Jaroslaw Krochmalski

Docker and Kubernetes for Java Developers

Scale, deploy, and monitor multi-container applications

Packt

Docker and Kubernetes for Java Developers

Scale, deploy, and monitor multi-container applications

Jaroslaw Krochmalski

Packt>

BIRMINGHAM - MUMBAI

Docker and Kubernetes for Java Developers

First published: August 2017

Production reference: 1240817

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

ISBN 978-1-78646-839-0

www.packtpub.com

Credits

Author
Jaroslaw Krochmalski

Reviewer
Pierre Mavro

Commissioning Editor
Vijin Boricha

Acquisition Editor
Prachi Bisht

Content Development Editor
Trusha Shriyan

Technical Editor
Varsha Shivhare

Copy Editor
Safis Editing

Project Coordinator
Kinjal Bari

Proofreader
Safis Editing

Indexer
Mariammal Chettiyar

Graphics
Kirk D'Penha

Production Coordinator
Shantanu Zagade

About the Author

Jaroslaw Krochmalski is a passionate software designer and developer who specializes in the financial domain. He has over 12 years of experience in software development. He is a clean-code and software craftsmanship enthusiast. He is a certified scrum master and a fan of Agile. His professional interests include new technologies in web application development, design patterns, enterprise architectures, and integration patterns.

He has been designing and developing software professionally since 2000 and has been using Java as his primary programming language since 2002. In the past, he has worked for companies such as **Kredyt Bank** (**KBC**) and Bank BPS on many large-scale projects, such as international money orders, express payments, and collection systems. He currently works as a consultant at Danish company 7N as an infrastructure architect for the Nykredit bank. You can reach him via Twitter at `@jkroch` or by email at `jarek@finsys.pl`.

About the Reviewer

Pierre Mavro lives in a suburb of Paris. He's an open source software lover and has been working with Linux for more than 10 years now. Currently, he is working as a lead SRE at Criteo, where he manages distributed systems and NoSQL technologies. During the last few years, he has been designing high-availability infrastructures, public and private cloud infrastructures, and worked for a high-frequency trading company. He also wrote a book on MariaDB named *MariaDB High Performance*. He’s also one of the co-founders of Nousmotards, an application for riders.

www.PacktPub.com

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://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

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

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https:/ / www. amazon. com/ dp/ 1786468395`.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

Imagine creating and testing Java EE applications on Apache Tomcat or Wildfly in minutes, along with deploying and managing Java applications swiftly. Sounds too good to be true? You have a reason to cheer, because such scenarios are possible by leveraging Docker and Kubernetes.

This book will start by introducing Docker and delve deep into its networking and persistent storage concepts. You will be then introduced to the concept of microservices and learn how to deploy and run Java microservices as Docker containers. Moving on, the book will focus on Kubernetes and its features. You will start by running the local cluster using Minikube. The next step will be to deploy your Java service in the real cloud, on Kubernetes running on top of Amazon AWS. At the end of the book, you will get hands-on experience of some more advanced topics to further extend your knowledge of Docker and Kubernetes.

What this book covers

`Chapter 1`, *Introduction to Docker*, introduces the reasoning behind Docker and presents the differences between Docker and traditional virtualization. The chapter also explains basic Docker concepts, such as images, containers, and Dockerfiles.

`Chapter 2`, *Networking and Persistent Storage*, explains how networking and persistent storage work in Docker containers.

`Chapter 3`, *Working with Microservices*, presents an overview of what microservices are and explains their advantages in comparison to monolithic architectures.

`Chapter 4`, *Creating Java Microservices*, explores a recipe for quickly constructing Java microservice, by utilizing either Java EE7 or the Spring Boot.

`Chapter 5`, *Creating Images with Java Applications*, teaches how to package the Java microservices into Docker images, either manually or from the Maven build file.

`Chapter 6`, *Running Containers with Java Applications*, shows how to run a containerized Java application using Docker.

`Chapter 7`, *Introduction to Kubernetes*, introduces the core concepts of Kubernetes, such as Pods, nodes, services, and deployments.

`Chapter 8`, *Using Kubernetes with Java*, shows how to deploy Java microservices, packaged as a Docker image, on the local Kubernetes cluster.

`Chapter 9`, *Working with Kubernetes API*, shows how the Kubernetes API can be used to automate the creation of Kubernetes objects such as services or deployments. This chapter gives examples of how to use the API to get information about the cluster's state.

`Chapter 10`, *Deploying Java on Kubernetes in the Cloud*, shows the reader how to configure Amazon AWS EC2 instances to make them suitable to run a Kubernetes cluster. This chapter also gives precise instructions on how to create a Kubernetes cluster on the Amazon AWS cloud.

`Chapter 11`, *More Resources*, explores how Java and Kubernetes point the reader to additional resources available on the internet that are of high quality, to further extend knowledge about Docker and Kubernetes.

What you need for this book

For this book, you will need any decent PC or Mac, capable of running a modern version of Linux, Windows 10 64-bit, or macOS.

Who this book is for

This book is for Java developers, who would like to get into the world of containerization. The reader will learn how Docker and Kubernetes can help with deployment and management of Java applications on clusters, either on their own infrastructure or in the cloud.

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: "The Dockerfile is used to create the image when you run the `docker build` command." A block of code is set as follows:

```
{
"apiVersion": "v1",
"kind": "Pod",
"metadata":{
```

```
"name": ”rest_service”,
"labels": {
"name": "rest_service"
}
},
"spec": {
"containers": [{
"name": "rest_service",
"image": "rest_service",
"ports": [{"containerPort": 8080}],
}]
}
}
```

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

```
docker rm $(docker ps -a -q -f status=exited)
```

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: "Clicking the **Skip For Now** will take you to the the images list without logging into the Docker Hub."

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 email `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 for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files emailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your email address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Docker-and-Kubernetes-for-Java-Developers`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `http:/ / www. packtpub. com/ sites/ default/ files/ downloads/ DockerandKubernetesforJavaDevelopers_ ColorImages. pdf`.

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
Introduction to Docker

The first thing we will do in this chapter will be to explain the reasoning behind Docker and its architecture. We will cover Docker concepts such as images, layers, and containers. Next, we will install Docker and learn how to pull a sample, basic Java application image from the `remote` registry and run it on the local machine.

Docker was created as the internal tool in the platform as a service company, dotCloud. In March 2013, it was released to the public as open source. Its source code is freely available to everyone on GitHub at: `https://github.com/docker/docker`. Not only do the core Docker Inc. team work on the development of Docker, there are also a lot of big names sponsoring their time and effort to enhance and contribute to Docker such as Google, Microsoft, IBM, Red Hat, Cisco systems, and many others. Kubernetes is a tool developed by Google for deploying containers across clusters of computers based on best practices learned by them on Borg (Google's homemade container system). It compliments Docker when it comes to orchestration, automating deployment, managing, and scaling containers; it manages workloads for Docker nodes by keeping container deployments balanced across a cluster. Kubernetes also provides ways for containers to communicate with each other, without the need for opening network ports. Kubernetes is also an open source project, living on the GitHub at `https://github.com/kubernetes/kubernetes`. Everyone can contribute. Let's begin our journey with Docker first. The following will be covered in:

- We will start with the basic idea behind this wonderful tool and show the benefits gained from using it, in comparison to traditional virtualization
- We will install Docker on three major platforms: macOS, Linux, and Windows

The idea behind Docker

The idea behind Docker is to pack an application with all the dependencies it needs into a single, standardized unit for the deployment. Those dependencies can be binaries, libraries, JAR files, configuration files, scripts, and so on. Docker wraps up all of it into a complete filesystem that contains everything your Java application needs to run the virtual machine itself, the application server such as Wildfly or Tomcat, the application code, and `runtime` libraries, and basically everything you would install and deploy on the server to make your application run. Packaging all of this into a complete image guarantees that it is portable; it will always run in the same way, no matter what environment it is deployed in. With Docker, you can run Java applications without having to install a Java runtime on the host machine. All the problems related to incompatible JDK or JRE, wrong version of the application server, and so on are gone. Upgrades are also easy and effortless; you just run the new version of your container on the host.

If you need to do some cleanup, you can just destroy the Docker image and it's as though nothing ever happened. Think about Docker, not as a programming language or a framework, but rather as a tool that helps in solving the common problems such as installing, distributing, and managing the software. It allows developers and DevOps to build, ship, and run their code anywhere. Anywhere means also on more than one machine, and this is where Kubernetes comes in handy; we will shortly get back to it.

Having all of your application code and runtime dependencies packaged as a single and complete unit of software may seem the same as a virtualization engine, but it's far from that, as we will explain now. To fully get to know what Docker really is, first we need to understand the difference between traditional virtualization and containerization. Let's compare those two technologies now.

Virtualization and containerization compared

A traditional virtual machine represents the hardware-level virtualization. In essence, it's a complete, virtualized physical machine with BIOS and an operating system installed. It runs on top of the host operating system. Your Java application runs in the virtualized environment as it would normally do on your own machine. There are a lot of advantages from using virtual machines for your applications. Each virtual machine can have a totally different operating system; those can be different Linux flavors, Solaris, or Windows, for example. Virtual machines are also very secure by definition; they are totally isolated, complete operating systems in a box.

However, nothing comes without a price. Virtual machines contain all the features that an operating system needs to have to be operational: core system libraries, device drivers, and so on. Sometimes they can be resource hungry and heavyweight. Virtual machines require full installation, which sometimes can be cumbersome and not so easy to set up. Last, but not least, you will need more compute power and resources to execute your application in the virtual machine the hypervisor needs to first import the virtual machine and then power it up and this takes time. However, I believe, when it comes to running Java applications, having the complete virtualized environment is not something that we would want very often. Docker comes to the rescue with the concept of containerization. Java applications (but of course, it's not limited to Java) run on Docker in an isolated environment called a container. A container is not a virtual machine in the popular sense. It behaves as a kind of operating system virtualization, but there's no emulation at all. The main difference is that while each traditional virtual machine image runs on an independent guest operating system, the Docker containers run within the same kernel running on the host machine. A container is self-sufficient and isolated not only from the underlying OS, but from other containers as well. It has its own separated filesystem and environment variables. Naturally, containers can communicate with each other (as an application and a database container for example) and also can share the files on disk. Here comes the main difference when comparing to traditional virtualization because the containers run within the same kernel they utilize fewer system resources. All the operating system core software is removed from the Docker image. The base container can be, and usually is, very lightweight. There is no overhead related to a classic virtualization hypervisor and a guest operating system. This way you can achieve almost bare metal, core performance for your Java applications. Also, the startup time of a containerized Java application is usually very low due to the minimal overhead of the container. You can also roll-out hundreds of application containers in seconds to reduce the time needed for provisioning your software. We will do this using Kubernetes in one of the coming chapters. Although Docker is quite different from the traditional virtualization engines. Be aware that containers cannot substitute virtual machines for all use cases; a thoughtful evaluation is still required to determine what is best for your application. Both solutions have their advantages. On the one hand, we have the fully isolated secure virtual machine with average performance. On the other hand, we have the containers that are missing some of the key features, but are equipped with high performance that can be provisioned very fast. Let's see what other benefits you will get when using Docker containerization.

Benefits from using Docker

As we have said before, the major visible benefit of using Docker will be very fast performance and short provisioning time. You can create or destroy containers quickly and easily. Containers share resources such as the operating system's kernel and the needed libraries efficiently with other Docker containers. Because of that, multiple versions of an application running in containers will be very lightweight. The result is faster deployment, easier migration, and startup times.

Docker can be especially useful when deploying Java microservices. We will get back to microservices in detail in one of the coming chapters. A microservices application is composed of a series of discrete services, communicating with others via an API. Microservices break an app into a large number of small processes. They are the opposite of the monolithic applications, which run all operations as a single process or a set of large processes.

Using Docker containers enables you to deploy ready-to-run software, which is portable and extremely easy to distribute. Your containerized application simply runs within its container; there's no need for installation. The lack of an installation process has a huge advantage; it eliminates problems such as software and library conflicts or even driver compatibility issues. Docker containers are portable; they can be run from anywhere: your local machine, a remote server, and private or public cloud. All major cloud computing providers, such as **Amazon Web Services** (**AWS**) and Google's compute platform support Docker now. A container running on, let's say, an Amazon EC2 instance, can easily be transferred to some other environment, achieving exactly the same consistency and functionality. The additional level of abstraction Docker provides on the top of your infrastructure layer is an indispensable feature. Developers can create the software without worrying about the platform it will later be run on. Docker has the same promise as Java; write once, run anywhere; except instead of code, you configure your server exactly the way you want it (by picking the operating system, tuning the configuration files, installing dependencies) and you can be certain that your server template will run exactly the same on any host that runs Docker.

Because of Docker's reproducible build environment, it's particularly well suited for testing, especially in your continuous integration or continuous delivery flow. You can quickly boot up identical environments to run the tests. And because the container images are all identical each time, you can distribute the workload and run tests in parallel without a problem. Developers can run the same image on their machine that will be run in production later, which again has a huge advantage in testing.

The use of Docker containers speeds up continuous integration. There are no more endless build-test-deploy cycles; Docker containers ensure that applications run identically in development, test, and production environments. The code grows over time and becomes more and more troublesome. That's why the idea of an immutable infrastructure becomes more and more popular nowadays and the concept of containerization has become so popular. By putting your Java applications into containers, you can simplify the process of deployment and scaling. By having a lightweight Docker host that needs almost no configuration management, you manage your applications simply by deploying and redeploying containers to the host. And again, because the containers are very lightweight, it takes only seconds.

We have been talking a lot about images and containers, without getting much into the details. Let's do it now and see what Docker images and containers are.

Docker concepts - images and containers

When dealing with Kubernetes, we will be working with Docker containers; it is an open source container cluster manager. To run our own Java application, we will need to create an image first. Let's begin with the concept of Docker images.

Images

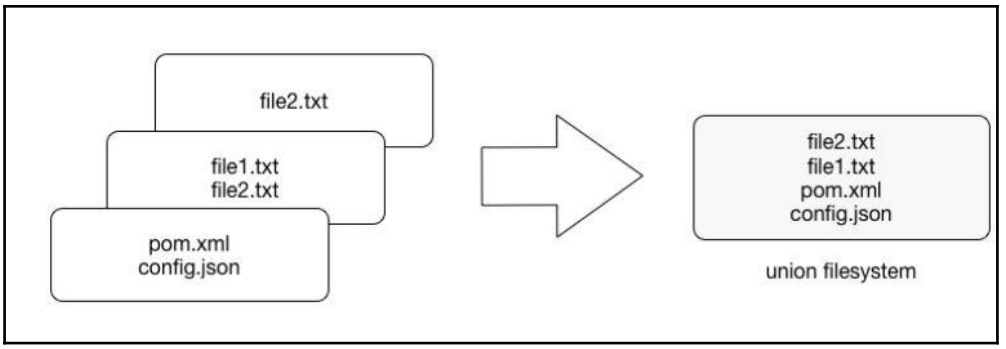
Think of an image as a read-only template which is a base foundation to create a container from. It's same as a recipe containing the definition of everything your application needs to operate. It can be Linux with an application server (such as Tomcat or Wildfly, for example) and your Java application itself. Every image starts from a base image; for example, Ubuntu; a Linux image. Although you can begin with a simple image and build your application stack on top of it, you can also pick an already prepared image from the hundreds available on the Internet. There are a lot of images especially useful for Java developers: `openjdk`, `tomcat`, `wildfly`, and many others. We will use them later as a foundation for our own images. It's a lot easier to have, let's say, Wildfly installed and configured properly as a starting point for your own image. You can then just focus on your Java application. If you're a novice in building images, downloading a specialized base image is a great way to get a serious speed boost in comparison to developing one by yourself.

Images are created using a series of commands, called instructions. Instructions are placed in the Dockerfile. The Dockerfile is just a plain text file, containing an ordered collection of `root` filesystem changes (the same as running a command that starts an application server, adding a file or directory, creating environmental variables, and so on.) and the corresponding execution parameters for use within a container runtime later on. Docker will read the Dockerfile when you start the process of building an image and execute the instructions one by one. The result will be the final image. Each instruction creates a new layer in the image. That image layer then becomes the parent for the layer created by the next instruction. Docker images are highly portable across hosts and operating systems; an image can be run in a Docker container on any host that runs Docker. Docker is natively supported in Linux, but has to be run in a VM on Windows and macOS. It's important to know that Docker uses images to run your code, not the Dockerfile. The Dockerfile is used to create the image when you run the `docker build` command. Also, if you publish your image to the Docker Hub, you publish a resulting image with its layers, not a source Dockerfile itself.

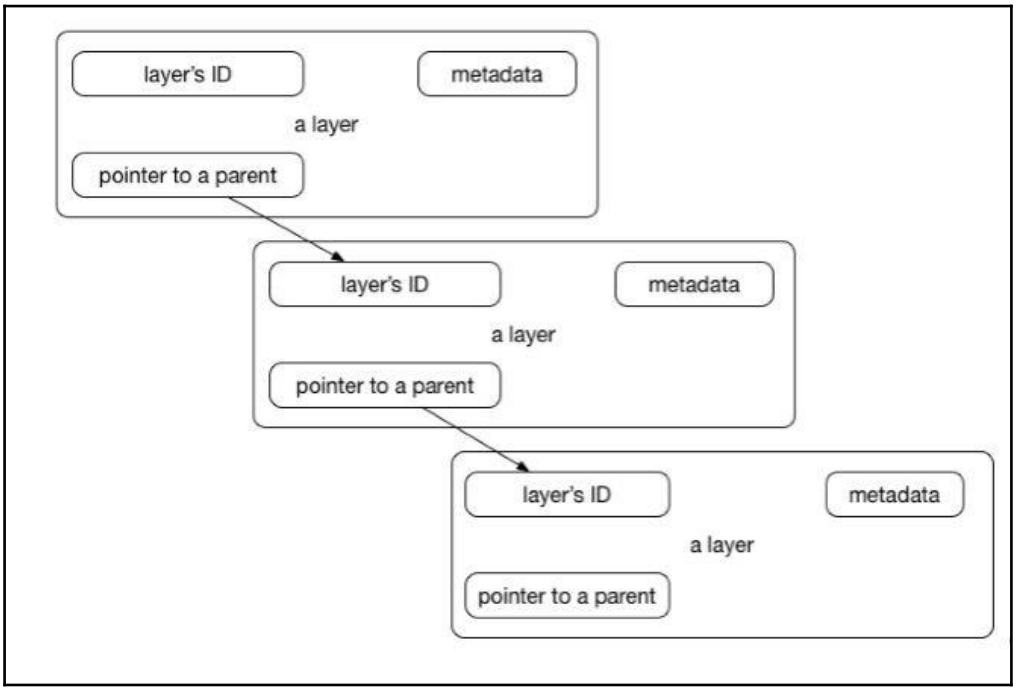
We have said before that every instruction in a Dockerfile creates a new layer. Layers are the internal nature of an image; Docker images are composed from them. Let's explain now what they are and what their characteristics are.

Layers

Each image consists of a series of layers which are stacked, one on top of the another. In fact, every layer is an intermediate image. By using the **union filesystem**, Docker combines all these layers into a single image entity. The union filesystem allows transparent overlaying files and directories of separate filesystems, giving a single, consistent filesystem as a result, as you can see the following diagram:



Contents and structure of directories which have the same path within these separate filesystems will be seen together in a single merged directory, within the new, virtual-like filesystem. In other words, the filesystem structure of the top layer will merge with the structure of the layer beneath. Files and directories which have the same path as in the previous layer will cover those beneath. Removing the upper layer will again reveal and expose the previous directory content. As we have mentioned earlier, layers are placed in a stack, one on the top of another. To maintain the order of layers, Docker utilizes the concept of layer IDs and pointers. Each layer contains the ID and a pointer to its parent layer. A layer without a pointer referencing the parent is the first layer in the stack, a base. You can see the relation in the following diagram:



Layers have some interesting features. First, they are reusable and cacheable. The pointer to a parent layer you can see in the previous diagram is important. As Docker is processing your Dockerfile it's looking at two things: the Dockerfile instruction being executed and the parent image. Docker will scan all of the children of the parent layer and look for one whose command matches the current instruction. If a match is found, Docker skips to the next Dockerfile instruction and repeats the process. If a matching layer is not found in the cache, a new one is created. For the instructions that add files to your image (we will get to know them later in detail), Docker creates a checksum for each file contents. During the building process, this checksum is compared against the checksum of the existing images to check if the layer can be reused from the cache. If two different images have a common part, let's say a Linux shell or Java runtime for example, Docker, which tracks all of the pulled layers, will reuse the shell layer in both of the images. It's a safe operation; as you already know, layers are read-only. When downloading another image, the layer will be reused and only the difference will be pulled from the Docker Hub. This saves time, bandwidth, and disk space of course, but it has another great advantage. If you modify your Docker image, for example by modifying your containerized Java application, only the application layer gets modified. After you've successfully built an image from your Dockerfile, you will notice that subsequent builds of the same Dockerfile finish a lot faster. Once Docker caches an image layer for an instruction, it doesn't need to be rebuilt. Later on, instead of distributing the whole image, you push just the updated part. It makes the process simpler and faster. This is especially useful if you use Docker in your continuous deployment flow: pushing a Git branch will trigger building an image and then publishing the application for users. Due to the layer-reuse feature, the whole process is a lot faster.

The concept of reusable layers is also a reason why Docker is so lightweight in comparison to full virtual machines, which don't share anything. It is thanks to layers that when you pull an image, you eventually don't have to download all of its filesystem. If you already have another image that has some of the layers of the image you pull, only the missing layers are actually downloaded. There is a word of warning though, related to another feature of layers: apart from being reusable, layers are also additive. If you create a large file in the container, then make a commit (we will get to that in a while), then delete the file, and do another commit; this file will still be present in the layer history. Imagine this scenario: you pull the base Ubuntu image, and install the Wildfly application server. Then you change your mind, uninstall the Wildfly and install Tomcat instead. All those files removed from the Wildfly installation will still be present in the image, although they have been deleted. Image size will grow in no time. Understanding of Docker's layered filesystem can make a big difference in the size of your images. Size can become a problem when you publish your images to a registry; it takes more requests and is longer to transfer.

Large images become an issue when thousands of containers need to be deployed across a cluster, for example. You should always be aware of the additivity of layers and try to optimize the image at every step of your Dockerfile, the same as using the command chaining, for example. We will be using the command chaining technique later on, when creating our Java application images.

Because layers are additive, they provide a full history of how a specific image was built. This gives you another great feature: the possibility to make a rollback to a certain point in the image's history. Since every image contains all of its building steps, we can easily go back to a previous step if we want to. This can be done by tagging a certain layer. We will cover image tagging later in our book.

Layers and images are closely related to each other. As we have said before, Docker images are stored as a series of read-only layers. This means that once the container image has been created, it does not change. But having all the filesystem read-only would not make a lot of sense. What about modifying an image? Or adding your software to a base web server image? Well, when we start a container, Docker actually takes the read-only image (with all its read-only layers) and adds a writable layer on top of the layers stack. Let's focus on the containers now.

Containers

A running instance of an image is called a container. Docker launches them using the Docker images as read-only templates. If you start an image, you have a running container of this image. Naturally, you can have many running containers of the same image. In fact, we will do it very often a little bit later, using Kubernetes.

To run a container, we use the `docker run` command:

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

There are a lot of `run` command options and switches that can be used; we will get to know them later on. Some of the options include the network configuration, for example (we will explain Docker's networking concepts in `Chapter 2`, *Networking and Persistent Storage*). Others, the same as the `-it` (from interactive), tell the Docker engine to behave differently; in this case, to make the container interactive and to attach a terminal to its output and input. Let's just focus on the idea of the container to better understand the whole picture. We are going to use the `docker run` command in a short while to test our setup.

So, what happens under the hood when we run the `docker run` command? Docker will check if the image that you would like to run is available on your local machine. If not, it will be pulled down from the `remote` repository. The Docker engine takes the image and adds a writable layer on top of the image's layers stack. Next, it initializes the image's name, ID, and resource limits, such as CPU and memory. In this phase, Docker will also set up a container's IP address by finding and attaching an available IP address from a pool. The last step of the execution will be the actual command, passed as the last parameter of the `docker run` command. If the `it` option has been used, Docker will capture and provide the container output, it will be displayed in the console. You can now do things you would normally do when preparing an operating system to run your applications. This can be installing packages (via `apt-get`, for example), pulling source code with Git, building your Java application using Maven, and so on. All of these actions will modify the filesystem in the top, writable layer. If you then execute the `commit` command, a new image containing all of your changes will be created, kind of frozen, and ready to be run later. To stop a container, use the `docker stop` command:

```
docker stop
```

A container when stopped will retain all settings and filesystem changes (in the top layer that is writeable). All processes running in the container will be stopped and you will lose everything in memory. This is what differentiates a stopped container from a Docker image.

To list all containers you have on your system, either running or stopped, execute the `docker ps` command:

```
docker ps –a
```

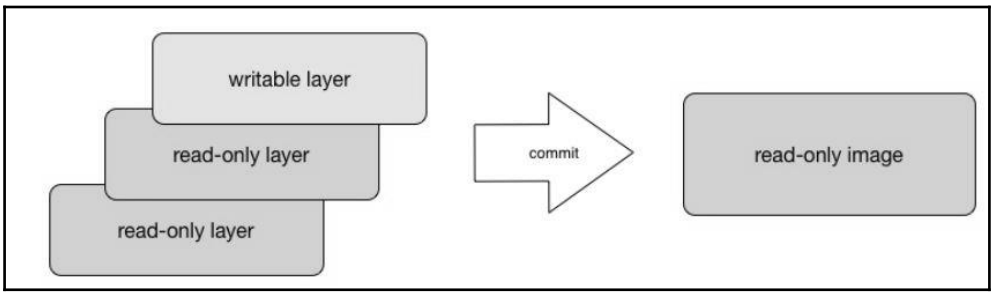
As a result, the Docker client will list a table containing container IDs (a unique identifier you can use to refer to the container in other commands), creation date, the command used to start a container, status, exposed ports, and a name, either assigned by you or the funny name Docker has picked for you. To remove a container, you can just use the `docker rm` command. If you want to remove a couple of them at once, you can use the list of containers (given by the `docker ps` command) and a filter:

```
docker rm $(docker ps –a –q –f status=exited)
```

We have said that a Docker image is always read-only and immutable. If it did not have the possibility to change the image, it would not be very useful. So how's the image modification possible except by, of course, altering a Dockerfile and doing a rebuild? When the container is started, the writable layer on top of the layers stack is for our disposal. We can actually make changes to a running container; this can be adding or modifying files, the same as installing a software package, configuring the operating system, and so on. If you modify a file in the running container, the file will be taken out of the underlying (parent) read-only layer and placed in the top, writable layer. Our changes are only possible in the top layer. The union filesystem will then cover the underlying file. The original, underlying file will not be modified; it still exists safely in the underlying, read-only layer. By issuing the `docker commit` command, you create a new read-only image from a running container (and all it changes in the writable layer):

```
docker commit <container-id> <image-name>
```

The `docker commit` command saves changes you have made to the container in the writable layer. To avoid data corruption or inconsistency, Docker will pause a container you are committing changes into. The result of the `docker commit` command is a brand new, read-only image, which you can create new containers from:



In response to a successful commit, Docker will output the full ID of a newly generated image. If you remove the container without issuing a `commit` first and then relaunch the same image again, Docker will start a fresh container without any of the changes made in the previously running container. In either case, with or without a `commit`, your changes to the filesystem will never affect the base image. Creating images by altering the top writable layer in the container is useful when debugging and experimenting, but it's usually better to use a Dockerfile to manage your images in a documented and maintainable way.