of `docker.io`. You can do this by using the following command:

```
$ sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker
```

Downloading the example code

The code files for this course are available at `https://github.com/EdwinMoses/Docker-Code`.

Installing the latest Docker using docker.io script

The official distributions might not package the latest version of Docker. In such a case, you can install the latest version of Docker either manually or by using the automated scripts provided by the Docker community.

For installing the latest version of Docker manually, follow these steps:

1. Add the Docker release tool's repository path to your APT sources, as shown here:

   ```
   $ sudo sh -c "echo deb https://get.docker.io/ubuntu \
   docker main > /etc/apt/sources.list.d/docker.list"
   ```

2. Import the Docker release tool's public key by running the following command:

   ```
   $ sudo apt-key adv --keyserver \
   hkp://keyserver.ubuntu.com:80 --recv-keys \
   36A1D7869245C8950F966E92D8576A8BA88D21E9
   ```

3. Resynchronize with the package repository by using the command shown here:

   ```
   $ sudo apt-get update
   ```

4. Install `docker` and then start the `docker` service.

   ```
   $ sudo apt-get install -y lxc-docker
   ```

> The `lxc-docker` command will install the Docker image using the name `docker`.

The Docker community has taken a step forward by hiding these details in an automated install script. This script enables the installation of Docker on most of the popular Linux distributions, either through the `curl` command or through the `wget` command, as shown here:

- For the `curl` command:
  ```
  $ sudo curl -sSL https://get.docker.io/ | sh
  ```
- For the `wget` command:
  ```
  $ sudo wget -qO- https://get.docker.io/ | sh
  ```

> The preceding automated script approach enforces AUFS as the underlying Docker file system. This script probes the AUFS driver, and then installs it automatically if it is not found in the system. In addition, it also conducts some basic tests upon installation for verifying the sanity.

Upgrading Docker

Now that we have Docker installed, we can get going at full steam! There is one problem though: software repositories like APT are usually behind times and often have older versions. Docker is a fast-moving project and a lot has changed in the last few versions. So it is always recommended to have the latest version installed. At the time of writing this, the latest version of Docker was 1.10.0.

To check and download upgrades, all you have to do is to execute this command in a terminal:

```
sudo apt-get update && sudo apt-get upgrade
```

You can upgrade Docker as and when it is updated in the APT repositories. An alternative (and better) method is to build from source. The best way to remain updated is to regularly get the latest version from the public GitHub repository. Traditionally, building software from a source has been painful and done only by people who actually work on the project. This is not so with Docker. From Docker 0.6, it has been possible to build Docker in Docker. This means that upgrading Docker is as simple as building a new version in Docker itself and replacing the binary. Let's see how this is done.

You need to have the following tools installed in a 64-bit Linux machine (VM or bare-metal) to build Docker:

- **Git**: It is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It is used here to clone the Docker public source code repository. Check out `git-scm.org` for more details.
- **Make**: This utility is a software engineering tool used to manage and maintain computer programs. Make provides most help when the program consists of many component files. A `Makefile` file is used here to kick off the Docker containers in a repeatable and consistent way.

Building Docker from source

To build Docker in Docker, we will first fetch the source code and then run a few `make` commands that will, in the end, create a `docker` binary, which will replace the current binary in the Docker installation path.

Run the following command in your terminal:

```
$ git clone https://git@github.com/dotcloud/docker
```

This command clones the official Docker source code repository from the GitHub repository into a directory named `docker`:

```
$ cd docker
$ sudo make build
```

This will prepare the development environment and install all the dependencies required to create the binary. This might take some time on the first run, so you can go and have a cup of coffee.

If you encounter any errors that you find difficult to debug, you can always go to `#docker` on freenode IRC. The developers and the Docker community are very helpful.

Now we are ready to compile that binary:

```
$ sudo make binary
```

This will compile a binary and place it in the `./bundles/<version>-dev/binary/` directory. And voila! You have a fresh version of Docker ready.

Before replacing your existing binary though, run the tests:

```
$ sudo make test
```

If the tests pass, then it is safe to replace your current binary with the one you've just compiled. Stop the `docker` service, create a backup of the existing binary, and then copy the freshly baked binary in its place:

```
$ sudo service docker stop
$ alias wd='which docker'
$ sudo cp $(wd) $(wd)_
$ sudo cp $(pwd)/bundles/<version>-dev/binary/docker-<version>-dev $(wd)
$ sudo service docker start
```

Congratulations! You now have the up-to-date version of Docker running.

OS X and Windows users can follow the same procedures as SSH in the boot2Docker VM.

User permissions

Create a group called `docker` and add your user to that group to avoid having to add the `sudo` prefix to every `docker` command. The reason you need to run a `docker` command with the `sudo` prefix by default is that the `docker` daemon needs to run with root privileges, but the `docker` client (the commands you run) doesn't. So, by creating a `docker` group, you can run all the client commands without using the `sudo` prefix, whereas the daemon runs with the root privileges:

```
$ sudo groupadd docker # Adds the docker group
$ sudo gpasswd -a $(whoami) docker # Adds the current user to the group
$ sudo service docker restart
```

You might need to log out and log in again for the changes to take effect.

UFW settings

Docker uses a bridge to manage network in the container. **Uncomplicated Firewall** (**UFW**) is the default firewall tool in Ubuntu. It drops all forwarding traffic. You will need to enable forwarding like this:

```
$ sudo vim /etc/default/ufw
# Change:
```

```
# DEFAULT_FORWARD_POLICY="DROP"
# to
DEFAULT_FORWARD_POLICY="ACCEPT"
```

Reload the firewall by running the following command:

```
$ sudo ufw reload
```

Alternatively, if you want to be able to reach your containers from other hosts, then you should enable incoming connections on the Docker port (default `2375`):

```
$ sudo ufw allow 2375/tcp
```

You do not need to follow user permission and UFW settings if you are using boot2Docker.

Installing Docker on Mac OS X

To be able to use Docker on Mac OS X, we have to run the Docker service inside a **virtual machine** (**VM**) since Docker uses Linux-specific features to run. We don't have to get frightened by this since the installation process is very short and straightforward.

Installation

There is an OS X installer that installs everything we need, that is, VirtualBox, boot2docker, and Docker.

VirtualBox is a virtualizer in which we can run the lightweight Linux distribution, and boot2docker is a virtual machine that runs completely in the RAM and occupies just about 27 MB of space.

The latest released version of the OS X installer can be found at `https://github.com/boot2docker/osx-installer/releases/latest`.

Now, let's take a look at how the installation should be done with the following steps:

1. Download the installer by clicking on the button named `Boot2Docker-1.x.0.pkg` to get the `.pkg` file.
2. Double-click on the downloaded `.pkg` file and go through with the installation process.
3. Open the **Finder** window and navigate to your `Applications` folder; locate `boot2docker` and double-click on it. A terminal window will open and issue a few commands.

   This runs a Linux VM, named `boot2docker-vm`, that has Docker pre-installed in VirtualBox. The Docker service in the VM runs in daemon (background) mode, and a Docker client is installed in OS X, which communicates directly with the Docker daemon inside the VM via the Docker Remote API.

4. You will see the following output, which tells you to set some environment variables:

   ```
   To connect the Docker client to the Docker daemon, please set:
       export DOCKER_HOST=tcp://192.168.59.103:2376
       export DOCKER_CERT_PATH=/Users/oscarhane/.boot2docker/certs/
   boot2docker-vm
       export DOCKER_TLS_VERIFY=1
   ```

   We open up the `~/.bash_profile` file and paste three lines from our output, as follows, at the end of this file:

   ```
   export DOCKER_HOST=tcp://192.168.59.103:2376
   export.DOCKER_CERT_PATH=/Users/xxx/.boot2docker/certs/boot2docker-
   vm
   export DOCKER_TLS_VERIFY=1
   ```

The reason why we do this is so that our Docker client knows where to find the Docker daemon. If you want to find the IP in the future, you can do so by executing the `boot2docker ip` command. Adding the preceding lines will set these variables every time a new terminal session starts. When you're done, close the terminal window. Then, open a new window and type `echo $DOCKER_HOST` to verify that the environment variable is set as it should be. You should see the IP and port your boot2docker VM printed.

5. Type `docker version` to verify that you can use the Docker command, which will show the Docker version currently installed.

Installing Docker on Windows

Just as we have to install a Linux virtual machine when installing Docker in OS X, we have to do the same in Windows in order to run Docker because of the Linux kernel features that Docker builds on. OS X has a native Docker client that directly communicates with the Docker daemon inside the virtual machine, but there isn't one available for Windows yet.

Installation

There is a Windows installer that installs everything we need in order to run Docker. For this, go to `https://github.com/boot2docker/windows-installer/releases/latest`.

Now, let's take a look at how the installation should be done with the help of the following steps:

1. Click on the **docker-install.exe** button to download the `.exe` file.
2. When the download is complete, run the downloaded installer. Follow through the installation process, and you will get VirtualBox, msysGit, and boot2docker installed.
3. Go to your `Program Files` folder and click on the newly installed `boot2docker` to start using Docker. If you are prompted to enter a passphrase, just press *Enter*.
4. Type `docker version` to verify that you can use the Docker command.

Upgrading Docker on Mac OS X and Windows

A new software changes often and to keep `boot2docker` updated, invoke these commands:

```
boot2docker stop
boot2docker download
boot2docker start
```

Downloading the first Docker image

Having installed the Docker engine successfully, the next logical step is to download the images from the Docker registry. The Docker registry is an application repository, which hosts a range of applications that vary between basic Linux images and advanced applications. The `docker pull` subcommand is used for downloading any number of images from the registry. In this section, we will download a tiny version of Linux called the `busybox` image by using the following command:

```
$ sudo docker pull busybox
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
ea13149945cb: Pull complete
4986bf8c1536: Pull complete
busybox:latest: The image you are pulling has been verified. Important: image verification is a tech preview feature and should not be relied on to provide security.
Status: Downloaded newer image for busybox:latest
```

Once the images have been downloaded, they can be verified by using the `docker images` subcommand, as shown here:

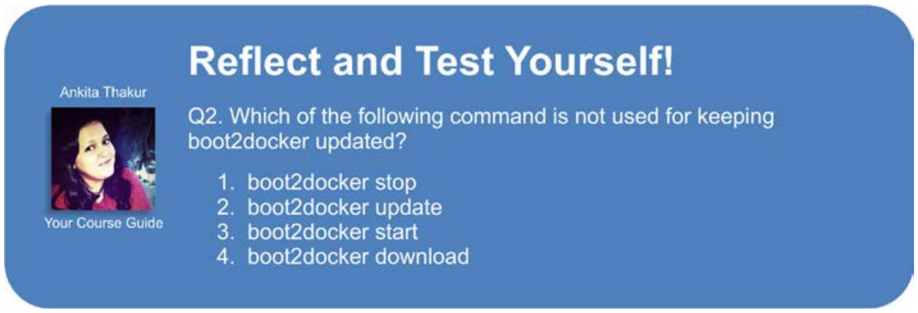
```
$ sudo docker images
REPOSITORY    TAG     IMAGE ID        CREATED      VIRTUAL SIZE
busybox       latest  4986bf8c1536    12 weeks ago 2.433 MB
```

Running the first Docker container

Now, you can start your first Docker container. It is standard practice to start with the basic *Hello World!* application. In the following example, we will echo `Hello World!` by using a `busybox` image, which we have already downloaded, as shown here:

```
$ sudo docker run busybox echo "Hello World!"
"Hello World!"
```

Cool, isn't it? You have set up your first Docker container in no time. In the preceding example, the `docker run` subcommand has been used for creating a container and for printing `Hello World!` by using the `echo` command.



Reflect and Test Yourself!

Ankita Thakur

Your Course Guide

Q2. Which of the following command is not used for keeping boot2docker updated?

1. boot2docker stop
2. boot2docker update
3. boot2docker start
4. boot2docker download

Running a Docker container on Amazon Web Services

Amazon Web Services (**AWS**) announced the availability of Docker containers at the beginning of 2014, as a part of its Elastic Beanstalk offering. At the end of 2014, they revolutionized Docker deployment and provided the users with options shown here for running Docker containers:

- The Amazon EC2 container service (only available in **preview** mode at the time of writing this book)
- Docker deployment by using the Amazon Elastic Beans services

The Amazon EC2 container service lets you start and stop the container-enabled applications with the help of simple API calls. AWS has introduced the concept of a cluster for viewing the state of your containers. You can view the tasks from a centralized service, and it gives you access to many familiar Amazon EC2 features, such as the security groups, the EBS volumes and the IAM roles.

Please note that this service is still not available in the AWS console. You need to install AWS CLI on your machine to deploy, run, and access this service.

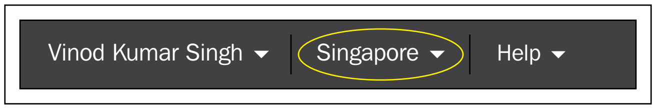
The AWS Elastic Beanstalk service supports the following:

- A single container that supports Elastic Beanstalk by using a console. Currently, it supports the PHP and Python applications.
- A single container that supports Elastic Beanstalk by using a command line tool called *eb*. It supports the same PHP and Python applications.
- Use of multiple container environments by using Elastic beanstalk.

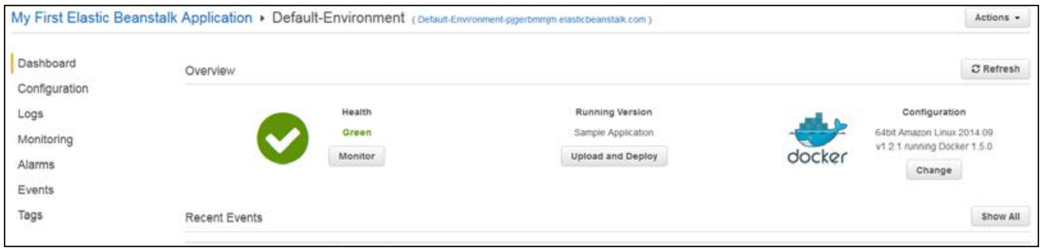
Currently, AWS supports the latest Docker version, which is 1.5.

This section provides a step-by-step process to deploy a sample application on a Docker container running on AWS Elastic Beanstalk.The following are the steps of deployment:

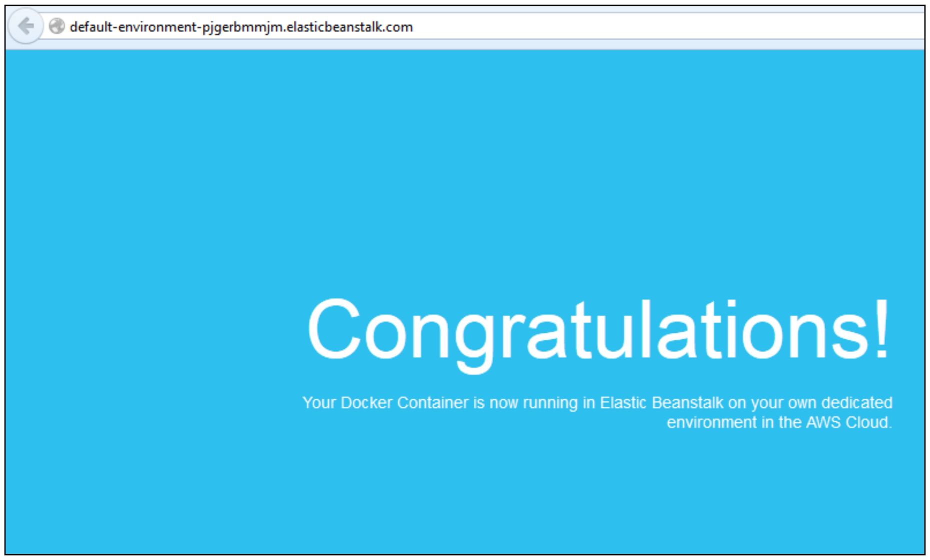
1. Log in to the AWS Elastic Beanstalk console by using this `https://console.aws.amazon.com/elasticbeanstalk/` URL.
2. Select a region where you want to deploy your application, as shown here:



3. Select the **Docker** option, which is in the drop down menu, and then click on **Launch Now**. The next screen will be shown after a few minutes, as shown here:



Now, click on the URL that is next to **Default-Environment (Default-Environment-pjgerbmmjm.elasticbeanstalk.com)**, as shown here:



Congratulations!

Your Docker Container is now running in Elastic Beanstalk on your own dedicated environment in the AWS Cloud.

Troubleshooting

Most of the time, you will not encounter any issues when installing Docker. However, unplanned failures might occur. Therefore, it is necessary to discuss prominent troubleshooting techniques and tips. Let's begin by discussing the troubleshooting knowhow in this section. The first tip is that the running status of Docker should be checked by using the following command:

```
$ sudo service docker status
```

However, if Docker has been installed by using the Ubuntu package, then you will have to use `docker.io` as the service name. If the `docker` service is running, then this command will print the status as `start/running` along with its process ID.

If you are still experiencing issues with the Docker setup, then you could open the Docker log by using the `/var/log/upstart/docker.log` file for further investigation.

Your Coding Challenge

Ankita Thakur

Your Course Guide

Now that we have covered all the basics of controlling your boot2docker VM, you can explore some other way to run Docker containers on your local machine. Kitematic is a recent addition to the Docker portfolio. Up until now, everything we have done has been command line based. With Kitematic, you can manage your Docker containers through a GUI. Kitematic can be used either on Windows or OS X, just not on Linux; besides who needs a GUI on Linux anyways! Try it out.

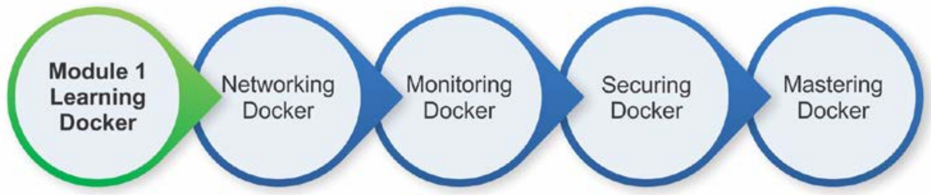
Summary of Module 1 Chapter 1

Ankita Thakur

Your Course Guide

In this chapter, we have covered what basic information you should already know or now know for the chapters ahead. We have gone over the basics of what Docker is and how it is compared to typical virtual machines. Containerization is going to be a dominant and decisive paradigm for the enterprise as well as cloud IT environments in the future because of its hitherto unforeseen automation and acceleration capabilities. There are several mechanisms in place for taking the containerization movement to greater heights. However, Docker has zoomed ahead of everyone in this hot race, and it has successfully decimated the previously-elucidated barriers.

Your Progress through the Course So Far



2
Up and Running

In the last chapter, we set up Docker in our development setup. In this chapter, we will explore the Docker command-line interface.

The following topics will be covered:

- Docker terminologies
- Docker commands
- Dockerfiles
- Docker workflow—pull-use-modify-commit-push workflow

Docker terminologies

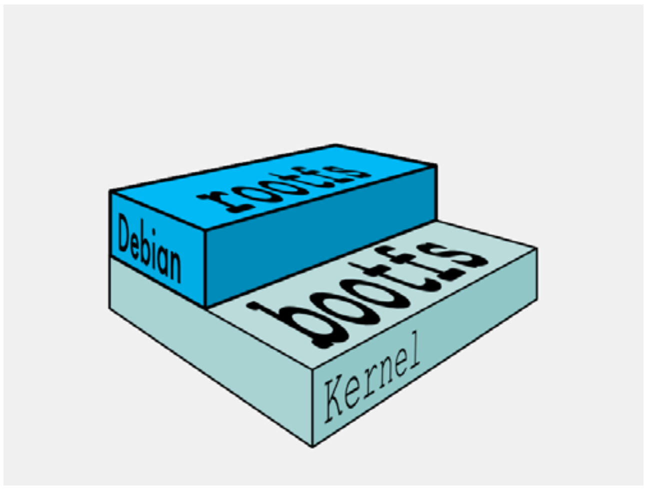
Before we begin our exciting journey into the Docker sphere, let's understand the Docker terminologies that will be used in this book a little better. Very similar in concept to VM images, a Docker image is a snapshot of a system. The difference between a VM image and a Docker image is that a VM image can have running services, whereas a Docker image is just a filesystem snapshot, which means that while you can configure the image to have your favorite packages, you can run only one command in the container. Don't fret though, since the limitation is one command, not one process, so there are ways to get a Docker container to do almost anything a VM instance can.

Docker has also implemented a Git-like distributed version management system for Docker images. Images can be stored in repositories (called a registry), both locally and remotely. The functionalities and terminologies borrow heavily from Git—snapshots are called commits, you pull an image repository, you push your local image to a repository, and so on.

Docker images and containers

A **Docker image** is a collection of all of the files that make up a software application. Each change that is made to the original image is stored in a separate layer. To be precise, any Docker image has to originate from a base image according to the various requirements. Additional modules can be attached to the base image for deriving the various images that can exhibit the preferred behavior. Each time you commit to a Docker image, you are creating a new layer on the Docker image, but the original image and each pre-existing layer remains unchanged. In other words, images are typically of the read-only type. If they are empowered through the systematic attachment of newer modules, then a fresh image will be created with a new name. The Docker images are turning out to be a viable base for developing and deploying the Docker containers.

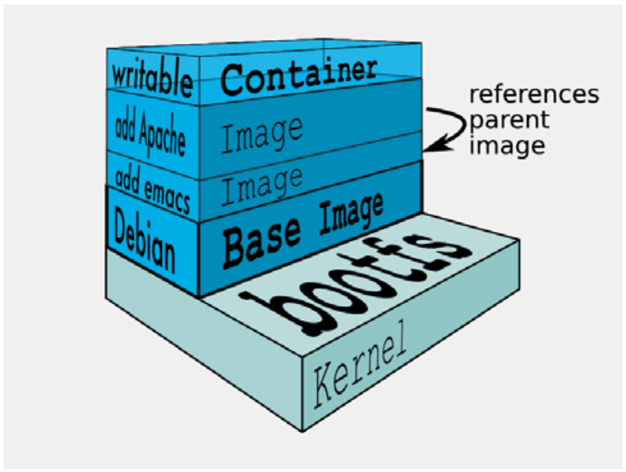
A base image has been illustrated here. Debian is the base image, and a variety of desired capabilities in the form of functional modules can be incorporated on the base image for arriving at multiple images:



Every image has a unique `ID`, as explained in the following section. The base images can be enhanced such that they can create the parent images, which in turn can be used for creating the child images. The base image does not have any parent, that is, the parent images sit on top of the base image. When we work with an image and if we don't specify that image through an appropriate identity (say, a new name), then the `latest` image (recently generated) will always be identified and used by the Docker engine.

As per the Docker home page, a Docker image has a read-only template. For example, an image could contain an Ubuntu operating system, with Apache and your web application installed on it. Docker provides a simple way for building new images or of updating the existing images. You can also download the Docker images that the other people have already created. The Docker images are the building components of the Docker containers. In general, the base Docker image represents an operating system, and in the case of Linux, the base image can be one of its distributions, such as Debian. Adding additional modules to the base image ultimately dawns a container. The easiest way of thinking about a container is as the read-write layer that sits on more read-only images. When the container is run, the Docker engine not only merges all of the required images together, but it also merges the changes from the read-write layer into the container itself. This makes it a self-contained, extensible, and executable system. The changes can be merged by using the Docker `docker commit` subcommand. The new container will accommodate all the changes that are made to the base image. The new image will form a new layer on top of the base image.

The following diagram will tell you everything clearly. The base image is the **Debian** distribution, then there is an addition of two images (the **emacs** and the **Apache** server), and this will result in the container:



Each commit invariably makes a new image. This makes the number of images go up steadily, and so managing them becomes a complicated affair. However, the storage space is not a big challenge because the new image that is generated is only comprised of the newly added modules. In a way, this is similar to the popular object storage in the cloud environments. Every time you update an object, there will be a new object that gets created with the latest modification and then it is stored with a new `ID`. In the case of object storage, the storage size balloons significantly.

A Docker layer

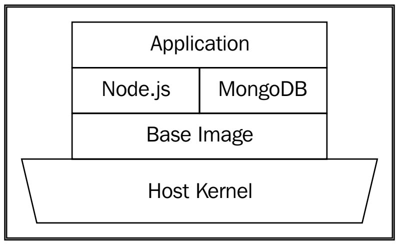
A **Docker layer** could represent either read-only images or read-write images. However, the top layer of a container stack is always the read-write (writable) layer, which hosts a Docker container.

A Docker container

From the preceding diagram, it is clear that the read-write layer is the container layer. There could be several read-only images beneath the container layer. Typically, a container originates from a read-only image through the act of a `commit`. When you `start` a container, you actually refer to an image through its unique `ID`. Docker pulls the required image and its parent image. It continues to pull all the parent images until it reaches the base image.

A Docker container can be correlated to an instance of a VM. It runs sandboxed processes that share the same kernel as the host. The term **container** comes from the concept of shipping containers. The idea is that you can ship containers from your development environment to the deployment environment and the applications running in the containers will behave the same way no matter where you run them.

The following image shows the layers of AUFS:



This is similar in context to a shipping container, which stays sealed until delivery but can be loaded, unloaded, stacked, and transported in between.

The visible filesystem of the processes in the container is based on AUFS (although you can configure the container to run with a different filesystem too). AUFS is a layered filesystem. These layers are all read-only and the merger of these layers is what is visible to the processes. However, if a process makes a change in the filesystem, a new layer is created, which represents the difference between the original state and the new state. When you create an image out of this container, the layers are preserved. Thus, it is possible to build new images out of existing images, creating a very convenient hierarchical model of images.

The docker daemon

The `docker` daemon is the process that manages containers. It is easy to get this confused with the Docker client because the same binary is used to run both the processes. The `docker` daemon, though, needs the `root` privileges, whereas the client doesn't.

Unfortunately, since the `docker` daemon runs with root privileges, it also introduces an attack vector. Read `https://docs.Docker.com/articles/security/` for more details.

Docker client

The Docker client is what interacts with the `docker` daemon to start or manage containers. Docker uses a RESTful API to communicate between the client and the daemon.

REST is an architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements within a distributed hypermedia system. In plain words, a RESTful service works over standard HTTP methods such as the `GET`, `POST`, `PUT`, and `DELETE` methods.

Dockerfile

A Dockerfile is a file written in a **Domain Specific Language** (**DSL**) that contains instructions on setting up a Docker image. Think of it as a Makefile equivalent of Docker.

Let's take a look at the following example:

```
FROM ubuntu:latest
MAINTAINER Scott P. Gallagher <email@somewhere.com>
RUN apt-get update && apt-get install -y apache2
ADD 000-default.conf /etc/apache2/sites-available/
RUN chown root:root /etc/apache2/sites-available/000-default.conf
EXPOSE 80
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

The FROM and MAINTAINER fields have information on what image is to be used and who is the maintainer of that image. The RUN instruction can be used to fetch and install packages along with other various commands. The ADD instruction allows you to add files or folders to the Docker image. The EXPOSE instruction allows you to expose ports from the image to the outside world. Lastly, the CMD instruction executes the said command and keeps the container alive.

Docker repository

A Docker repository is a namespace that is used for storing a Docker image. For instance, if your app is named helloworld and your username or namespace for the Registry is thedockerbook then, in the Docker Repository, where this image would be stored in the Docker Registry would be named thedockerbook/helloworld.

The base images are stored in the Docker Repository. The base images are the fountainheads for realizing the bigger and better images with the help of a careful addition of new modules. The child images are the ones that have their own parent images. The base image does not have any parent image. The images sitting on a base image are named as parent images because the parent images bear the child images.

Docker commands

Now let's get our hands dirty on the Docker CLI. We will look at the most common commands and their use cases. The Docker commands are modeled after Linux and Git, so if you have used either of these, you will find yourself at home with Docker.

Only the most commonly used options are mentioned here. For the complete reference, you can check out the official documentation at https://docs.docker.com/reference/commandline/cli/.

The daemon command

If you have installed the docker daemon through standard repositories, the command to start the docker daemon would have been added to the init script to automatically start as a service on startup. Otherwise, you will have to first run the docker daemon yourself for the client commands to work.

Now, while starting the daemon, you can run it with arguments that control the **Domain Name System** (**DNS**) configurations, storage drivers, and execution drivers for the containers:

```
$ export DOCKER_HOST="tcp://0.0.0.0:2375"
$ Docker -d -D -e lxc -s btrfs --dns 8.8.8.8 --dns-search example.com
```

You'll need these only if you want to start the daemon yourself. Otherwise, you can start the `docker` daemon with `$ sudo service Docker start`. For OS X and Windows, you need to run the commands mentioned in *Chapter 1, Installing Docker*.

The following table describes the various flags:

Flag	**Explanation**
`-d`	This runs Docker as a daemon.
`-D`	This runs Docker in debug mode.
`-e [option]`	This is the execution driver to be used. The default execution driver is native, which uses `libcontainer`.
`-s [option]`	This forces Docker to use a different storage driver. The default value is "", for which Docker uses AUFS.
`--dns [option(s)]`	This sets the DNS server (or servers) for all Docker containers.
`--dns-search [option(s)]`	This sets the DNS search domain (or domains) for all Docker containers.
`-H [option(s)]`	This is the socket (or sockets) to bind to. It can be one or more of `tcp://host:port, unix:///path/to/socket, fd://* or fd://socketfd`.

If multiple `docker` daemons are being simultaneously run, the client honors the value set by the `DOCKER_HOST` parameter. You can also make it connect to a specific daemon with the `-H` flag.

Consider this command:

```
$ docker -H tcp://0.0.0.0:2375 run -it ubuntu /bin/bash
```

The preceding command is the same as the following command:

```
$ DOCKER_HOST="tcp://0.0.0.0:2375" docker run -it ubuntu /bin/bash
```

The version command

The `version` command prints out the version information:

```
$ docker -v
Docker version 1.1.1, build bd609d2
```

The info command

The `info` command prints the details of the `docker` daemon configuration such as the execution driver, the storage driver being used, and so on:

```
$ docker info # Running it in boot2docker on OS X
Containers: 0
Images: 0
Storage Driver: aufs
 Root Dir: /mnt/sda1/var/lib/docker/aufs
 Dirs: 0
Execution Driver: native-0.2
Kernel Version: 3.15.3-tinycore64
Debug mode (server): true
Debug mode (client): false
Fds: 10
Goroutines: 10
EventsListeners: 0
Init Path: /usr/local/bin/docker
Sockets: [unix:///var/run/docker.sock tcp://0.0.0.0:2375]
```

The run command

The run command is the command that we will be using most frequently. It is used to run Docker containers:

```
$ docker run [options] IMAGE [command] [args]
```

Flags	Explanation
`-a, --attach=[]`	Attach to the `stdin`, `stdout`, or `stderr` files (standard input, output, and error files.).
`-d, --detach`	This runs the container in the background.
`-i, --interactive`	This runs the container in interactive mode (keeps the `stdin` file open).
`-t, --tty`	This allocates a pseudo `tty` flag (which is required if you want to attach to the container's terminal).
`-p, --publish=[]`	This publishes a container's port to the host (`ip:hostport:containerport`).
`--rm`	This automatically removes the container when exited (it cannot be used with the `-d` flag).

Flags	Explanation
`--privileged`	This gives additional privileges to this container.
`-v, --volume=[]`	This bind mounts a volume (from host => `/host:/container`; from docker => `/container`).
`--volumes-from=[]`	This mounts volumes from specified containers.
`-w, --workdir=""`	This is the working directory inside the container.
`--name=""`	This assigns a name to the container.
`-h, --hostname=""`	This assigns a hostname to the container.
`-u, --user=""`	This is the username or UID the container should run on.
`-e, --env=[]`	This sets the environment variables.
`--env-file=[]`	This reads environment variables from a new line-delimited file.
`--dns=[]`	This sets custom DNS servers.
`--dns-search=[]`	This sets custom DNS search domains.
`--link=[]`	This adds link to another container (`name:alias`).
`-c, --cpu-shares=0`	This is the relative CPU share for this container.
`--cpuset=""`	These are the CPUs in which to allow execution; starts with 0. (For example, 0 to 3).
`-m, --memory=""`	This is the memory limit for this container (`<number><b|k|m|g>`).
`--restart=""`	(v1.2+) This specifies a restart policy in case the container crashes.
`--cap-add=""`	(v1.2+) This grants a capability to a container (refer to *Chapter 4, Security Best Practices*).
`--cap-drop=""`	(v1.2+) This blacklists a capability to a container (refer to *Chapter 4, Security Best Practices*).
`--device=""`	(v1.2+) This mounts a device on a container.

While running a container, it is important to keep in mind that the container's lifetime is associated with the lifetime of the command you run when you start the container. Now try to run this:

```
$ docker run -dt ubuntu ps
b1d037dfcff6b076bde360070d3af0d019269e44929df61c93dfcdfaf29492c9
$ docker attach b1d037
2014/07/16 16:01:29 You cannot attach to a stopped container, start
it first
```

What happened here? When we ran the simple command, `ps`, the container ran the command and exited. Therefore, we got an error.

The `attach` command attaches the standard input and output to a running container.

Another important piece of information here is that you don't need to use the whole 64-character ID for all the commands that require the container ID. The first couple of characters are sufficient. With the same example as shown in the following code:

```
$ docker attach b1d03
2014/07/16 16:09:39 You cannot attach to a stopped container, start
it first
$ docker attach b1d0
2014/07/16 16:09:40 You cannot attach to a stopped container, start
it first
$ docker attach b1d
2014/07/16 16:09:42 You cannot attach to a stopped container, start
it first
$ docker attach b1
2014/07/16 16:09:44 You cannot attach to a stopped container, start
it first
$ docker attach b

2014/07/16 16:09:45 Error: No such container: b
```

A more convenient method though would be to name your containers yourself:

```
$ docker run -dit --name OD-name-example ubuntu /bin/bash
1b21af96c38836df8a809049fb3a040db571cc0cef000a54ebce978c1b5567ea
$ docker attach OD-name-example
root@1b21af96c388:/#
```

The `-i` flag is necessary to have any kind of interaction in the container, and the `-t` flag is necessary to create a pseudo-terminal.

The previous example also made us aware of the fact that even after we exit a container, it is still in a `stopped` state. That is, we will be able to start the container again, with its filesystem layer preserved. You can see this by running the following command:

```
$ docker ps -a
CONTAINER ID IMAGE         COMMAND CREATED    STATUS    NAMES
eb424f5a9d3f ubuntu:latest ps      1 hour ago Exited OD-name-example
```

While this can be convenient, you may pretty soon have your host's disk space drying up as more and more containers are saved. So, if you are going to run a disposable container, you can run it with the `--rm` flag, which will remove the container when the process exits:

```
$ docker run --rm -it --name OD-rm-example ubuntu /bin/bash
root@0fc99b2e35fb:/# exit
exit
$ docker ps -a
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

Running a server

Now, for our next example, we'll try running a web server. This example is chosen because the most common practical use case of Docker containers is the shipping of web applications:

```
$ docker run -it --name OD-pythonserver-1 --rm python:2.7 \
python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000
```

Now we know the problem; we have a server running in a container, but since the container's IP is assigned by Docker dynamically, it makes things difficult. However, we can bind the container's ports to the host's ports and Docker will take care of forwarding the networking traffic. Now let's try this command again with the `-p` flag:

```
$ docker run -p 0.0.0.0:8000:8000 -it --rm --name OD-pythonserver-2 \
python:2.7 python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000 ...
172.17.42.1 - - [18/Jul/2014 14:25:46] "GET / HTTP/1.1" 200 -
```

Now open your browser and go to `http://localhost:8000`. Voilà!

If you are an OS X user and you realize that you are not able to access `http://localhost:8000`, it is because VirtualBox hasn't been configured to respond to **Network Address Translation** (**NAT**) requests to the boot2Docker VM. Adding the following function to your aliases file (`bash_profile` or `.bashrc`) will save a lot of trouble:

```
natboot2docker () {
   VBoxManage controlvm boot2docker-vm natpf1 \
   "$1,tcp,127.0.0.1,$2,,$3";
}


removeDockerNat() {
    VBoxManage modifyvm boot2docker-vm \
    --natpf1 delete $1;
}
```

After this, you should be able to use the `$ natboot2docker mypythonserver 8000 8000` command to be able to access the Python server. But remember to run the `$ removeDockerDockerNat mypythonserver` command when you are done. Otherwise, when you run the boot2Docker VM next time, you will be faced with a bug that won't allow you to get the IP address or the `ssh` script into it:

```
$ boot2docker ssh
ssh_exchange_identification: Connection closed by remote host
2014/07/19 11:55:09 exit status 255
```

Your browser now shows the `/root` path of the container. What if you wanted to serve your host's directories? Let's try mounting a device:

```
root@eb53f7ec79fd:/# mount -t tmpfs /dev/random /mnt
mount: permission denied
```

As you can see, the `mount` command doesn't work. In fact, most kernel capabilities that are potentially dangerous are dropped, unless you include the `--privileged` flag.

However, you should never use this flag unless you know what you are doing. Docker provides a much easier way to bind mount host volumes and bind mount host volumes with the `-v` and `–volumes` options. Let's try this example again in the directory we are currently in:

```
$ docker run -v $(pwd):$(pwd) -p 0.0.0.0:8000:8000 -it -rm \
--name OD-pythonserver-3 python:2.7 python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.2.2 - - [18/Jul/2014 14:40:35] "GET / HTTP/1.1" 200 -
```

You have now bound the directory you are running the commands from to the container. However, when you access the container, you still get the directory listing of the root of the container. To serve the directory that has been bound to the container, let's set it as the working directory of the container (the directory the containerized process runs in) using the `-w` flag:

```
$ docker run -v $(pwd):$(pwd) -w $(pwd) -p 0.0.0.0:8000:8000 -it \ --name
OD-pythonserver-4 python:2.7 python -m SimpleHTTPServer 8000;
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.2.2 - - [18/Jul/2014 14:51:35] "GET / HTTP/1.1" 200 -
```

Boot2Docker users will not be able to utilize this yet, unless you use guest additions and set up shared folders, the guide to which can be found at `https://medium.com/boot2docker-lightweight-linux-for-docker/boot2docker-together-with-virtualbox-guest-additions-da1e3ab2465c`. Though this solution works, it is a hack and is not recommended. Meanwhile, the Docker community is actively trying to find a solution (check out issue `#64` in the boot2Docker GitHub repository and `#4023` in the Docker repository).

Now `http://localhost:8000` will serve the directory you are currently running in, but from a Docker container. Take care though, because any changes you make are written into the host's filesystem as well.

Since v1.1.1, you can bind mount the root of the host to a container using `$ docker run -v /:/my_host:ro ubuntu ls /my_host`, but mounting on the / path of the container is forbidden.

The volume can be optionally suffixed with the `:ro` or `:rw` commands to mount the volumes in read-only or read-write mode, respectively. By default, the volumes are mounted in the same mode (read-write or read-only) as they are in the host.

This option is mostly used to mount static assets and to write logs.

But what if I want to mount an external device?

Before v1.2, you had to mount the device in the host and bind mount using the `-v` flag in a privileged container, but v1.2 has added a `--device` flag that you can use to mount a device without needing to use the `--privileged` flag.

For example, to use the webcam in your container, run this command:

```
$ docker run --device=/dev/video0:/dev/video0
```

Docker v1.2 also added a `--restart` flag to specify a restart policy for containers. Currently, there are three restart policies:

- `no`: Do not restart the container if it dies (default).
- `on-failure`: Restart the container if it exits with a non-zero exit code. It can also accept an optional maximum restart count (for example, `on-failure:5`).
- `always`: Always restart the container no matter what exit code is returned.

The following is an example to restart endlessly:

```
$ docker run --restart=always code.it
```

The next line is used to try five times before giving up:

```
$ docker run --restart=on-failure:5 code.it
```

The search command

The `search` command allows us to search for Docker images in the public registry. Let's search for all images related to Python:

```
$ docker search python | less
```

The pull command

The `pull` command is used to pull images or repositories from a registry. By default, it pulls them from the public Docker registry, but if you are running your own registry, you can pull them from it too:

```
$ docker pull python # pulls repository from Docker Hub
$ docker pull python:2.7 # pulls the image tagged 2.7
$ docker pull <path_to_registry>/<image_or_repository>
```

The start command

We saw when we discussed `docker run` that the container state is preserved on exit unless it is explicitly removed. The `docker start` command starts a stopped container:

```
$ docker start [-i] [-a] <container(s)>
```

The stop command

The `stop` command stops a running container by sending the `SIGTERM` signal and then the `SIGKILL` signal after a grace period:

> `SIGTERM` and `SIGKILL` are Unix signals. A signal is a form of interprocess communication used in Unix, Unix-like, and other POSIX-compliant operating systems. `SIGTERM` signals the process to terminate. The `SIGKILL` signal is used to forcibly kill a process.

```
docker run -dit --name OD-stop-example ubuntu /bin/bash
$ docker ps
CONTAINER ID IMAGE         COMMAND   CREATED  STATUS    NAMES
679ece6f2a11 ubuntu:latest /bin/bash 5h ago   Up 3s   OD-stop-example
$ docker stop OD-stop-example
OD-stop-example
$ docker ps
CONTAINER ID IMAGE         COMMAND   CREATED  STATUS    NAMES
```

You can also specify the `-t` flag or `--time` flag, which allows you to set the wait time.

The restart command

The `restart` command restarts a running container:

```
$ docker run -dit --name OD-restart-example ubuntu /bin/bash
$ sleep 15s # Suspends execution for 15 seconds
$ docker ps
CONTAINER ID IMAGE         COMMAND   STATUS    NAMES
cc5d0ae0b599 ubuntu:latest /bin/bash Up 20s    OD-restart-example

$ docker restart OD-restart-example
$ docker ps
CONTAINER ID IMAGE         COMMAND   STATUS    NAMES
cc5d0ae0b599 ubuntu:latest /bin/bash Up 2s    OD-restart-example
```

If you observe the status, you will notice that the container was rebooted.

The rm command

The `rm` command removes Docker containers:

```
$ Docker ps -a # Lists containers including stopped ones
CONTAINER ID  IMAGE  COMMAND   CREATED  STATUS NAMES
cc5d0ae0b599  ubuntu /bin/bash 6h ago   Exited OD-restart-example
679ece6f2a11  ubuntu /bin/bash 7h ago   Exited OD-stop-example
e3c4b6b39cff  ubuntu /bin/bash 9h ago   Exited OD-name-example
```

We seem to be having a lot of containers left over after our adventures. Let's remove one of them:

```
$ dockerDocker rm OD-restart-example
cc5d0ae0b599
```

We can also combine two Docker commands. Let's combine the `docker ps -a -q` command, which prints the ID parameters of the containers in the `docker ps -a`, and `docker rm` commands, to remove all containers in one go:

```
$ docker rm $(docker ps -a -q)
679ece6f2a11
e3c4b6b39cff
$ docker ps -a
CONTAINER ID    IMAGE    COMMAND     CREATED    STATUS      NAMES
```

This evaluates the `docker ps -a -q` command first, and the output is used by the `docker rm` command.

The ps command

The `ps` command is used to list containers. It is used in the following way:

```
$ docker ps [option(s)]
```

Flag	Explanation
`-a, --all`	This shows all containers, including stopped ones.
`-q, --quiet`	This shows only container ID parameters.
`-s, --size`	This prints the sizes of the containers.
`-l, --latest`	This shows only the latest container (including stopped containers).
`-n=""`	This shows the last *n* containers (including stopped containers). Its default value is -1.

Flag	Explanation
`--before=""`	This shows the containers created before the specified ID or name. It includes stopped containers.
`--after=""`	This shows the containers created after the specified ID or name. It includes stopped containers.

The `docker ps` command will show only running containers by default. To see all containers, run the `docker ps -a` command. To see only container ID parameters, run it with the `-q` flag.

The logs command

The `logs` command shows the logs of the container:

```
Let us look at the logs of the python server we have been running
$ docker logs OD-pythonserver-4
Serving HTTP on 0.0.0.0 port 8000 ...
10.0.2.2 - - [18/Jul/2014 15:06:39] "GET / HTTP/1.1" 200 -
^CTraceback (most recent call last):
  File ...
  ...
KeyboardInterrupt
```

You can also provide a `--tail` argument to follow the output as the container is running.

The inspect command

The `inspect` command allows you to get the details of a container or an image. It returns those details as a JSON array:

```
$ Docker inspect ubuntu # Running on an image
[{
    "Architecture": "amd64",
    "Author": "",
    "Comment": "",
     .......
     .......
     .......
    "DockerVersion": "0.10.0",
    "Id":
"e54ca5efa2e962582a223ca9810f7f1b62ea9b5c3975d14a5da79d3bf6020f37",
    "Os": "linux",
    "Parent":
"6c37f792ddacad573016e6aea7fc9fb377127b4767ce6104c9f869314a12041e",
    "Size": 178365
}]
```

Similarly, for a container we run the following command:

```
$ Docker inspect OD-pythonserver-4 # Running on a container
[{
    "Args": [
        "-m",
        "SimpleHTTPServer",
        "8000"
    ],
    ......
    ......
    "Name": "/OD-pythonserver-4",
    "NetworkSettings": {
        "Bridge": "Docker0",
        "Gateway": "172.17.42.1",
        "IPAddress": "172.17.0.11",
        "IPPrefixLen": 16,
        "PortMapping": null,
        "Ports": {
            "8000/tcp": [
                {
                    "HostIp": "0.0.0.0",
                    "HostPort": "8000"
                }
            ]
        }
    },
    ......
    ......
    "Volumes": {
        "/home/Docker": "/home/Docker"
    },
    "VolumesRW": {
        "/home/Docker": true
    }
}]
```

Docker inspect provides all of the low-level information about a container or image. In the preceding example, find out the IP address of the container and the exposed port and make a request to the `IP:port`. You will see that you are directly accessing the server running in the container.

However, manually looking through the entire JSON array is not optimal. So the `inspect` command provides a flag, `-f` (or the `--format` flag), which allows you to specify exactly what you want using `Go` templates. For example, if you just want to get the container's IP address, run the following command:

```
$ docker inspect -f  '{{.NetworkSettings.IPAddress}}' \
OD-pythonserver-4;
172.17.0.11
```

The `{{.NetworkSettings.IPAddress}}` is a `Go` template that was executed over the JSON result. `Go` templates are very powerful, and some of the things that you can do with them have been listed at `http://golang.org/pkg/text/template/`.

The top command

The `top` command shows the running processes in a container and their statistics, mimicking the Unix `top` command.

Let's download and run the `ghost` blogging platform and check out what processes are running in it:

```
$ docker run -d -p 4000:2368 --name OD-ghost dockerfile/ghost
ece88c79b0793b0a49e3d23e2b0b8e75d89c519e5987172951ea8d30d96a2936


$ docker top OD-ghost-1
PID                 USER                COMMAND
1162                root                bash /ghost-start
1180                root                npm
1186                root                sh -c node index
1187                root                node index
```

Yes! We just set up our very own `ghost` blog, with just one command. This brings forth another subtle advantage and shows something that could be a future trend. Every tool that exposes its services through a TCP port can now be containerized and run in its own sandboxed world. All you need to do is expose its port and bind it to your host port. You don't need to worry about installations, dependencies, incompatibilities, and so on, and the uninstallation will be clean because all you need to do is stop all the containers and remove the image.

> Ghost is an open source publishing platform that is beautifully designed, easy to use, and free for everyone. It is coded in Node.js, a server-side JavaScript execution engine.

The attach command

The `attach` command attaches to a running container.

Let's start a container with Node.js, running the node interactive shell as a daemon, and later attach to it.

> Node.js is an event-driven, asynchronous I/O web framework that runs applications written in JavaScript on Google's V8 runtime environment.

The container with Node.js is as follows:

```
$ docker run -dit --name OD-nodejs shykes/nodejs node
8e0da647200efe33a9dd53d45ea38e3af3892b04aa8b7a6e167b3c093e522754

$ docker attach OD-nodejs
console.log('Docker rocks!');Docker rocks!
```

The kill command

The `kill` command kills a container and sends the `SIGTERM` signal to the process running in the container:

```
Let us kill the container running the ghost blog.
$ docker kill OD-ghost-1
OD-ghost-1

$ docker attach OD-ghost-1 # Verification
```

```
2014/07/19 18:12:51 You cannot attach to a stopped container, start
it first
```

The cp command

The `cp` command copies a file or folder from a container's filesystem to the host path. Paths are relative to the root of the filesystem.

It's time to have some fun. First, let's run an Ubuntu container with the `/bin/bash` command:

```
$ docker run -it –name OD-cp-bell ubuntu /bin/bash
```

Now, inside the container, let's create a file with a special name:

```
# touch $(echo -e '\007')
```

The `\007` character is an ASCII `BEL` character that rings the system bell when printed on a terminal. You might have already guessed what we're about to do. So let's open a new terminal and execute the following command to copy this newly created file to the host:

```
$ docker cp OD-cp-bell:/$(echo -e '\007') $(pwd)
```

> For the `docker cp` command to work, both the container path and the host path must be complete, so do not use shortcuts such as `.`, `,`, `*`, and so on.

So we created an empty file whose filename is the `BEL` character, in a container. Then we copied the file to the current directory in the host container. Just one last step is remaining. In the host tab where you executed the `docker cp` command, run the following command:

```
$ echo *
```

You will hear the system bell ring! We could have copied any file or directory from the container to the host. But it doesn't hurt to have some fun!

> If you found this interesting, you might like to read `http://www.dwheeler.com/essays/fixing-unix-linux-filenames.html`. This is a great essay that discusses the edge cases in filenames, which can cause simple to complicated issues in a program.

The port command

The `port` command looks up the public-facing port that is bound to an exposed port in the container:

```
$ docker port CONTAINER PRIVATE_PORT
$ docker port OD-ghost 2368

4000
```

Ghost runs a server at the `2368` port that allows you to write and publish a blog post. We bound a host port to the `OD-ghost` container's port `2368` in the example for the `top` command.

Running your own project

By now, we are considerably familiar with the basic Docker commands. Let's up the ante. For the next couple of commands, I am going to use one of my side projects. Feel free to use a project of your own.

Let's start by listing out our requirements to determine the arguments we must pass to the `docker run` command.

Our application is going to run on Node.js, so we will choose the well-maintained `dockerfile/nodejs` image to start our base container:

- We know that our application is going to bind to port `8000`, so we will expose the port to `8000` of the host.
- We need to give a descriptive name to the container so that we can reference it in future commands. In this case, let's choose the name of the application:

  ```
  $ docker run -it --name code.it dockerfile/nodejs /bin/bash
  [ root@3b0d5a04cdcd:/data ]$ cd /home
  [ root@3b0d5a04cdcd:/home ]$
  ```

Once you have started your container, you need to check whether the dependencies for your application are already available. In our case, we only need Git (apart from Node.js), which is already installed in the `dockerfile/nodejs` image.

Now that our container is ready to run our application, all that is remaining is for us to fetch the source code and do the necessary setup to run the application:

```
$ git clone https://github.com/shrikrishnaholla/code.it.git
$ cd code.it && git submodule update --init --recursive
```

This downloads the source code for a plugin used in the application.

Then run the following command:

```
$ npm install
```

Now all the node modules required to run the application are installed.

Next, run this command:

```
$ node app.js
```

Now you can go to `localhost:8000` to use the application.

The diff command

The `diff` command shows the difference between the container and the image it is based on. In this example, we are running a container with `code.it`. In a separate tab, run this command:

```
$ docker diff code.it
C /home
A /home/code.it
...
```

The commit command

The `commit` command creates a new image with the filesystem of the container. Just as with Git's `commit` command, you can set a commit message that describes the image:

```
$ docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
```

Flag	Explanation
`-p, --pause`	This pause the container during commit (availabe from v1.1.1+ onwards).
`-m, --message=""`	This is a commit message. It can be a description of what the image does.
`-a, --author=""`	This displays the author details.

For example, let's use this command to commit the container we have set up:

```
$ docker commit -m "Code.it - A browser based text editor and
interpreter" -a "Shrikrishna Holla <s**a@gmail.com>" code.it
shrikrishna/code.it:v1
```

Replace the author details and the username portion of the image name in this example if you are copying these examples.

The output will be a lengthy image ID. If you look at the command closely, we have named the image `shrikrishna/code.it:v1`. This is a convention. The first part of an image/repository's name (before the forward slash) is the Docker Hub username of the author. The second part is the intended application or image name. The third part is a tag (usually a version description) separated from the second part by a colon.

`Docker Hub` is a public registry maintained by Docker, Inc. It hosts public Docker images and provides services to help you build and manage your Docker environment. More details about it can be found at `https://hub.docker.com`.

A collection of images tagged with different versions is a repository. The image you create by running the `docker commit` command will be a local one, which means that you will be able to run containers from it but it won't be available publicly. To make it public or to push to your private Docker registry, use the `docker push` command.

The images command

The `images` command lists all the images in the system:

```
$ docker images [OPTIONS] [NAME]
```

Flag	Explanation
`-a, --all`	This shows all images, including intermediate layers.
`-f, --filter=[]`	This provides filter values.
`--no-trunc`	This doesn't truncate output (shows complete ID).
`-q, --quiet`	This shows only the image IDs.

Now let's look at a few examples of the usage of the `image` command:

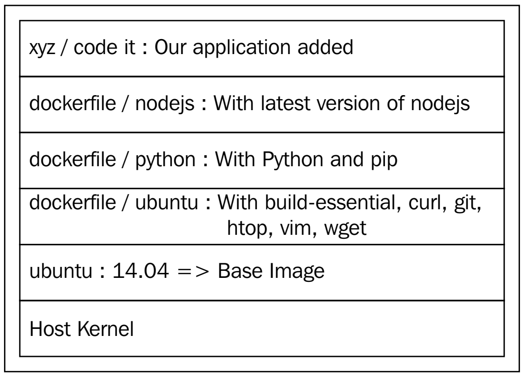
```
$ docker images
REPOSITORY           TAG   IMAGE ID       CREATED    VIRTUAL SIZE
shrikrishna/code.it  v1    a7cb6737a2f6   6m ago     704.4 MB
```

This lists all top-level images, their repository and tags, and their virtual size.

Docker images are nothing but a stack of read-only filesystem layers. A union filesystem, such as AUFS, then merges these layers and they appear to be one filesystem.

In Docker-speak, a read-only layer is an image. It never changes. When running a container, the processes think that the entire filesystem is read-write. But the changes go only at the topmost writeable layer, which is created when a container is started. The read-only layers of the image remain unchanged. When you commit a container, it freezes the top layer (the underlying layers are already frozen) and turns it into an image. Now, when a container is started this image, all the layers of the image (including the previously writeable layer) are read-only. All the changes are now made to a new writeable layer on top of all the underlying layers. However, because of how union filesystems (such as AUFS) work, the processes believe that the filesystem is read-write.

A rough schematic of the layers involved in our `code.it` example is as follows:


xyz / code it : Our application added
dockerfile / nodejs : With latest version of nodejs
dockerfile / python : With Python and pip
dockerfile / ubuntu : With build-essential, curl, git, htop, vim, wget
ubuntu : 14.04 => Base Image
Host Kernel

At this point, it might be wise to think just how much effort is to be made by the union filesystems to merge all of these layers and provide a consistent performance. After some point, things inevitably break. AUFS, for instance, has a 42-layer limit. When the number of layers goes beyond this, it just doesn't allow the creation of any more layers and the build fails. Read `https://github.com/docker/docker/issues/1171` for more information on this issue.

The following command lists the most recently created images:

```
$ docker images | head
```

The `-f` flag can be given arguments of the `key=value` type. It is frequently used to get the list of dangling images:

```
$ docker images -f "dangling=true"
```

This will display untagged images, that is, images that have been committed or built without a tag.

The rmi command

The `rmi` command removes images. Removing an image also removes all the underlying images that it depends on and were downloaded when it was pulled:

```
$ docker rmi [OPTION] {IMAGE(s)]
```

Flag	Explanation
`-f, --force`	This forcibly removes the image (or images).
`--no-prune`	This command does not delete untagged parents.

This command removes one of the images from your machine:

```
$ docker rmi test
```

The save command

The `save` command saves an image or repository in a tarball and this streams to the `stdout` file, preserving the parent layers and metadata about the image:

```
$ docker save -o codeit.tar code.it
```

The `-o` flag allows us to specify a file instead of streaming to the `stdout` file. It is used to create a backup that can then be used with the `docker load` command.

The load command

The `load` command loads an image from a tarball, restoring the filesystem layers and the metadata associated with the image:

```
$ docker load -i codeit.tar
```

The `-i` flag allows us to specify a file instead of trying to get a stream from the `stdin` file.

The export command

The `export` command saves the filesystem of a container as a tarball and streams to the `stdout` file. It flattens filesystem layers. In other words, it merges all the filesystem layers. All of the metadata of the image history is lost in this process:

```
$ sudo Docker export red_panda > latest.tar
```

Here, `red_panda` is the name of one of my containers.

The import command

The `import` command creates an empty filesystem image and imports the contents of the tarball to it. You have the option of tagging it the image:

```
$ docker import URL|- [REPOSITORY[:TAG]]
```

URLs must start with `http`.

```
$ docker import http://example.com/test.tar.gz # Sample url
```

If you would like to import from a local directory or archive, you can use the - parameter to take the data from the `stdin` file:

```
$ cat sample.tgz | docker import – testimage:imported
```

The tag command

You can add a `tag` command to an image. It helps identify a specific version of an image.

For example, the `python` image name represents `python:latest`, the latest version of Python available, which can change from time to time. But whenever it is updated, the older versions are tagged with the respective Python versions. So the `python:2.7` command will have Python 2.7 installed. Thus, the `tag` command can be used to represent versions of the images, or for any other purposes that need identification of the different versions of the image:

```
$ docker tag IMAGE [REGISTRYHOST/][USERNAME/]NAME[:TAG]
```

The `REGISTRYHOST` command is only needed if you are using a private registry of your own. The same image can have multiple tags:

```
$ docker tag shrikrishna/code.it:v1 shrikrishna/code.it:latest
```

Whenever you are tagging an image, follow the `username/repository:tag` convention.

Now, running the `docker images` command again will show that the same image has been tagged with both the `v1` and `latest` commands:

```
$ docker images
REPOSITORY            TAG     IMAGE ID      CREATED     VIRTUAL SIZE
shrikrishna/code.it   v1      a7cb6737a2f6  8 days ago  704.4 MB
shrikrishna/code.it   latest  a7cb6737a2f6  8 days ago  704.4 MB
```

The login command

The `login` command is used to register or log in to a Docker registry server. If no server is specified, `https://index.docker.io/v1/` is the default:

```
$ Docker login [OPTIONS] [SERVER]
```

Flag	Explanation
`-e, --email=""`	Email
`-p, --password=""`	Password
`-u, --username=""`	Username

If the flags haven't been provided, the server will prompt you to provide the details. After the first login, the details will be stored in the `$HOME/.dockercfg` path.

The push command

The `push` command is used to push an image to the public image registry or a private Docker registry:

```
$ docker push NAME[:TAG]
```

The history command

The `history` command shows the history of the image:

```
$ docker history shykes/nodejs
IMAGE         CREATED        CREATED BY                          SIZE
6592508b0790  15 months ago  /bin/sh -c wget http://nodejs.  15.07 MB
0a2ff988ae20  15 months ago  /bin/sh -c apt-get install ...  25.49 MB
43c5d81f45de  15 months ago  /bin/sh -c apt-get update       96.48 MB
b750fe79269d  16 months ago  /bin/bash                       77 B
27cf78414709  16 months ago                                  175.3 MB
```

The events command

Once started, the `events` command prints all the events that are handled by the `docker` daemon, in real time:

```
$ docker events [OPTIONS]
```

Flag	Explanation
`--since=""`	This shows all events created since timestamp (in Unix).
`--until=""`	This stream events until timestamp.

For example the `events` command is used as follows:

```
$ docker events
```

Now, in a different tab, run this command:

```
$ docker start code.it
```

Then run the following command:

```
$ docker stop code.it
```

Now go back to the tab running Docker events and see the output. It will be along these lines:

```
[2014-07-21 21:31:50 +0530 IST]
c7f2485863b2c7d0071477e6cb8c8301021ef9036afd4620702a0de08a4b3f7b: (from
dockerfile/nodejs:latest) start


[2014-07-21 21:31:57 +0530 IST]
c7f2485863b2c7d0071477e6cb8c8301021ef9036afd4620702a0de08a4b3f7b: (from
dockerfile/nodejs:latest) stop


[2014-07-21 21:31:57 +0530 IST]
c7f2485863b2c7d0071477e6cb8c8301021ef9036afd4620702a0de08a4b3f7b: (from
dockerfile/nodejs:latest) die
```

You can use flags such as `--since` and `--until` to get the event logs of specific timeframes.

The wait command

The `wait` command blocks until a container stops, then prints its exit code:

```
$ docker wait CONTAINER(s)
```

The build command

The build command builds an image from the source files at a specified path:

```
$ Docker build [OPTIONS] PATH | URL | -
```

Flag	Explanation
`-t, --tag=""`	This is the repository name (and an optional tag) to be applied to the resulting image in case of success.
`-q, --quiet`	This suppresses the output, which by default is verbose.
`--rm=true`	This removes intermediate containers after a successful build.
`--force-rm`	This always removes intermediate containers, even after unsuccessful builds.
`--no-cache`	This command does not use the cache while building the image.

This command uses a Dockerfile and a context to build a Docker image.

A Dockerfile is like a Makefile. It contains instructions on the various configurations and commands that need to be run in order to create an image. We will look at writing Dockerfiles in the next section.

[It would be a good idea to read the section about Dockerfiles first and then come back here to get a better understanding of this command and how it works.]

The files at the `PATH` or `URL` paths are called **context** of the build. The context is used to refer to the files or folders in the Dockerfile, for instance in the `ADD` instruction (and that is the reason an instruction such as `ADD ../file.txt` won't work. It's not in the context!).

When a GitHub URL or a URL with the `git://` protocol is given, the repository is used as the context. The repository and its submodules are recursively cloned in your local machine, and then uploaded to the `docker` daemon as the context. This allows you to have Dockerfiles in your private Git repositories, which you can access from your local user credentials or from the **Virtual Private Network** (**VPN**).

Uploading to Docker daemon

Remember that Docker engine has both the `docker` daemon and the Docker client. The commands that you give as a user are through the Docker client, which then talks to the `docker` daemon (either through a TCP or a Unix socket), which does the necessary work. The `docker` daemon and Docker host can be in different hosts (which is the premise with which boot2Docker works), with the `DOCKER_HOST` environment variable set to the location of the remote `docker` daemon.

When you give a context to the `docker build` command, all the files in the local directory get tared and are sent to the `docker` daemon. The `PATH` variable specifies where to find the files for the context of the build in the `docker` daemon. So when you run `docker build .`, all the files in the current folder get uploaded, not just the ones listed to be added in the Dockerfile.

Since this can be a bit of a problem (as some systems such as Git and some IDEs such as Eclipse create hidden folders to store metadata), Docker provides a mechanism to ignore certain files or folders by creating a file called `.dockerignore` in the `PATH` variable with the necessary exclusion patterns. For an example, look up `https://github.com/docker/docker/blob/master/.dockerignore`.

If a plain URL is given or if the Dockerfile is streamed through the `stdin` file, then no context is set. In these cases, the `ADD` instruction works only if it refers to a remote URL.