



Community Experience Distilled

Learning Docker

Optimize the power of Docker to run your applications quickly and easily

Pethuru Raj
Vinod Singh

Jeeva S. Chelladurai

[PACKT]
PUBLISHING

Learning Docker

Optimize the power of Docker to run your applications quickly and easily

Pethuru Raj

Jeeva S. Chelladhurai

Vinod Singh



BIRMINGHAM - MUMBAI

Learning Docker

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: June 2015

Production reference: 1240615

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-793-7

www.packtpub.com

Credits

Authors

Pethuru Raj
Jeeva S. Chelladhurai
Vinod Singh

Copy Editors

Vikrant Phadke
Rashmi Sawant
Trishla Singh

Reviewers

Shashikant Bangera
Sergei Vizel
Baohua Yang

Project Coordinator

Nidhi Joshi

Proofreader

Safis Editing

Commissioning Editor

Sarah Crofton

Indexer

Hemangini Bari

Acquisition Editor

Larissa Pinto

Graphics

Sheetal Aute

Content Development Editor

Kirti Patil

Production Coordinator

Nitesh Thakur

Technical Editors

Dhiraj Chandanshive
Narsimha Pai

Cover Work

Nitesh Thakur

About the Authors

Pethuru Raj, PhD, works as a cloud architect at the IBM Global Cloud Center of Excellence (CoE) in Bangalore, India. He completed his CSIR-sponsored PhD degree at Anna University, Chennai, and continued his UGC-sponsored postdoctoral research at the Department of Computer Science and Automation of IISc, Bangalore. Thereafter, he was granted a couple of international research fellowships (JSPS and JST) to work as a research scientist for 3 years at two leading Japanese universities.

Pethuru has contributed to a number of high-quality technology books that are edited by internationally acclaimed professionals. In association with another IBMer, he has recently submitted the complete manuscript for a book called *Smart Cities: the Enabling Technologies and Tools*, to be published by the CRC Press in the USA in May 2015. He has collaborated with a few established authors to publish a book called *High-Performance Big Data Analytics*, which will be published by Springer-Verlag, UK, in 2015. He maintains an IT portal at <http://www.peterindia.net>, and his LinkedIn profile can be found at <https://www.linkedin.com/in/peterindia>.

Jeeva S. Chelladurai has been working as a technical project manager at the IBM Global Cloud Center of Excellence (CoE) in India for the last 8 years. He has more than 18 years of experience in the IT industry. In various capacities, he has technically managed and mentored diverse teams across the globe in envisaging and building pioneering telecommunication products. He specializes in cloud solution delivery, with a focus on data center optimization, software-defined environments (SDEs), and distributed application development, deployment, and delivery using the newest Docker technology. Jeeva is also a strong proponent of Agile methodologies, DevOps, and IT automation. He holds a master's degree in computer science from Manonmaniam Sundaranar University and a graduation certificate in project management from Boston University. He has been instrumental in crafting reusable assets for IBM solution architects and consultants in Docker-inspired containerization technology.

Vinod Singh is a lead architect for IBM's cloud computing offerings. He has more than 18 years of experience in the cloud computing, networking, and data communication domains. Currently, he works for IBM's cloud application services and partner marketplace offerings. Vinod has worked on architecting, deploying, and running IBM's PaaS offering (BlueMix) on the SoftLayer infrastructure cloud. He also provides consultancy and advisory services to clients across the globe on the adoption of cloud technologies. He is currently focusing on various applications and services on the IBM Marketplace/BlueMix/SoftLayer platform. He is a graduate engineer from the National Institute of Technology, Jaipur, and completed his master's degree at BITS, Pilani.

About the Reviewers

Shashikant Bangera is a DevOps architect with 16 years of IT experience. He has vast exposure to DevOps tools across the platform, with core expertise in open source. He has helped his customers adopt DevOps practice and implemented Enterprise DevOps for them and has also contributed to many open source platforms, such as DevOps Publication. He has designed an automated on-demand environment with a set of open source tools and also an environment booking tool, which is available on GitHub. His Twitter handle is @shzshi.

Sergei Vizel is a senior software engineer at Modera (modera.org). He is a full-stack web application developer with more than 10 years of impressive experience. He is a firm believer of the value and power of open source software and contributes to projects on GitHub. Sergei has published numerous pieces of open source code of his own. You can learn more about him and contact him on GitHub via <https://github.com/cravler>.

Baohua Yang is a research scientist on cloud-computing-related technologies at IBM. He is a contributor to many open source communities such as OpenStack, OpenvSwitch, Docker, and OpenDaylight. He is also a TPC member and a reviewer of a number of international conferences and journals.

Baohua's interests mainly include system and application architecture, performance optimization, and security issues in cloud networking and distributed systems, especially in emerging technologies such as cloud computing, SDN, and NFV. He has written many technical books and articles to introduce and analyze these techniques. He loves open source technologies and enjoys designing and implementing efficient systems with elegant architecture.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	vii
Chapter 1: Getting Started with Docker	1
An introduction to Docker	2
Docker on Linux	3
Differentiating between containerization and virtualization	4
The convergence of containerization and virtualization	5
Containerization technologies	6
Installing the Docker engine	7
Installing from the Ubuntu package repository	7
Installing the latest Docker using docker.io script	8
Understanding the Docker setup	9
Client server communication	11
Downloading the first Docker image	12
Running the first Docker container	12
Running a Docker container on Amazon Web Services	13
Troubleshooting	14
Summary	15
Chapter 2: Handling Docker Containers	17
Clarifying the Docker terms	18
Docker images and containers	18
A Docker layer	20
A Docker container	20
Docker Registry	20
Docker Repository	21

Working with Docker images	21
Docker Hub Registry	23
Searching Docker images	24
Working with an interactive container	26
Tracking changes inside containers	28
Controlling Docker containers	30
Housekeeping containers	34
Building images from containers	35
Launching a container as a daemon	37
Summary	38
Chapter 3: Building Images	39
Docker's integrated image building system	39
A quick overview of the Dockerfile's syntax	42
The Dockerfile build instructions	43
The FROM instruction	44
The MAINTAINER instruction	44
The COPY instruction	45
The ADD instruction	46
The ENV instruction	47
The USER instruction	47
The WORKDIR instruction	48
The VOLUME instruction	48
The EXPOSE instruction	49
The RUN instruction	49
The CMD instruction	51
The ENTRYPOINT instruction	53
The ONBUILD instruction	55
The .dockerignore file	55
A brief overview of the Docker image management	56
Best practices for writing Dockerfiles	58
Summary	58
Chapter 4: Publishing Images	59
Understanding the Docker Hub	60
Pushing images to the Docker Hub	63
Automating the building process for images	68
Private repositories on the Docker Hub	71
Organizations and teams on the Docker Hub	72
The REST APIs for the Docker Hub	73
Summary	75

Chapter 5: Running Your Private Docker Infrastructure	77
The Docker registry and index	78
Docker registry use cases	78
Run your own index and registry	80
Step 1 – Deployment of the index components and the registry from GitHub	80
Step 2 – Configuration of nginx with the Docker registry	82
Step 3 – Set up SSL on the web server for secure communication	84
Push the image to the newly created Docker registry	87
Summary	88
Chapter 6: Running Services in a Container	89
A brief overview of container networking	90
Envisaging the Container as a Service	93
Building an HTTP server image	93
Running the HTTP server Image as a Service	95
Connecting to the HTTP service	96
Exposing container services	97
Publishing container ports – the -p option	98
Network Address Translation for containers	99
Retrieving the container port	100
Binding a container to a specific IP address	102
Auto-generating the Docker host port	103
Port binding using EXPOSE and the -P option	105
Summary	108
Chapter 7: Sharing Data with Containers	109
The data volume	110
Sharing host data	113
The practicality of host data sharing	117
Sharing data between containers	119
Data-only containers	119
Mounting data volume from other containers	120
The practicality of data sharing between containers	122
Avoiding common pitfalls	125
Directory leaks	125
The undesirable effect of data volume	126
Summary	128

Chapter 8: Orchestrating Containers	129
Linking containers	130
Orchestration of containers	138
Orchestrate containers using docker-compose	140
Installing docker-compose	141
The docker-compose.yml file	142
The docker-compose command	144
Common usage	145
Summary	150
Chapter 9: Testing with Docker	151
A brief overview of the test-driven development	152
Testing your code inside Docker	152
Running the test inside a container	157
Using a Docker container as a runtime environment	159
Integrating Docker testing into Jenkins	162
Preparing the Jenkins environment	162
Automating the Docker testing process	167
Summary	173
Chapter 10: Debugging Containers	175
Process level isolation for Docker containers	176
Control groups	180
Debugging a containerized application	181
The Docker exec command	182
The Docker ps command	183
The Docker top command	184
The Docker stats command	185
The Docker events command	186
The Docker logs command	186
Installing and using nsenter	187
Summary	188
Chapter 11: Securing Docker Containers	191
Are Docker containers secure enough?	192
The security facets – virtual machines versus Docker containers	192
The security features of containers	194
Resource isolation	195
Resource accounting and control	196

The root privilege – impacts and best practices	197
The trusted user control	197
Non-root containers	197
Loading the Docker images and the security implications	199
The emerging security approaches	199
Security-Enhanced Linux for container security	199
SELinux-inspired benefits	201
The best practices for container security	203
Digital signature verification	206
Secure deployment guidelines for Docker	207
The future	208
Summary	209
Index	211

Preface

We have been fiddling with virtualization techniques and tools for quite a long time now in order to establish the much-demanded software portability. The inhibiting dependency factor between software and hardware needs to be decimated by leveraging virtualization, a kind of beneficial abstraction, through an additional layer of indirection. The idea is to run any software on any hardware. This is achieved by creating multiple virtual machines (VMs) out of a single physical server, with each VM having its own operating system (OS). Through this isolation, which is enacted through automated tools and controlled resource sharing, heterogeneous applications are accommodated in a physical machine.

With virtualization, IT infrastructures become open, programmable, remotely monitorable, manageable, and maintainable. Business workloads can be hosted in appropriately-sized virtual machines and delivered to the outside world, ensuring broader and more frequent utilization. On the other hand, for high-performance applications, virtual machines across multiple physical machines can be readily identified and rapidly combined to guarantee any kind of high-performance requirement.

The virtualization paradigm has its own drawbacks. Because of the verbosity and bloatedness (every VM carries its own operating system), VM provisioning typically takes a while, the performance goes down due to excessive usage of computational resources, and so on. Furthermore, the growing need for portability is not fully met by virtualization. Hypervisor software from different vendors comes in the way of ensuring application portability. Differences in the OS and application distributions, versions, editions, and patches hinder smooth portability. Computer virtualization has flourished, whereas the other, closely associated concepts of network and storage virtualization are just taking off. Building distributed applications through VM interactions invites and involves some practical difficulties.

Let's move on to containerization. All of these barriers contribute to the unprecedented success of the idea of containerization. A container generally contains an application, and all of the application's libraries, binaries, and other dependencies are stuffed together to be presented as a comprehensive, yet compact, entity for the outside world. Containers are exceptionally lightweight, highly portable, easily and quickly provisionable, and so on. Docker containers achieve native system performance. The greatly articulated DevOps goal gets fully fulfilled through application containers. As best practice, it is recommended that every container hosts one application or service.

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.

This book is meticulously designed and developed in order to empower developers, cloud architects, 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.

What this book covers

Chapter 1, Getting Started with Docker, talks about the Docker platform and how it simplifies and speeds up the process of realizing containerized workloads to be readily deployed and run on a variety of platforms. This chapter also has step-by-step details on installing the Docker engine, downloading a Docker image from the centralized Docker Hub, creating a Docker container out of that image, and troubleshooting the Docker container.

Chapter 2, Handling Docker Containers, is primarily meant to expound the commands required to manage Docker images and containers. This chapter provides the basic Docker terminologies needed to understand the output of Docker commands. Other details covered here include starting an interactive session inside a container, managing your images, running containers, and tracking changes inside containers.

Chapter 3, Building Images, introduces Docker's integrated image building system. The other important topics covered in this chapter include a quick overview of a Dockerfile's syntax and a bit of theory on how Docker stores images.

Chapter 4, Publishing Images, focuses on publishing images on the centralized Docker Hub and how to get the most out of the Docker Hub. The other important contents in the chapter include greater details about the Docker Hub, how to push images to the Docker Hub, the automatic building of images, creating organizations on Docker Hub, and finally private repositories.

Chapter 5, Running Your Private Docker Infrastructure, explains how corporates can set up their own private repositories. Due to certain reasons, corporates may not want to host specific Docker images in publicly-available image repositories, such as the Docker Hub. Here, the need for their own private repository to keep up those images arises. This chapter has all of the information required to set up and sustain private repositories.

Chapter 6, Running Services in a Container, illustrates how a web application can be run inside a Docker container as a service, and how to expose the service for the outside world to find and access it. How the appropriate Dockerfile gets developed to simplify this task is also described in detail.

Chapter 7, Sharing Data with Containers, shows you how to use Docker's volumes feature to share data between the Docker host and its containers. The other topics covered here are how to share data between containers, the common use cases, and the typical pitfalls to avoid.

Chapter 8, Orchestrating Containers, focuses on orchestrating multiple containers towards composite, containerized workloads. It is a well-known truth that orchestration plays a major role in producing composite applications. This chapter includes some information about orchestration and the toolset made available for enabling the process of orchestration. Finally, you will find a well-orchestrated example of how containers can be orchestrated to bring forth highly reusable and business-aware containers.

Chapter 9, Testing with Docker, focuses on testing your code inside Docker images. In this chapter, you find out how to run the tests inside an ad hoc Docker image. Finally, you come across details of how to integrate Docker testing into a continuous integration server, such as Jenkins.

Chapter 10, Debugging Containers, teaches you how to debug applications running inside containers. Also, the details regarding how Docker ensures that processes running inside containers are isolated from the outside world are covered. Furthermore, descriptions of the usage of the `nsenter` and `nsinit` tools for effective debugging are included.

Chapter 11, Securing Docker Containers, is crafted to explain the brewing security and privacy challenges and concerns, and how they are addressed through the liberal usage of competent standards, technologies, and tools. This chapter inscribes the mechanism on dropping user privileges inside an image. There is also a brief introduction on how the security capabilities introduced in SELinux come in handy when securing Docker containers.

What you need for this book

The Docker platform requires a 64-bit hardware system to run on. Docker applications have been developed on Ubuntu 14.04 for this book, but this does not mean that the Docker platform cannot run on other Linux distributions, such as Redhat, CentOS, CoreOS, and so on. However, the Linux kernel version must be 3.10 or above.

Who this book is for

If you are an application developer who wants to learn about Docker in order to utilize its features for application deployment, then this book is for you. No prior knowledge of Docker is required.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

"If the `docker` service is running, then this command will print the status as `start/running`, along with its process ID."


A block of code is set as follows:


```
FROM busybox:latest
CMD echo Hello World!!
```

Any command-line input or output is written as follows:

```
$ sudo docker tag 224affbf9a65 localhost:5000/vinoddandy/  
dockerfileimageforhub
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Select the **Docker** option, which is in the drop-down menu, and then click on **Launch Now**."

[ Warnings or important notes appear in a box like this.]

[ Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

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 pathbreaking 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)**. However, as indicated in this chapter, there are pragmatic workarounds for running Docker on other mainstream operating systems, such as Windows, Mac, and so on. The Docker container has its own process space and network interface. It can also run things as root, and have its own `/sbin/init`, which can be different from the host machines'.

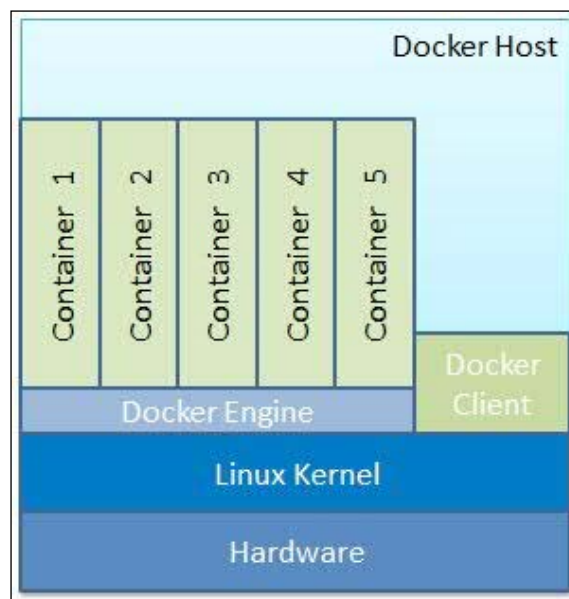
In a nutshell, the Docker solution lets us quickly assemble composite, enterprise-scale, and business-critical applications. For doing this, we can use different and distributed software components: Containers eliminate the friction that comes with shipping code to distant locations. Docker also lets 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

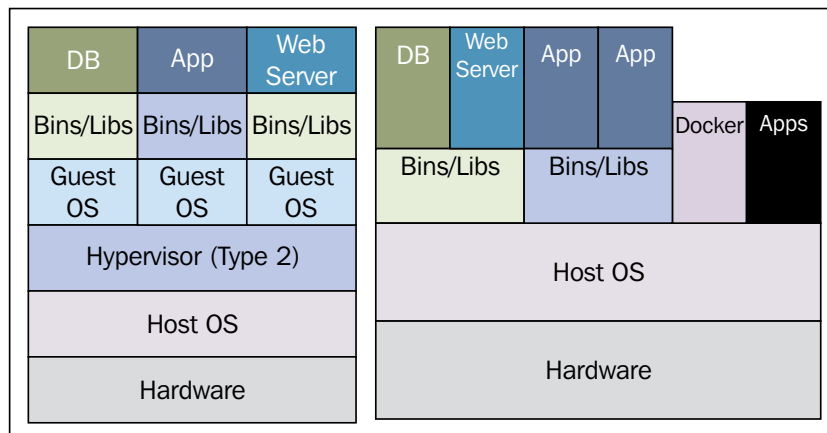
Suppose that we want to directly run the containers on a Linux machine. The Docker engine produces, monitors, and manages multiple containers as illustrated in the following diagram:



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 SSH, TTY, and the other security functionalities in the containers. 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)**: This is 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 *LXC* on 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.

In this book, considering the surging popularity and the mass adoption happening to Docker, we have chosen to dig deeper, dwell in detail on the Docker platform, the one-stop solution for the simplified and streamlined containerization movement.

Installing the Docker engine

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 RedHat, Fedora, and CentOS.



We have chosen the *Ubuntu Trusty 14.04 (LTS) (64-bit)* Linux distribution for all practical purposes.

Installing from the Ubuntu package repository

This section explains the steps involved in installing the Docker engine from the Ubuntu package repository in detail. At the time of writing this book, the Ubuntu repository had packaged Docker 1.0.1, whereas the latest version of Docker was 1.5. We strongly recommend installing Docker version 1.5 or greater by using any one of the methods described in the next section.

However, if for any reason you have to install the Ubuntu packaged version, then please follow the steps described here:

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
```



Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

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
```



The Docker package is called `docker.io` because an older version of the Ubuntu package was called `docker`. As a result, 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`.

3. 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
```



The official Ubuntu package does not come with the latest stable version of `docker`.

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 `curl` command:

```
$ sudo curl -sSL https://get.docker.io/ | sh
```

- For `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.

Understanding the Docker setup

It's important to understand Docker's components and their versions, storage, execution drivers, file locations, and so on. Incidentally, the quest for understanding the Docker setup would also reveal whether the installation was successful or not. You can accomplish this by using two `docker` subcommands, namely `docker version`, and `docker info`.

Let's start our docker journey with the `docker version` subcommand, as shown here:

```
$ sudo docker version
Client version: 1.5.0
Client API version: 1.17
Go version (client): go1.4.1
Git commit (client): a8a31ef
OS/Arch (client): linux/amd64
Server version: 1.5.0
Server API version: 1.17
Go version (server): go1.4.1
Git commit (server): a8a31ef
```

Although the `docker version` subcommand lists many lines of text, as a Docker user, you should know what these following output lines mean:

- The client version
- The client API version
- The server version
- The server API version

The client and server versions that have been considered here are 1.5.0 and the client API and the server API, versions 1.17.

If we dissect the internals of the `docker version` subcommand, then it will first list the client-related information that is stored locally. Subsequently, it will make a REST API call to the server over HTTP to obtain the server-related details.

Let's learn more about the Docker environment using the `docker info` subcommand:

```
$ sudo docker -D info
Containers: 0
Images: 0
Storage Driver: aufs
```

```
Root Dir: /var/lib/docker/aufs
Backing Filesystem: extfs
Dirs: 0
Execution Driver: native-0.2
Kernel Version: 3.13.0-45-generic
Operating System: Ubuntu 14.04.1 LTS
CPUs: 4
Total Memory: 3.908 GiB
Name: dockerhost
ID: ZNXR:QOSY:IGKJ:ZLYU:G4P7:AXVC:2KAJ:A3Q5:YCRQ:IJD3:7RON:IJ6Y
Debug mode (server): false
Debug mode (client): true
Fds: 10
Goroutines: 14
EventsListeners: 0
Init Path: /usr/bin/docker
Docker Root Dir: /var/lib/docker
WARNING: No swap limit support
```

As you can see in the output of a freshly installed Docker engine, the number of Containers and Images is invariably nil. The Storage Driver has been set up as aufs, and the directory has been given the `/var/lib/docker/aufs` location. The Execution Driver has been set to the native mode. This command also lists details, such as the Kernel Version, the Operating System, the number of CPUs, the Total Memory, and Name, the new Docker hostname.

Client server communication

On Linux installations, Docker is usually programmed for carrying out server-client communication by using the Unix socket (`/var/run/docker.sock`). Docker also has an IANA registered port, which is 2375. However, for security reasons, this port is not enabled by default.

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.

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.

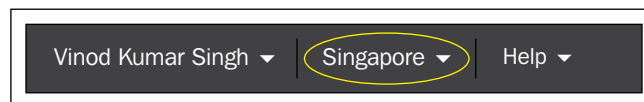
The AWS Elastic Beanstalk service supports the following:

- 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.

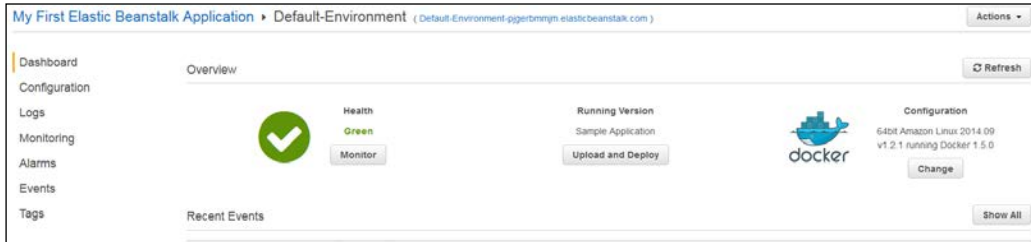
Currently, AWS supports the latest Docker version, which is 1.5.

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:

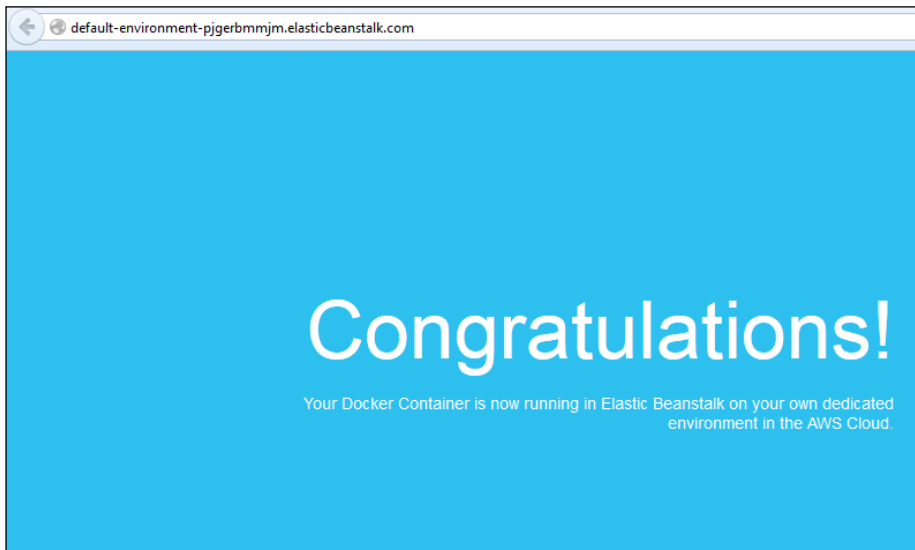
1. Log in to the AWS Elastic Beanstalk console by using this <https://console.aws.amazon.com/elasticbeanstalk/> URL.
2. Select a region where you want to deploy your application, as shown here:



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:



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.

Summary

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.

In this chapter, we have exclusively concentrated on the practical side of Docker for giving you a head start in learning about the most promising technology. We have listed the appropriate steps and tips for effortlessly installing the Docker engine in different environments, for leveraging and for building, installing, and running a few sample Docker containers, both in local as well as remote environments. We will dive deep into the world of Docker and dig deeper to extract and share tactically and strategically sound information with you in the ensuing chapters. Please read on to gain the required knowledge about advanced topics, such as container integration, orchestration, management, governance, security, and so on, through the Docker engine. We will also discuss a bevy of third-party tools.

2

Handling Docker Containers

In the previous chapter, we explained stimulating and sustainable concepts, which showed the Docker's way of crafting futuristic and flexible application-aware containers. We discussed all the relevant details of producing the Docker containers in multiple environments (on-premise as well as off-premise). Using these techniques, you can easily replicate these features in your own environments to get a rewarding experience. Therefore, the next step for us is to understand the container's life cycle aspects in a decisive manner. You will learn the optimal utilization of containers of our own as well as those of other third-party containers in an effective and risk-free way. Containers are to be found, assessed, accessed, and leveraged toward bigger and better applications. There are several tools that have emerged to streamline the handling of containers.

In this chapter, we will dig deeper and describe the critical aspects of container handling at length. A number of practical tips and execution commands for the leveraging of containers will also be discussed in this chapter.

In this chapter, we will cover the following topics:

- Clarifying the Docker terms
- Working with the Docker images and containers
- The meaning of the Docker registry and its repository
- The Docker Hub Registry
- Searching the Docker images
- Working with an interactive container
- Tracking the changes inside the containers
- Controlling and housekeeping the Docker containers
- Building images from containers
- Launching a container as a daemon

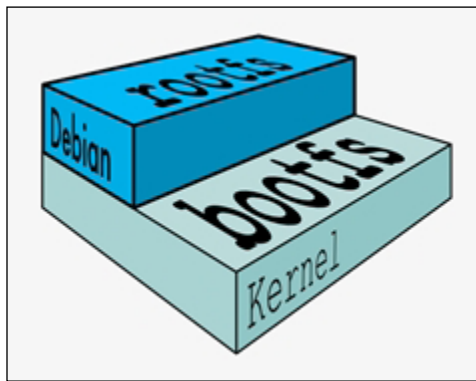
Clarifying the Docker terms

To make this chapter substantially simpler to understand and to minimize any kind of ambiguity, the frequently used terms will be explained in the following section.

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.

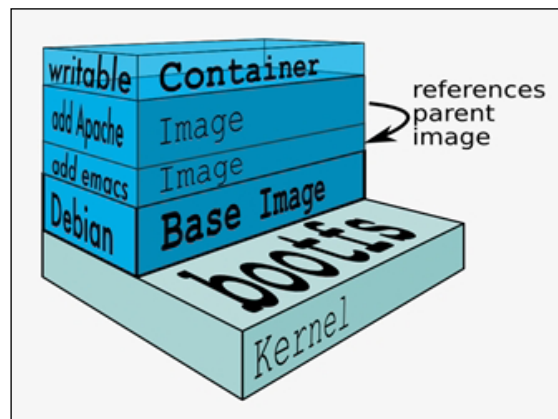
A base image has been illustrated here. Debian is the base image, and a variety of desired capabilities in the form of functional modules can be incorporated on the base image for arriving at multiple images:



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 one or 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.

Docker Registry

A **Docker Registry** is a place where the Docker images can be stored in order to be publicly found, accessed, and used by the worldwide developers for quickly crafting fresh and composite applications without any risks. Because all the stored images would have gone through multiple validations, verifications, and refinements, the quality of those images will be really high. Using the Docker `push` command, you can dispatch your Docker image to the Registry so that it is registered and deposited. As a clarification, the registry is for registering the Docker images, whereas the repository is for storing those registered Docker images in a publicly discoverable and centralized place. A Docker image is stored within a Repository in the Docker Registry. Each Repository is unique for each user or account.

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.

Working with Docker images

In the previous chapter, we demonstrated the typical `Hello World!` example by using a `busybox` image. Now there is a need for a close observation of the output of the `docker pull` subcommand, which is a standard command for downloading the Docker images. You would have noticed the presence of the `busybox:latest` text in the output text, and we will explain this mystery in a detailed manner by bringing in a small twist to the `docker pull` subcommand by adding the `-a` option, as shown here:

```
$ sudo docker pull -a busybox
```

Surprisingly, you will observe that the Docker engine downloads a few more images with the `-a` option. You can easily check the images that are available on the Docker host by running the `docker images` subcommand, which comes in handy, and it reveals more details with respect to `:latest` and the additional images that are downloaded by running this command. Let us run this command:


```
$ sudo docker images
```

You will get the list of images, as follows:

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
busybox	ubuntu-14.04	f6169d24347d	3 months ago	5.609 MB
busybox	ubuntu-12.04	492dad4279ba	3 months ago	5.455 MB
busybox	buildroot-2014.02	4986bf8c1536	3 months ago	2.433 MB
busybox	latest	4986bf8c1536	3 months ago	2.433 MB
busybox	buildroot-2013.08.1	2aed48a4e41d	3 months ago	2.489 MB

Evidently, we have five items in the preceding list, and to gain a better understanding of those, we need to comprehend the information that is printed out by the Docker images subcommand. Here is a list of the possible categories:

- **REPOSITORY:** This is the name of the repository or image. In the preceding example, the repository name is `busybox`.
- **TAG:** This is the tag associated with the image, for example `buildroot-2014.02`, `ubuntu-14.04`, `latest`. One or more tags can be associated with one image.

 The `ubuntu-*` tagged images are built by using the `busybox-static` Ubuntu package and the `buildroot-*` tagged images are built from scratch by using the `buildroot` tool-chain.

- **IMAGE ID:** Every image is associated with a unique ID. The image ID is represented by using a 64 Hex digit long random number. By default, the Docker images subcommand will only show 12 Hex digits. You can display all the 64 Hex digits by using the `--no-trunc` flag (for example: `sudo docker images --no-trunc`).
- **CREATED:** Indicates the time when the image was created.
- **VIRTUAL SIZE:** Highlights the virtual size of the image.