Community Experience Distilled

Developing with Docker

A fast-paced guide to change the way your organization deploys software at scale

Jaroslaw Krochmalski

[PACKT] PUBLISHING open source* community experience distilled

Table of Contents

1
Introduction to Docker

At the beginning, Docker was created as an internal tool by a Platform as a Service company, called dotCloud. Later on, in March 2013, it was released as open source. Apart from the Docker Inc. team, which is the main sponsor, there are some other big names contributing to the tool –`Red Hat`, `IBM`, Microsoft, `Google` and Cisco Systems, just to name a few. Software development today needs to be agile and react quickly for changes. We use methodologies like Scrum, estimate our work in story points and attend the daily stand-ups. But what about preparing our software for shipment and the deployment? Let's see how Docker fits into that scenario and can help us being agile. We will begin with a basic idea behind this wonderful tool.

The basic idea

The basic idea behind Docker is to pack an application with all of its dependencies (let it be binaries, libraries, configuration files, scripts, jars and so on) into a single, standardized unit for software development and deployment. Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, and system libraries – anything you can install on a server. This guarantees that it will always run in the same way, no matter what environment it will be deployed in. With Docker, you can build some Node.js or Java project (but you are of course not limited to those two) without having to install Node.js or Java on your host machine. Once you're done with it, you can just destroy the Docker image and it's as though nothing ever happened. It's not a programming language or a framework, rather think of it as about a tool that helps solving the common problems like installing, distributing and managing the software – it allows programmers and DevOps to build, ship and run their code anywhere.

You can think that maybe Docker is a virtualization engine – but it's far from it as we will explain in a while.

Containerization vs Virtualization

To fully understand what Docker really is, first we need to understand the difference between traditional virtualization and containerization. Let's compare those two technologies now.

Traditional virtualization

A traditional virtual machine, which represents the hardware-level virtualization, is basically a complete operating system running on top of the host operating system. There can be a lot of use cases that would make an advantage from using virtualization – the biggest asset is that you can run many virtual machines with totally different operating systems on a single host. Virtual machines are fully isolated, hence very secure. But nothing comes without a price – there are many drawbacks: they contain all the features that operating system needs to have: device drivers, core system libraries and so on. They are heavyweight, usually resource-hungry and not so easy to setup – virtual machines require full installation. They require more computing resources to execute – to successfully run an application on a virtual machine, the hypervisor needs to first import the virtual machine and then power it up – and this takes time. Furthermore their performance gets substantially degraded. As a result, only a few virtual machines can be provisioned and made available to work on a single machine.

Containerization

Docker software runs in an isolated environment called a Docker container. A Docker container is not a virtual machine in the popular sense. It represents operating system virtualization. While each virtual machine image runs on an independent guest OS, the Docker images run within the same operating system kernel. A container has its own filesystem and environment variables. It's self-sufficient. Because of the containers run within the same kernel, they utilize fewer system resources – the base container can be, and usually is – very lightweight. Worth knowing is that Docker containers are isolated not only from the underlying OS, but from each other as well. When it comes to the performance, all the unneeded operating system core software is removed from the Docker image. There is no overhead related to a classic virtualization hypervisor and a guest operating system. This allows achieving almost bare metal, core performance. The boot time of a “dockerized” application is usually very fast due to the low overhead of container. It is also possible to speed up the roll-out of hundreds of application containers in seconds and to reduce the time taken for provisioning your software.

As you can see, 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 one hand we have the fully isolated, secure virtual machine with average performance and 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 swiftly. Let's see what other benefits you will get when using Docker containerization.

Benefits of using Docker

When comparing the Docker containers with traditional virtual machines, we have mentioned some of its advantages. Let's summarize them now in more detail and add some more.

Speed and size

As we have said before, the first visible benefit of using Docker will be very satisfactory performance and short provisioning time. You can create or destroy containers quickly and easily. Containers share resources, like the operating system's kernel and 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 nimble boot times.

Reproducible and portable builds

Using Docker containers enables you to deploy ready-to-run software, which is portable and extremely easy to distribute (we will cover the process of creating an image in Chapter 6 – "Building images"). Your containerized application simply run within its container, there's no need for installation. The lack of installation process has a huge advantage – this eliminates problems such as software and library conflicts or even a driver compatibility issues. Because of Docker's reproducible build environment, it's particularly well suited for testing, especially in your Continuous Integration 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.

One of Docker's greatest features is the movability. Docker containers are portable – they can be run from anywhere: your local machine, a nearby or distant server, and private or public cloud. When speaking about the cloud, all major cloud computing providers, like Amazon Web Services, Google's Compute Platform have perceived Docker's availability and now support it. Docker containers can be run inside an Amazon EC2 instance, Google Compute Engine instance provided that the host operating system supports Docker. A container running on an Amazon EC2 instance can easily be transferred to some other environment, achieving the same consistency and functionality. Docker works very well with various other IaaS (Infrastructure-as-a-service) providers like Microsoft's Azure or OpenStack. This additional level of abstraction from your infrastructure layer is an indispensable feature. You can just develop your software without worrying about the platform it will be run later on. It's truly "write once run everywhere" solution.

Immutable and agile infrastructure

The maintaining a truly idempotent configuration management code base can be tricky and time consuming process. 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. Containerization comes to the rescue. By using containers during the process of development and deployment of your applications, you can simplify the process. Having a lightweight Docker server that needs almost no configuration management, you manage your applications simply by deploying and redeploying containers to the server. And again, because the containers are very lightweight, it takes only seconds of time.

As a starting point, you can download a pre-built Docker images from the Docker Hub, which is like a repository of ready-to-use images. There are many choices of web servers, runtime platforms, databases, messaging servers and so on. It's like a real gold mine of software you can use for free to get a base foundation for your own project. We will cover the Docker Hub and looking for images in the Chapter 5 – "Finding images".

The effect of the immutability of Docker's images is the result of a way they are being created. Docker makes use of a special file called a "Dockerfile". This file contains all the setup instructions how to create an image, like must-have components, libraries, exposed shared directories, network configuration and so on. An image can be very basic, containing nothing but the operating system foundations, or – which is a more common thing – containing a whole pre-built technology stack which is ready to launch. You can create images by hand, but it can be an automated process also. Docker creates images in a layered fashion: every feature you include will be added as another layer in the base image. This is another serious speed boost, comparing to the traditional virtualization techniques. We will get into details of creating images later, in the Chapter 6 – "Building images.

Tools and APIs

Of course, Docker is not just a Dockerfile processor and the runtime engine. It's a complete package with wide selection of tools and APIs that are helpful during the developer's and DevOp's daily work. First of all, there's a The Docker Toolbox – an installer to quickly and easily install and setup a Docker environment on your own machine. The Kinematic is desktop developer environment for using Docker on Windows and Mac OS X. Docker distribution contains also a whole bunch of command line tools we will be using through the whole book.

From a developer's perspective, there are tools especially useful in a programmer's daily job, let it be IntelliJ IDEA Docker Integration Plugin for Java fans or Visual Studio 2015 Tools for Docker for those who prefer C#. Those let you download and build Docker images, create and start containers, and carry out other related tasks straight from your favorite IDE. We will cover them in more details in the next chapters.

Apart from that tools included in the Docker's distribution package, there are hundreds third-party tools, like Kubernetes and Helios (for Docker orchestration), Prometheus (for monitoring of statistics) or Swarm and Shipyard for managing clusters. As Docker captures higher attention, more and more Docker-related tools pop-up almost every week. We will try to briefly cover the most interesting ones in the last chapter, Chapter 9 – "More resources".

But there are not only the tools available for you. Additionally, Docker provides a set of APIs that can be very handy. One of them is the Remote API for the management of the images and containers. Using this API you will be able to distribute your images to the runtime Docker engine. The container can be shifted to a different machine that runs Docker, and executed there without compatibility concerns. This may be especially useful when creating PaaS (platform-as-a-service) architectures. There's also the Stats API that will expose live resource usage information (such as CPU, memory, network IO and block IO) for your containers. This API endpoint can be used create tools that show how you containers behave, for example, on a production system.

By now we understand the difference between the virtualization and containerization and also – I hope – can see the advantages of using the latter. Let's begin our journey to the world of containers and go straight to the action – by installing the software.

Summary

2
Installing Docker

In this chapter we will find out how to install Docker on Windows, Mac OS and Linux operating systems. There will be also step-by-step instruction how to setup Docker in the Cloud – and Amazon EC2 will be used as an example. Next, we will run sample "hello-world" image to verify the setup and check if everything works fine after the installation process.

Docker installation is quite straightforward, but there are some things you will need to focus on to make it running. We will point them out to make the installation process painless.

It's worth mentioning that Linux is the natural environment for Docker. The Docker engine is built on top of the Linux kernel. To make it running under Windows or Mac OS, the Linux kernel needs to be virtualized.

The Docker engine could be run on the Mac and MS Windows operating systems by using the lightweight Linux distribution, made specifically to run Docker containers. It runs completely from RAM, weights just several dozens of megabytes and boots in couple of seconds. During the installation of the main Docker package – the Docker Toolbox, also the virtualization engine VirtualBox will be installed by default. Therefore, there are some special hardware requirements for your machine.

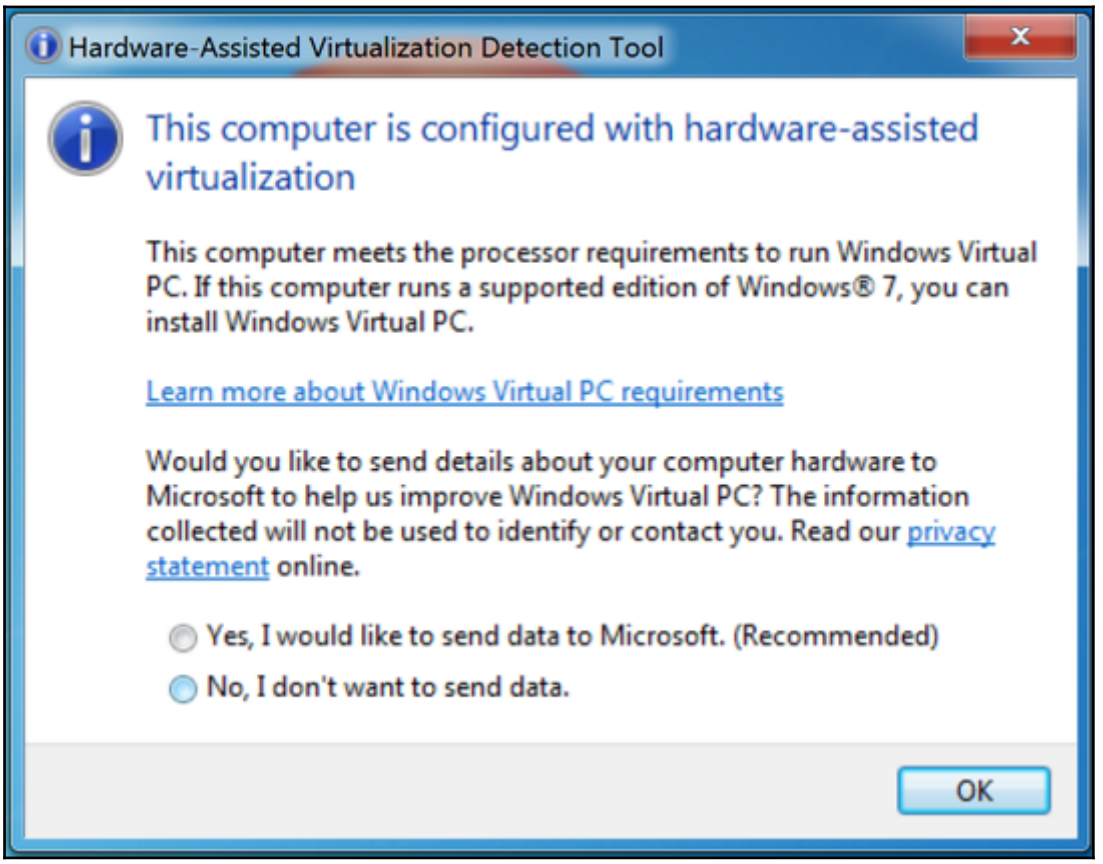
Hardware requirements

To use Docker, you will need some reasonably new machine, which supports hardware-level virtualization – it will be VT-x for Intel-based PC and AMD-V for AMD processors. Most of the Mac machines support it out of the box, but for PC you will need to make sure it's turned on and perhaps enable it in the BIOS settings – it will be different for different

BIOSes, just look for VT-x / AMD-V switch.

In Windows 8, you can check virtualization support in the task manager in the "Performance" tab:

If you want to check if your PC supports the hardware level virtualization in Windows 7, look for the "Microsoft Hardware-Assisted Virtualization Detection Tool". It's free, tiny utility to check if your system supports virtualization. Download and run it, to see the report:



If the report is different for you, saying that the hardware-assisted virtualization is not enabled, you will need to check the BIOS settings on your machine – maybe the hardware-assisted virtualization support is just switched off. In such case switch it on and re-run the tool again.

If your PC doesn't support hardware-assisted virtualization and you decide to install Docker anyway, it will result in an error during the start of the virtualized Linux distribution.

Additionally, when installing on Windows PC, you need to make sure your Windows OS is 64-bit (x64). Docker will not run on the 32-bit system by default.

Knowing the hardware requirements, we need to know what Docker components are available to install.

Tools overview

The installation package for Windows and Mac OS is wrapped in an executable called the Docker Toolbox. The package contains all the tools you need to begin working with Docker. Of course there are tons of additional third party utilities compatible with Docker, some of them very useful. We will present some of them briefly in the Chapter 9 – Appendix and More Resources. But for now, let's focus on the default toolset. Before we start the installation, let's look at the tools that the installer package contains to better understand what changes will be made to your system.

Docker Engine and Docker Engine client

Docker is a client-server application. It consists of the daemon that does the important job: builds and downloads images, starts and stops containers and so on. It exposes a REST API that specifies interfaces for interacting with the daemon and is being used for remote management. Docker Engine accepts Docker commands from the command line, such as docker run to run the image, docker ps to list running containers, docker images to list images, and so on.

The Docker client is a command line program that is being used to manage Docker hosts running Linux containers. It communicates with the Docker server using the REST API wrapper. You will interact with Docker by using the client to send commands to the server.

Docker Engine works only on Linux. If you want run Docker on Windows or Mac OS, or want to provision multiple Docker hosts on a network or in the Cloud, you will the need the Docker Machine.

Docker machine

Docker-machine is a fairly new command line tool created by Docker team to manage Docker servers you can deploy containers to. It deprecated the old way of installing Docker with “Boot2Docker” utility. The Docker Machine eliminates the need to create virtual machines manually and install Docker before starting Docker containers on them. It handles the provisioning and installation process for you behind the scenes. In other words, it's a quick way to get a new virtual machine provisioned and ready to run Docker containers. This is an indispensable tool when developing Platform as a Service (PaaS) architecture. Docker Machine not only creates a new VM with the Docker engine installed in it, but sets up the certificate files for authentication and then configures the Docker client to talk to it. For the flexibility purposes, the Docker Machine introduces the concept of “drivers”. Using drivers, Docker is able to communicate with various virtualization software and cloud

providers. In fact, when you install Docker for Windows or Mac OS, the default VirtualBox driver will be used. The following command will be executed behind the scenes:

```
docker-machine create --driver=virtualbox default
```

Another available driver is amazonec2 for Amazon Web Services. It can be used to install Docker on the Amazon's cloud – we will do it later in this chapter. There are a lot of drivers ready to be used, and more are coming all the time. The list of existing official drivers with their documentation is always available at the Docker Drivers website: https://docs.docker.com/machine/drivers. The list contains the following drivers at the moment:

- Amazon Web Services
- Microsoft Azure
- Digital Ocean
- Exoscale
- Google Compute Engine
- Generic
- Microsoft Hyper-V
- OpenStack
- Rackspace
- IBM Softlayer
- Oracle VirtualBox
- VMware vCloud Air
- VMware Fusion
- VMware vSphere

Apart from these, there is also a lot of 3rd-party driver plugins available freely on the Internet sites like GitHub. You can find additional drivers for different cloud providers and virtualization platforms, like OVH Cloud or Parallels for Mac OS for example – you are not limited to Amazon's AWS or Oracle's VirtualBox. As you can see, the choice is very broad.

If you cannot find a specific driver for your Cloud provider, try looking for it on the GitHub.

When installing the Docker Toolbox on Windows or Mac OS, Docker Machine will be selected by default – it's mandatory and currently the only way to run Docker on these operating systems. Installing the Docker Machine is not obligatory for Linux – there is no need to virtualize the Linux kernel there. However, if you want to deal with the Cloud

providers or just want to have common runtime environment portable between Mac OS, Windows and Linux, you can install Docker Machine for Linux as well. We will describe the process later in this chapter. . Machine will be also used behind the scenes when using the graphical tool – Kitematic, which we will present in a while.

After the installation process, Docker Machine will be available as a command line tool: docker-machine.

Kitematic

Kitematic is the software tool you can use to run containers through a plain, yet robust graphical user interface. In 2015 Docker has acquired Kitematic team, expecting to attract many more developers and hoping to open up the containerization solution to more developers and a wider, more general public.

Kitematic is now included by default when installing Docker Toolbox on Mac OS and MS Windows. You can use it to comfortably search and fetch images you need from the Docker Hub. The tool can be also used to run your own app containers. Using the GUI you can edit environment variables, map ports, configure volumes, study logs and have command line access to the containers. Worth mentioning is that you can seamlessly switch between Kitematic GUI and command line interface to run and manage application containers. Kitematic is very convenient, however, if you want to have more control when dealing with the containers or just want to use scripting – the command line will be the better solution. In fact, Kitematic allows you to switch back and forth between the Docker CLI and the graphical interface. Any changes you make on the command line interface will be directly reflected in Kitematic. The tool is simple to use, as you will see at the end of this chapter, when we are going to test our setup on Mac or Windows PC. For the rest of the book we will be using the command line interface for working with Docker.

Docker Compose

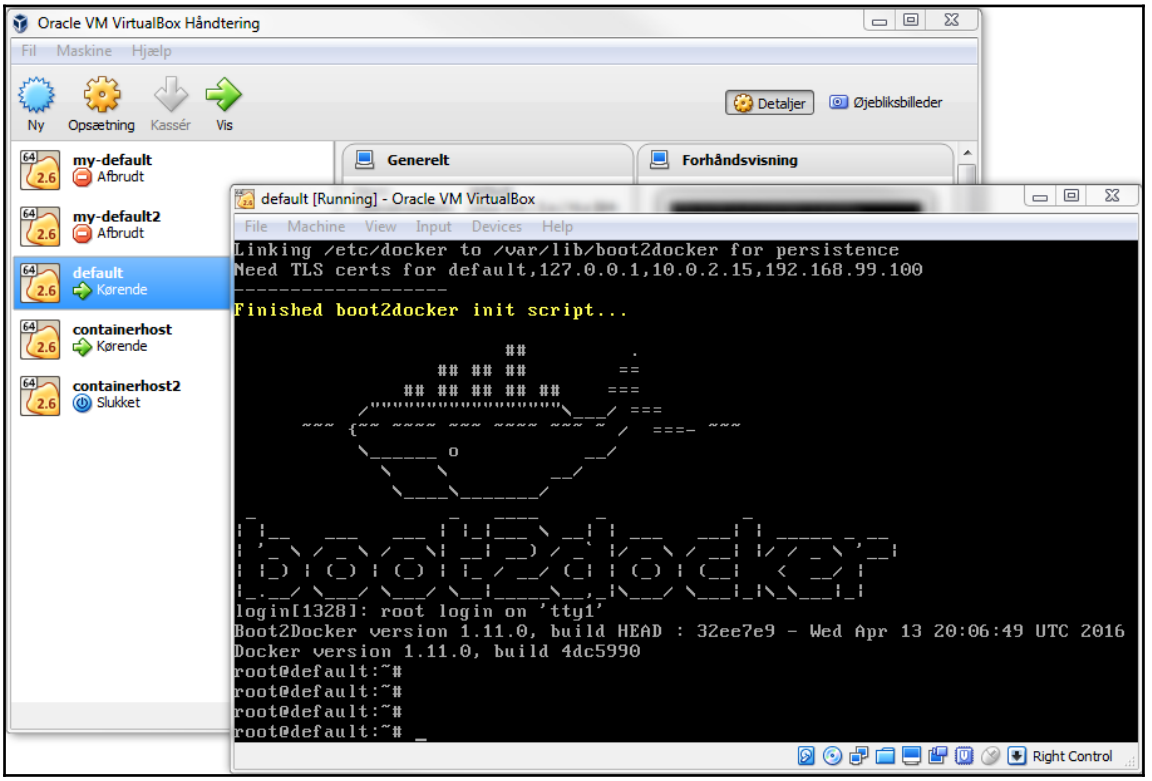
Compose is a tool, executed from the command line as `docker-compose`. It replaces the old "fig" utility. It's used for defining and running multi-container Docker applications. Although it's very easy to imagine a multi-container application (like a web server in one container and a database in the other) – it's not mandatory. So if you decide that your application will fit in a single Docker container, there will be no use for docker-compose. In real life, it's very likely that your application will span into multiple containers. With docker-compose, you use a "compose file" to configure your application's services, so they can be run together in an isolated environment. Then – using a single command – you create and start all the services from your configuration. When it comes to multi-container

applications, docker-compose is great for development and testing, as well as continuous integration workflows.

We will use docker-compose to create multi-container applications in the Chapter 6: Building images later in this book.

Oracle VirtualBox

Oracle VM VirtualBox is a free and open-source hypervisor for x86 computers from Oracle. It will be installed by default when installing the Docker Toolbox. It supports the creation and management of virtual machines running Windows, Linux, BSD, OS/2, Solaris and so on. In our case – the docker-machine, using Virtual Box driver, will use VirtualBox to create and boot a bitsy Linux distribution capable of running docker-engine. It's worth mentioning, that you can also run the teensy-weensy virtualized Linux straight from the VirtualBox itself. Every Docker machine you have created using the docker-machine or Kitematic, will be visible and available to boot in the VirtualBox, when you run it directly, as seen of the following screenshot:



You can start, stop, reset, change settings and read logs in the same way as for other virtualized operating system.

You can use VirtualBox in Windows or Mac for other purposes than Docker.

Git

Git is a distributed version control system that is widely used for software development and other version control tasks. It has emphasis on speed, data integrity, and support for distributed, non-linear workflows. Docker Machine and Docker client uses Git internally for fetching the needed dependencies from the network. For example, if you decide to run the Docker image which is not present on your local machine, Docker will use Git to fetch this

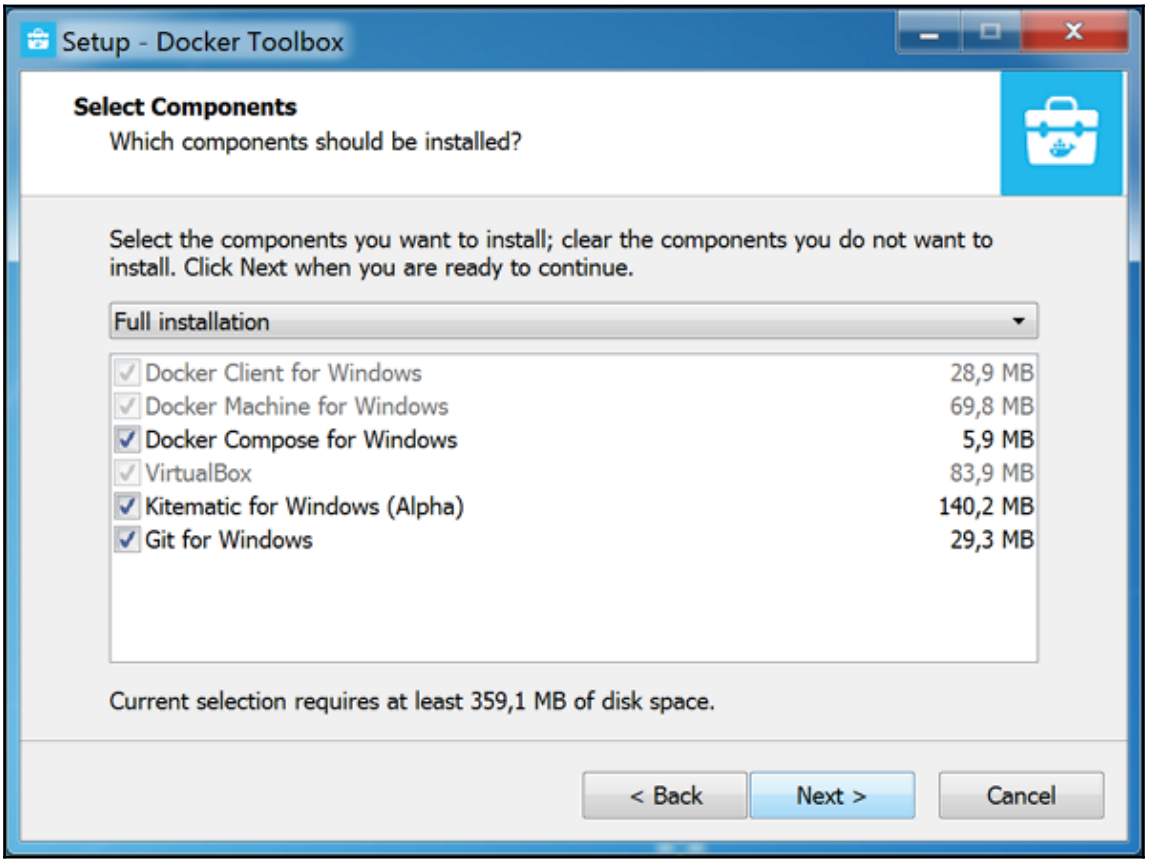
image from the Docker Hub. Therefore it's mandatory to have Git up and running on your system. Git is included in the Docker Toolbox installation package.

Now we have an idea which components are included in the default installation package. Let's download it and install the software. Docker up-to-date installation guides are always available on the https://www.docker.com/ website. Head to the **Get started** section and the pick the installation guide according to your operating system: Windows, Mac OS or Linux.

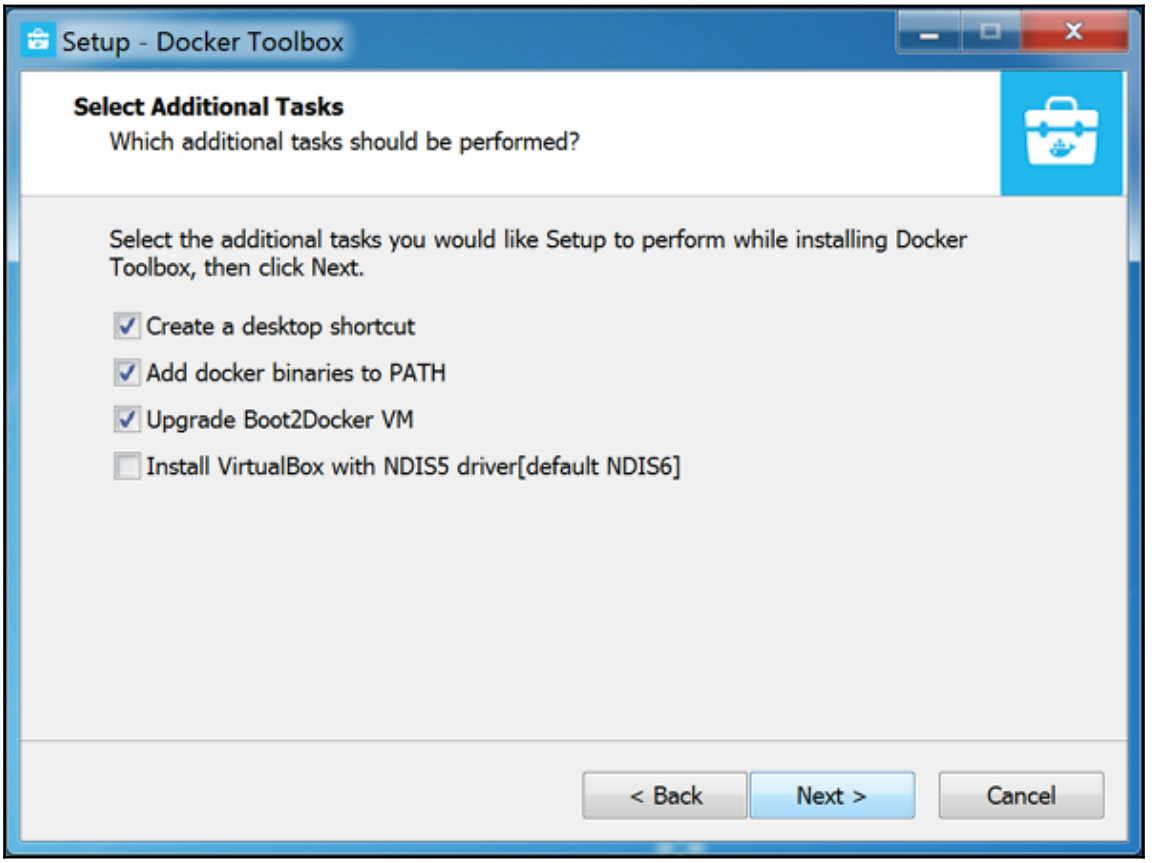
The Toolbox packages for Windows and Mac OS are available on the https://www.docker.com/products/docker-toolbox website.

Installing on Windows

After downloading the Docker Toolbox package, run it. The first screen will ask you about permission to provide anonymous usage statistics, to help Docker developers improve their software. It's up to you if you allow it or not – depending on your privacy concerns. The next screen presents the components available for installation:



Docker Client and Docker Machine are mandatory – you will be able to do nothing without them. For the first time setup it's better to leave all of the options checked. Kitematic tool is in alpha version in the time of writing this book, but don't worry – it behaves just fine. The last installation screen will ask if you would like to add Docker's binaries to path or create the desktop shortcuts, as seen on the following screenshot:

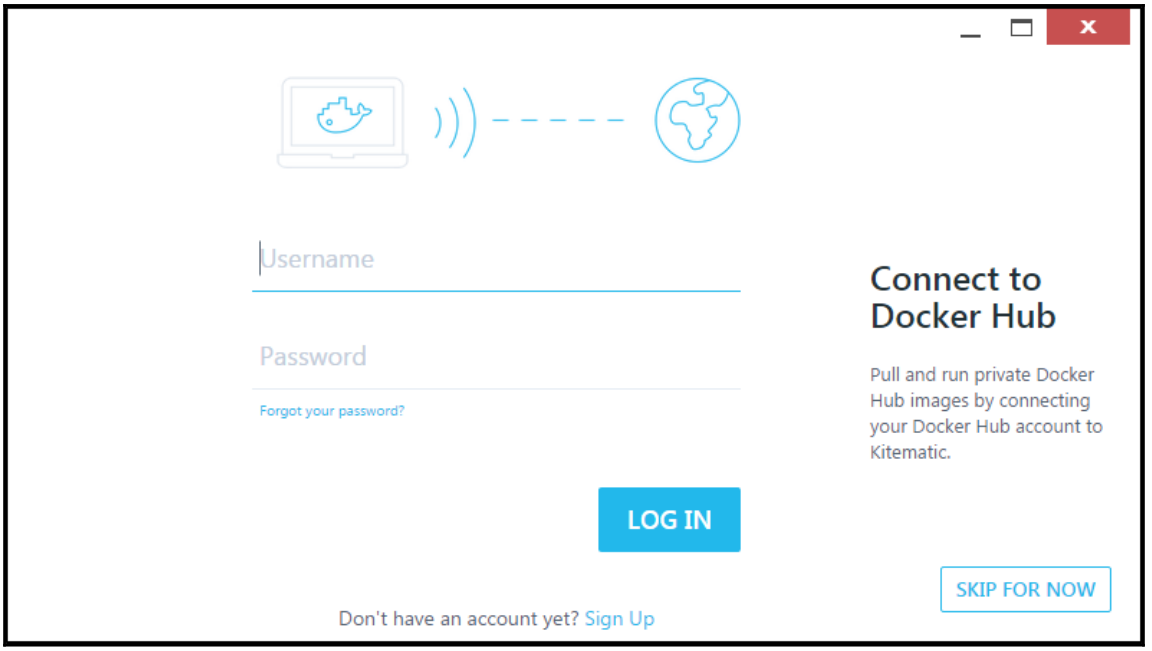


Having Docker command line tools on path will be very handy in the future, so it's better to have this option marked. It's worth mentioning that you have a choice of installing the older NDIS5 host network filter driver instead of the default one, – NDIS6. If you happen to use an older version of Windows – older than Windows Vista – this may help with some problems like slowdowns or network issues. If you choose the NDIS6 and notice problems later, just execute the installer again and then pick the NDIS5 instead.

When experiencing slowdowns or network issues, try the NDIS5 driver instead.

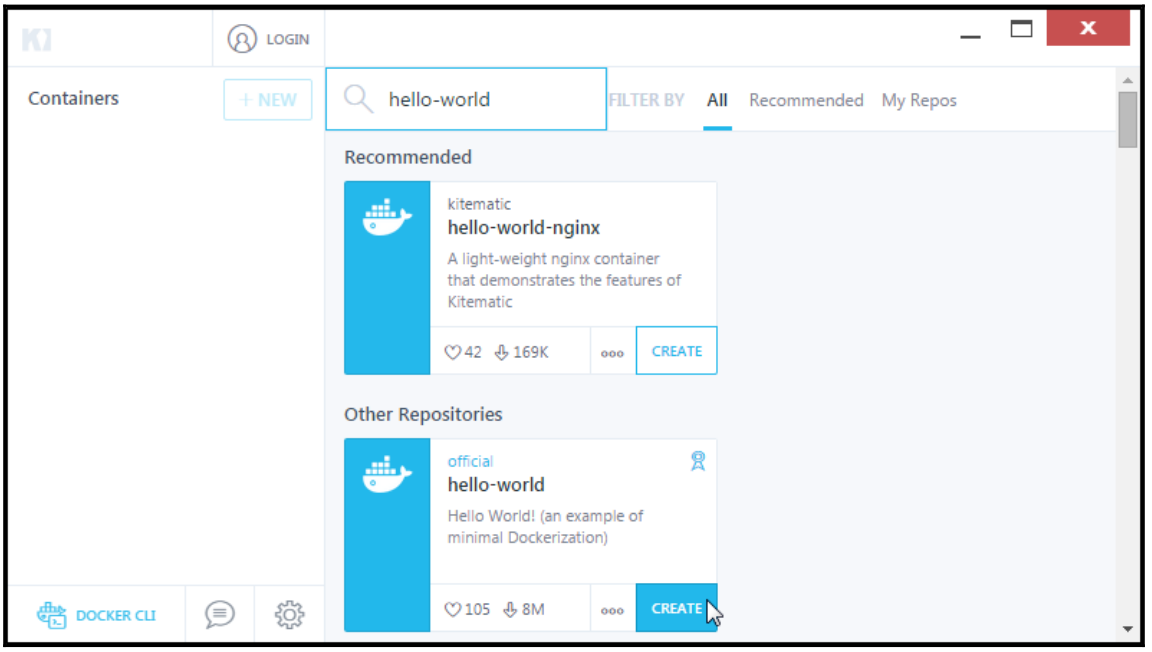
After the installation, run the Kitematic tool we described earlier. It will present the progress window during starting the Docker Linux VM. After successful start, Kitematic

will ask you for your Docker Hub credentials. You may now login into the Docker Hub now, create the Docker Hub account, or skip this process at this time – it's not mandatory for searching and running images. We will use create the account and will be using Docker Hub heavily in the Chapter 5 – Finding Images:

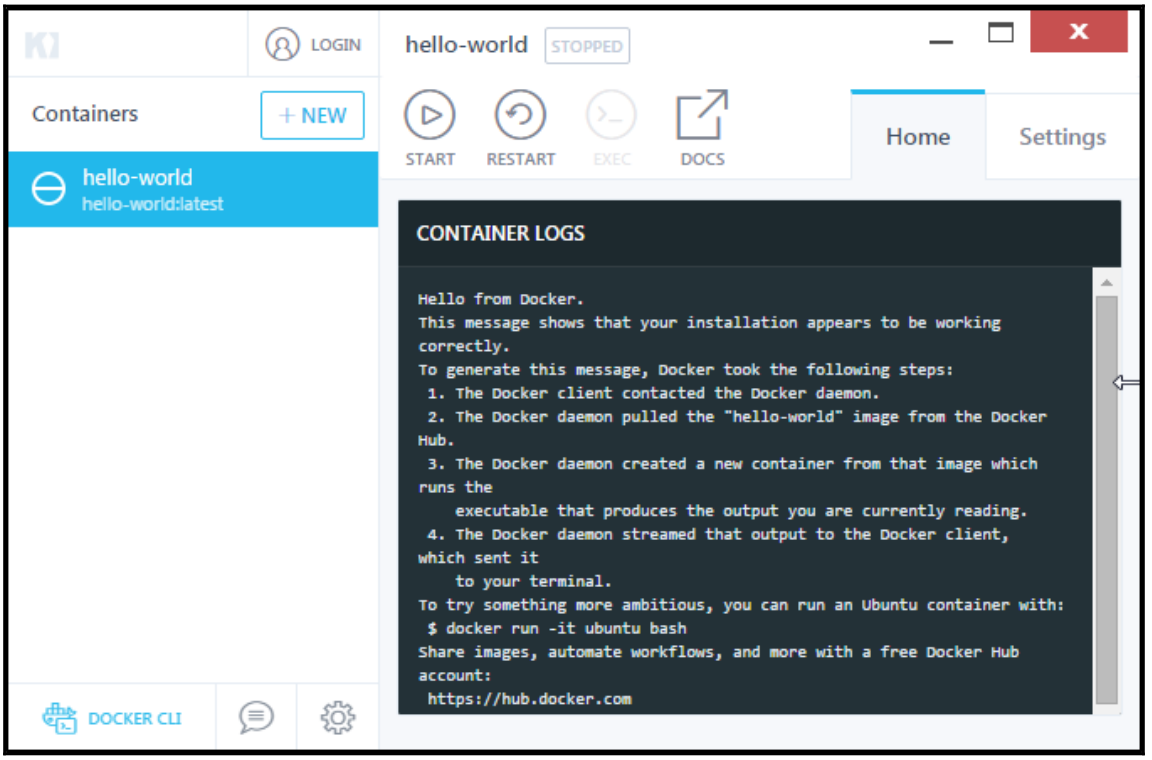


No matter if you decide to do or skip the login process, you will be presented with the main window of Kitematic. Let's discuss it briefly. On the left side there is a list of containers available to run – empty if you run Kitematic for the first time. Let's find a simple image to run. Just start typing **hello-world** in the search box, to list Docker Hub images containing such phrase in the name. Experiment with different searches to see what's available in the Hub. You can download and run databases, like MySQL or Mongo, web servers like nginx, and many, many more. All in couple of mouse clicks! And best of all – no dependencies are required to be installed on your machine. For example, you don't even need to have Java installed on your machine to be able to run Jboss or Tomcat. This is the magic of containerization – and this is just the beginning.

One of the images from the official repository is famous **hello-world** image. Click **Create** next to it:



Now the magic happens. Docker will fetch the selected image from the Hub and execute it. From now on, you can use Kitematic to start, stop, restart and configure your container:



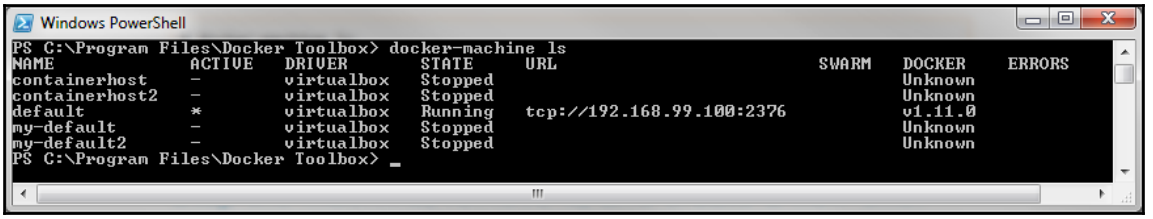
As you can see in the **Container Logs** section, a lot has happened behind the scenes just to print the **Hello from Docker** message. But of course, it's not the usual message – it comes from a containerized software – and it makes a huge difference comparing to standard "hello world" programs. First, it checks if the Docker daemon is running on your system and connects to it if so. Next, it looks if the **hello-world** image is present on your local systems. If not – and it will be your case when running for the first time – it fetches the image from the Docker Hub. Docker runs the image and streams its output back to you to see.

Let's do the same from the Docker CLI. Click the **Docker CLI** icon in the Kitematic, to execute the Windows PowerShell command prompt. You should not worry about the command line syntax at the moment – we will explain it in the next chapters. Also, we will be using command line tools through the whole book, so you will easily get familiar with the syntax.

At first, let's verify if the virtualized Linux machine is working properly, by executing the command:

```
docker-machine ls
```

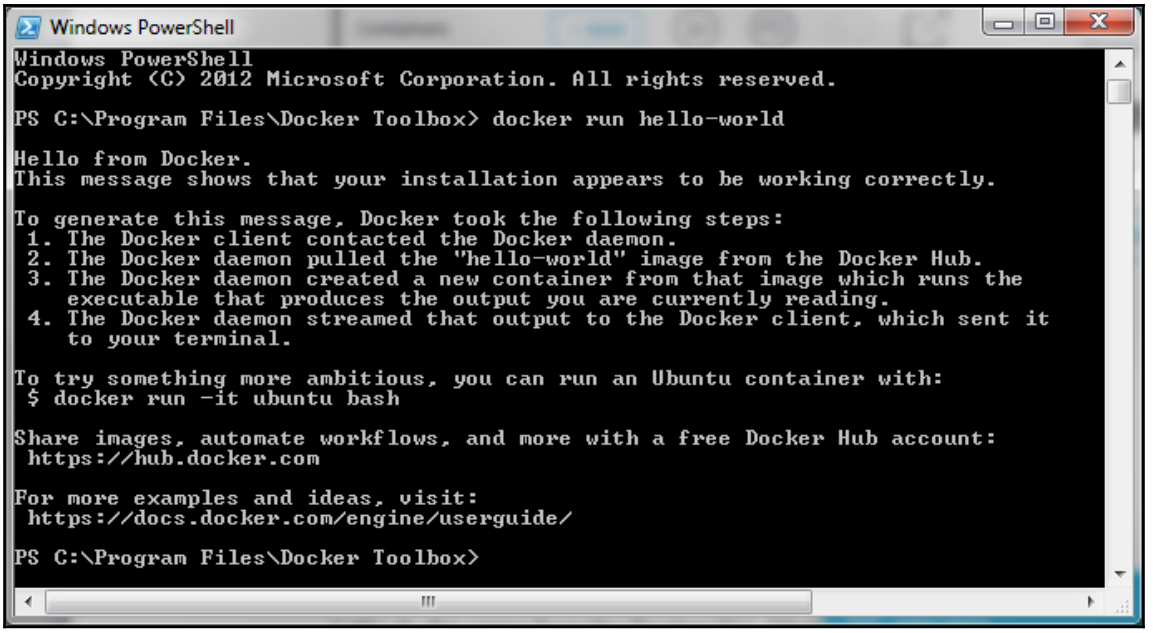
`ls` stands for `list` command and lists the virtual machines configured on your Windows. The currently running machine will have a star in the "ACTIVE" column and status **Running** in the **STATUS** column:



Next, execute the following command to run the image:

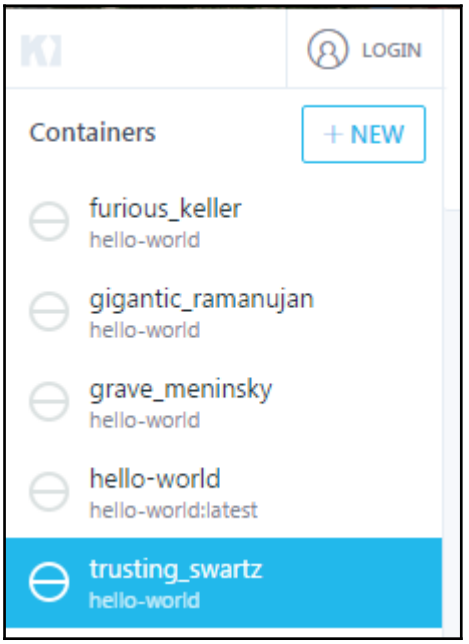
```
docker run hello-world
```

This will give you the same exact output as in Kitematic:



As we have said before, the Docker CLI and Kitematic are seamlessly connected and everything you do in the command line will be reflected in Kitematic. You can notice that

the image you just have run shows up in the GUI of Kitematic. Worth noticing is when you create a new Docker container and don't give it a custom name (by passing "-name" option with the Docker CLI) Docker generates a name for you:



These auto-generated names may be entertaining, but they can also be very useful later, when you will need to distinguish the container by name. It's easier to remember furious_keller than container231 isn't it?

Docker will automatically generate a container's name if you forget to do so.

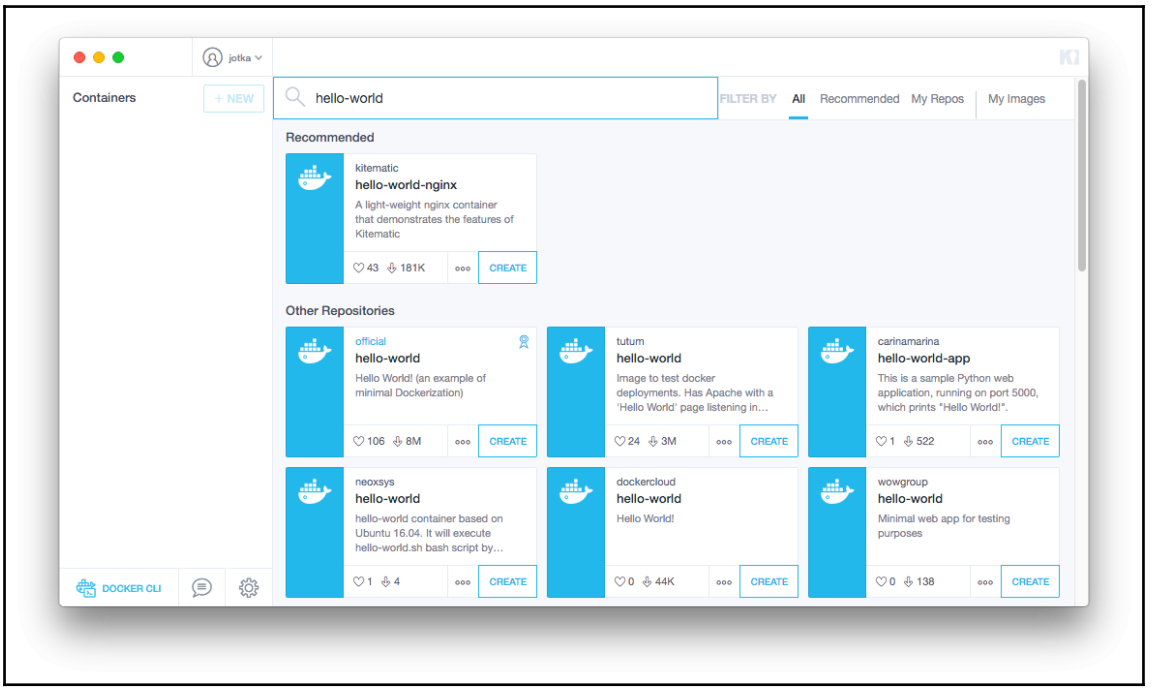
Installing on Mac OS

Docker installation on Mac OS is very similar to the installation on Windows PC. Again, we have complete Docker Toolbox package, containing all the tools to get you started. It contains Docker Client, Machine, Compose, Kinematic and VirtualBox. Your Mac must be running OS X 10.8 "Mountain Lion" or newer to run Docker software. Head straight to the

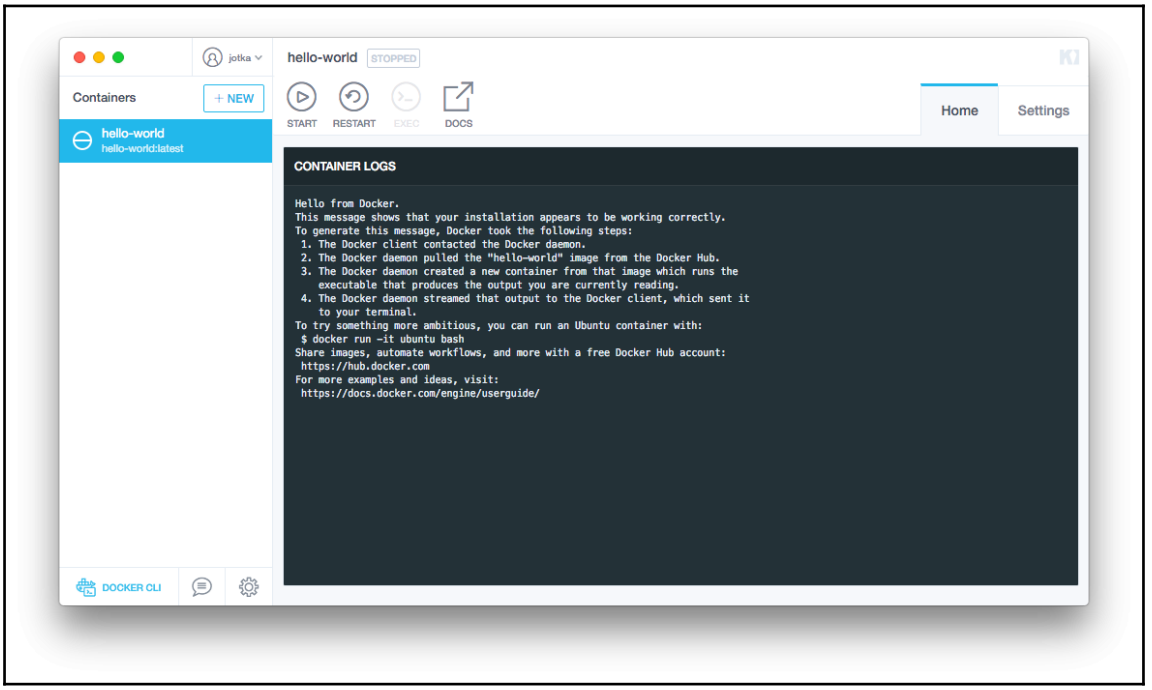
Docker Toolbox website https://www.docker.com/products/docker-toolbox and download Mac OS version. Docker Toolbox for Mac is wrapped into a pkg package, so you need to run it, instead of just moving it into the Applications folder. Similar to Windows version, it will as if you would like to give it a permission to report usage statistics to improve the future releases. After the installation, the Quick Start page will give you a choice to quickly execute the Terminal or the Kitematic tool:

You will be also able to find Docker tools and VirtualBox in the **Applications** folder and **Launchpad** menu later:

To verify the setup, execute **Kitematic**, skip **Docker Hub login** and type **hello-world** in the search field:



Next, click **Create** button next to the **hello-world** image. Kitematic (docker-machine to be precise) will download the image from the Docker Hub and present you the output:



From now on, also Docker's command-line tools are available for your disposal. Opening the Terminal from Kitematic will automatically set all needed environment variables and connect you to the default machine. But if you would like to be able to work from your own terminal session, you may get a message saying that Docker client cannot connect to the Docker daemon. The reason is that client doesn't know what machine it should control. To attach your client to the specific machine, type the following command in your Terminal session:

```
eval "$(docker-machine env default)"
```

The eval command sets environment variables to dictate that docker should run a command against a particular machine, which is “default” machine in our example.

To see the list of Docker related environment variables, list the using env command:

```
env | grep DOCKER
```

The output will contain Docker variables, like machine name, host and certificate path:

```
DOCKER_TLS_VERIFY=1
DOCKER_HOST=tcp://192.168.99.100:2376
```

```
DOCKER_CERT_PATH=/Users/jarek/.docker/machine/machines/default
DOCKER_MACHINE_NAME=default
```

Always check environment variables when having Docker daemon connection issues.

Another way of testing the setup is the docker info command. Execute the following from the command line shell:

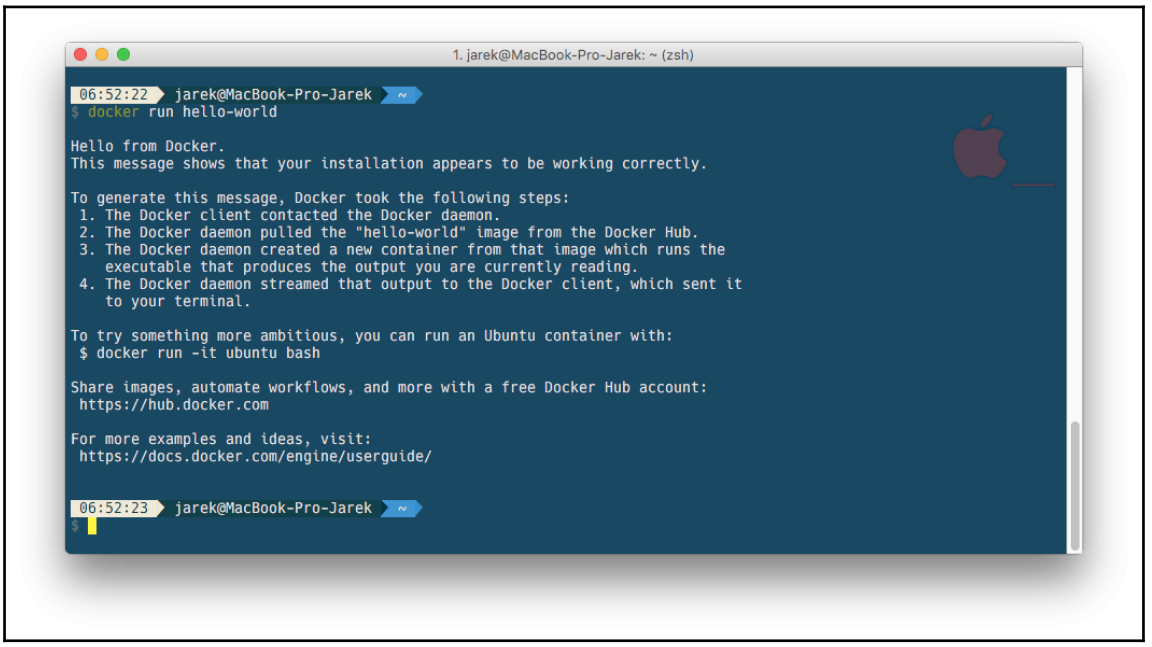
```
docker info
```

The output will contain a lot of useful information about the server's kernel version, memory available, the number of running containers, the name of the machine and so on.

Let's now type the following command to run the **hello-world** image straight in the Terminal:

```
docker run hello-world
```

If you can see the output from the image, you are all set and have a working Docker setup on your Mac:



Installing on Linux

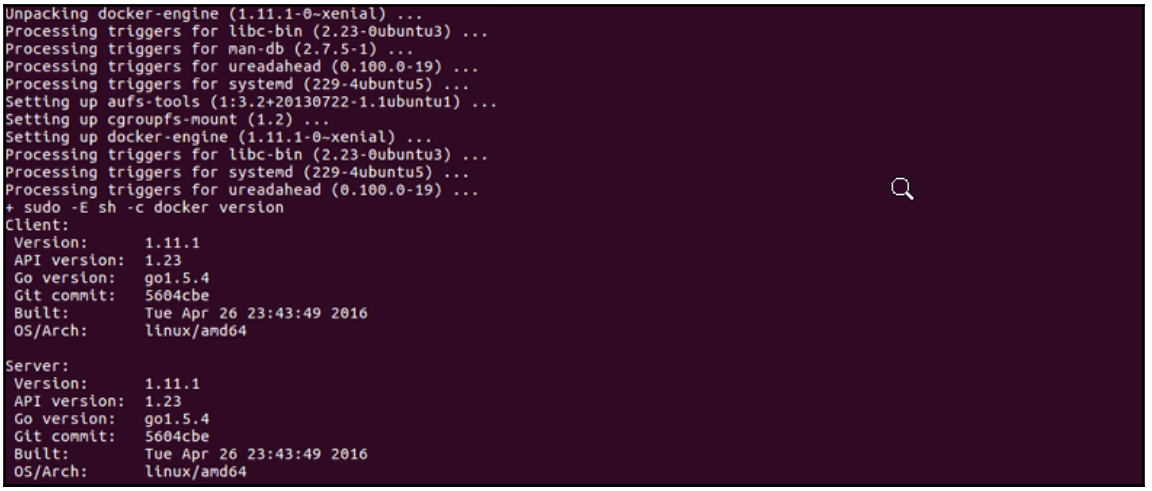
As we have said before, Linux is natural habitat for Docker. Therefore there is no need to virtualize the Linux kernel. You can run Docker without the need of VT-x or similar technologies in your processor, since Docker only requires cgroups to be available on kernel to get the majority of its features working. Cgroups (control groups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes. Docker will simply use the kernel of your own operating system. This also makes the installation package smaller – there is no need for virtualization engine and another virtualized operating system. This is the reason that the installation process is a little bit different than on Mac OS or Windows. First, there is no Docker Machine included in the installation – it's simply not mandatory for Linux. Second, there is no fancy GUI installer – you will need to most of the tasks from the command line, but this should not be a problem for a Linux user. And last but not least, there is no Kitematic tool available for Linux.

On the Docker website you can find installation steps for the specific Linux distribution (this will be yum package manager for Red Hat or apt-get for Ubuntu, for example). If you are not willing to use the package manager directly, you can use the installation script

provided by the Docker team. In fact, the script will execute the package manager valid for your Linux OS and then install the software using packages. To get the most recent Docker release for Linux, type the following in the shell:

```
curl -fsSL https://get.docker.com/ | sh
```

The process of downloading required package will begin and you will observe keys, packages and their dependencies being downloaded. At the end of the output, the installer will print out the version of the just installed Docker client and server:



Installing this way will make Docker service available for root to run. It's not always a good idea to run software as a root, so you will probably want to make it runnable also for your user. First, you will need to create the "docker" group:

```
sudo groupadd docker
```

and then add your current user to the group (assuming that `yourUsername` is the login name for your user:

```
sudo usermod -aG docker yourUsername
```

If you are going to deal with the cloud setup using your Linux, you will want to install Docker Machine as well. To do this, execute this script:

```
    $ curl -L
https://github.com/docker/machine/releases/download/v0.7.0/docker-machine-`
uname -s`-`uname -m` > /usr/local/bin/docker-machine
    chmod +x /usr/local/bin/docker-machine
```

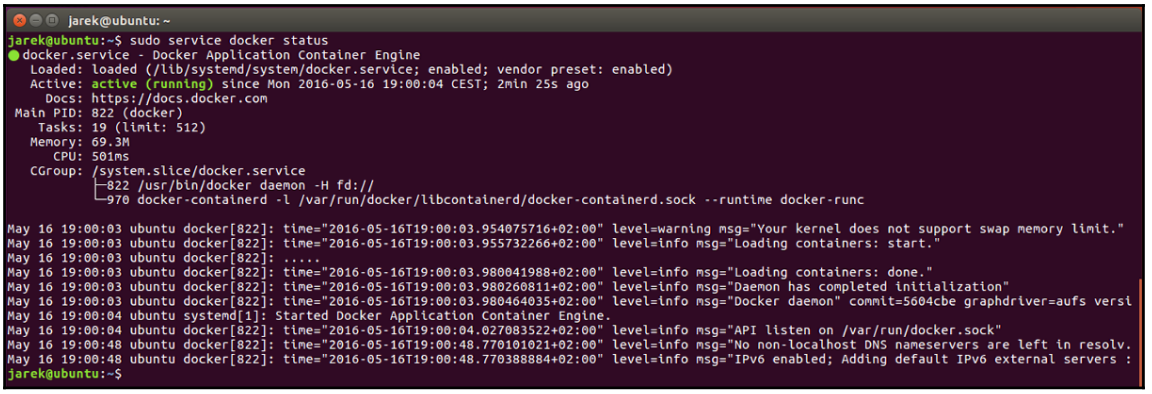
You can also go to the Linux releases page directly and pick your desired version here:

https://github.com/docker/machine/releases/

To test the installation, let's print out the Docker service status, by executing the script (this is an example for the latest Ubuntu Linux, which I will be using for the rest of the book):

```
sudo service docker status
```

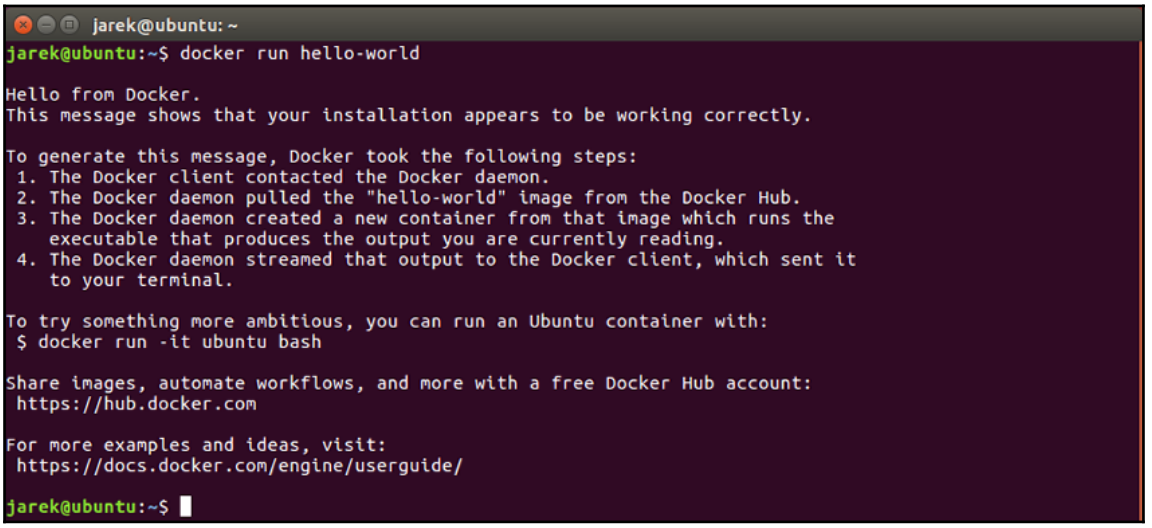
If the service responds properly, it will show the running status and also some statistics, like memory available for the service and some recent log lines:



The service seems to be working fine, so the next thing we are going to do will be running the simple **hello-world** image. This is where the fun begins – if Docker will not be able to find the image on your local machine, it will fetch if from the Docker Hub and the run it. We will talk a lot more about finding images and the Docker Hub later, in the Chapter 5 – Finding images. To execute the sample **hello-world** image, type the following command in your shell after logging out and logging again:

```
docker run hello-world
```

After executing the command, the correct result will be just a **Hello from Docker** message along with some more interesting facts:

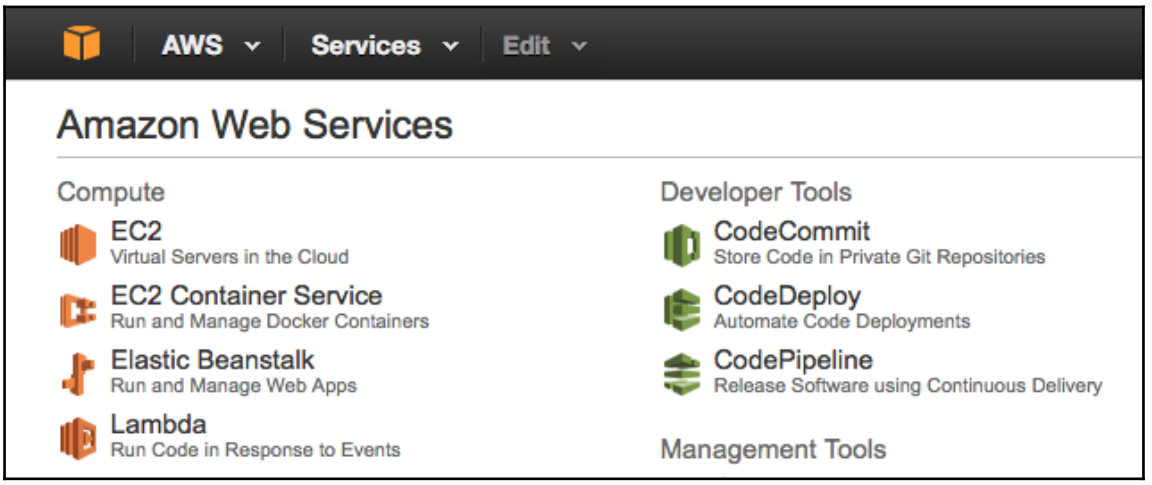


This is the same output you will get when running the image using Kitematic for Windows or Mac OS. Docker prints out steps that it needed to do to run the image. Again, a lot of has happened in the background just to print the simple message, but running the sample image is a great way of testing the setup.

Installing on the cloud – Amazon AWS

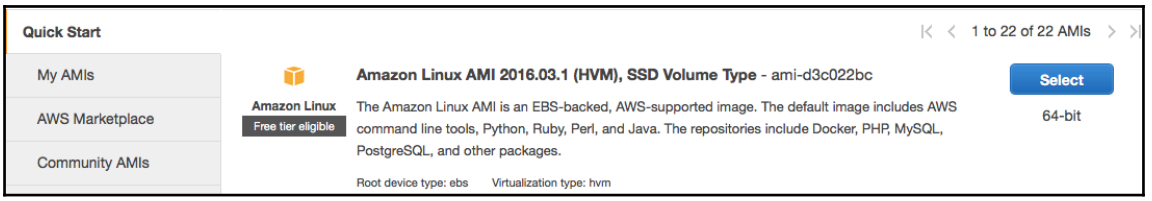
In the next chapters we will show how to create Docker instance remotely, using the Docker Machine. This time we will just install Docker and run the sample image on the Amazon EC2 Cloud manually. If you have a running Linux on the EC2 Cloud, Docker installation procedure is almost identical as for any Linux system. Let's begin with creating Linux instance first.

For using EC2 Cloud you will need to create an account. It's free for basic purposes, so go ahead to http://aws.amazon.comand fill out the registration form. Also, the basic “t2.micro” instance is free for you to use and enough for testing Docker installation. After creating the account, login into the AWS Console and select EC2 from the list of available services:



Next, launch the instance using the **Launch Instance** button:

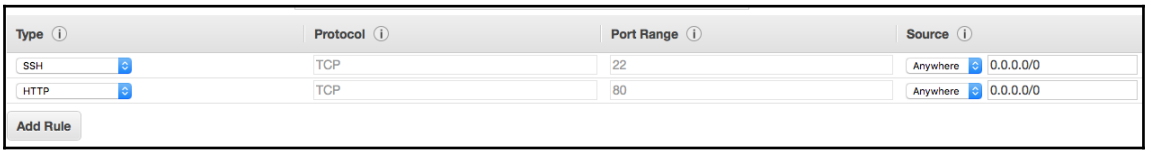
The next page asks what operating system should be available on your new EC2 instance. For our purposes, Amazon Linux will be fine. It's first on the list, select it:



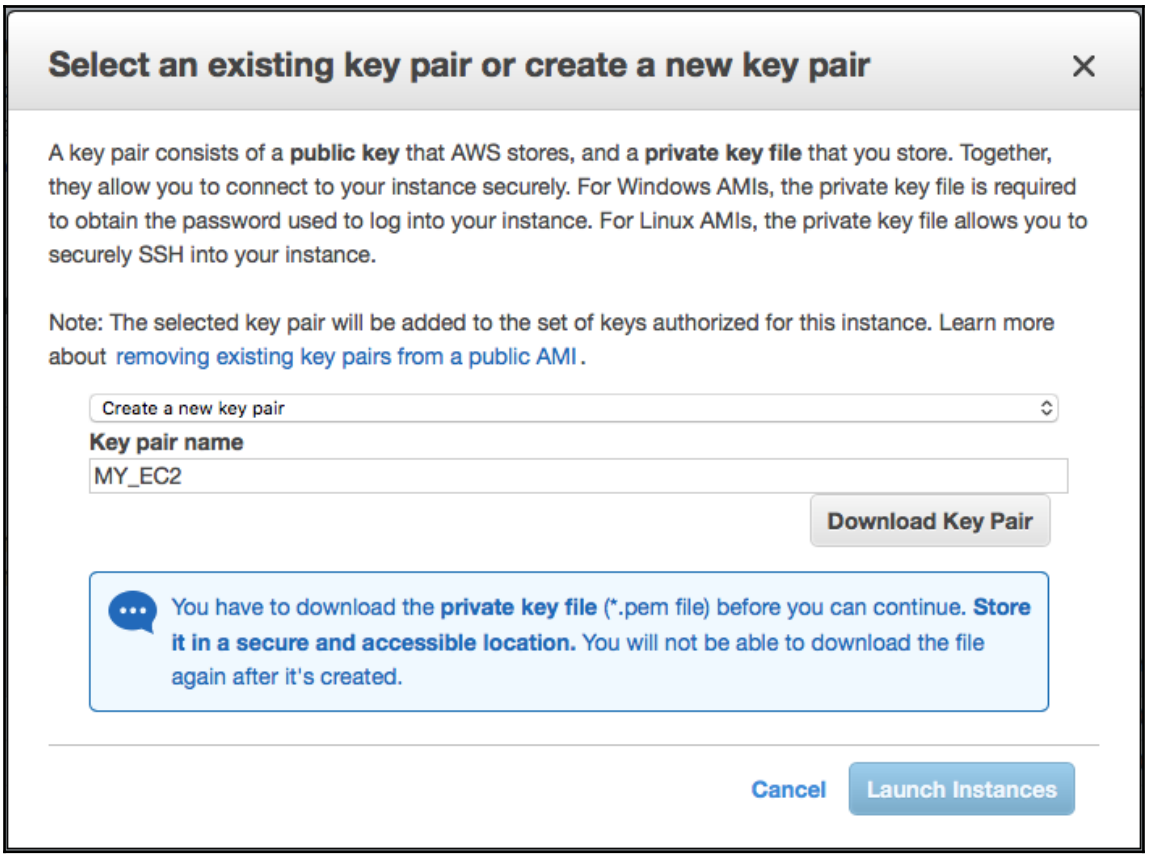
Next, you need to pick the `Instance Type`, which determines what kind of CPU, memory, storage, and network capacity your server will have. Stick with the default option, t2.micro (it's free of charge) and click the gray **Next: Configure Instance Details** button.

Amazon EC2 wizard will then present the instance configuration page, with the SSH port

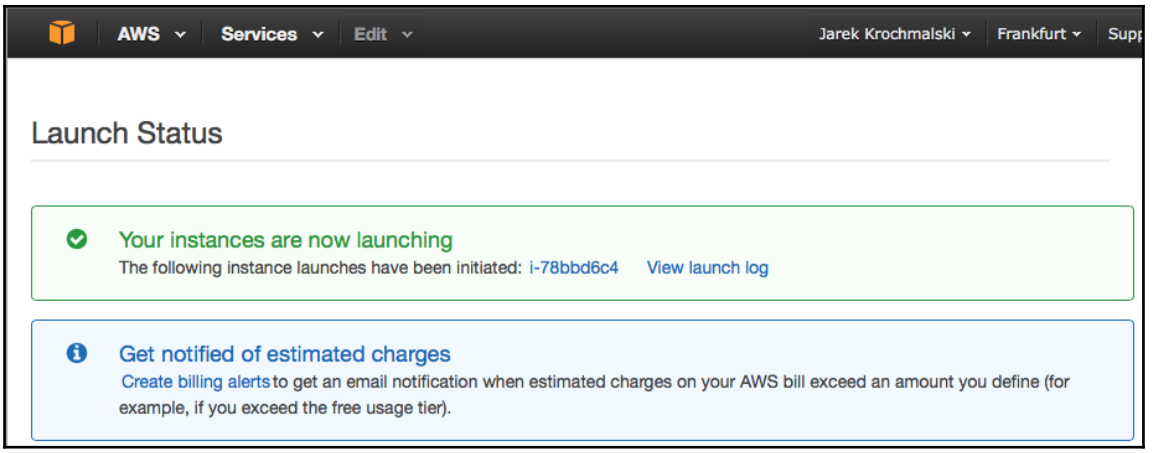
(22) open by default. Depending on your needs, you can add open more ports, like HTTP (80) if you plan to run a web application accessible through a web browser:



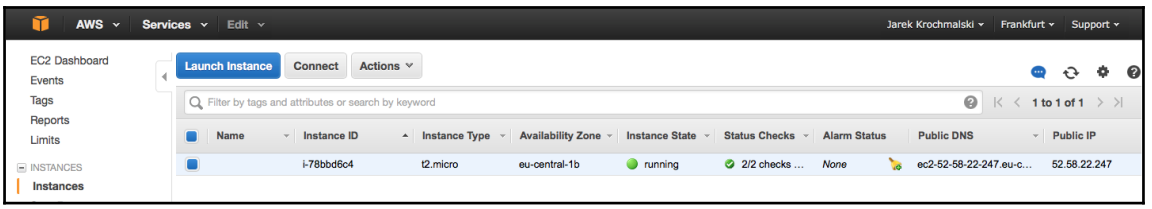
Now comes the important part – to be able to remotely login into your instance, you will need a key pair. It consists of a public and private key file that you must use to connect to your EC2 instance over SSH. Select **Create a new key pair** from the drop-down list, give it a name like MY_EC2 for example, and click **Download Key Pair** button:



If you click the **Launch Instances** button, the start process begins and the status will be shown:



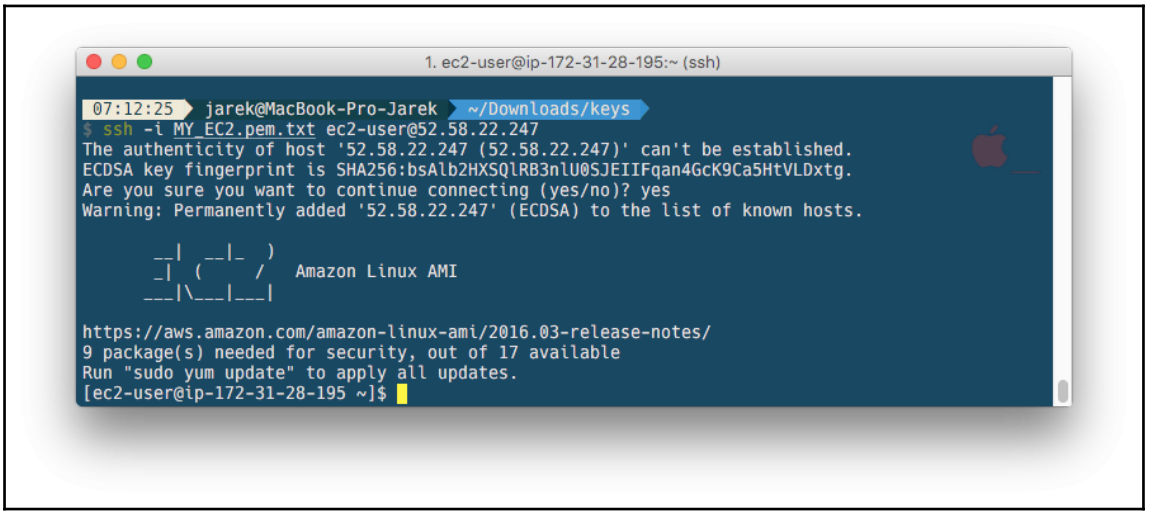
You can also check the status of your AWS instance by picking the Instances from the Management Console menu. It's a handy page, useful to manage all your instances. The status page will list all your cloud machines, their status like running or stopped and also their public IP you can use to login into the instance:



Our newly created instance seems to be running fine, so let's try to login to its shell. Head to the directory you previously saved the keys generated by Amazon and execute the commands:

```
chmod 400 MY_EC2.pem.txt
ssh -i MY_EC2.pem.txt ec2-user@52.58.22.247
```

where `MY_EC2.pem.txt` is the filename of the generated keys and 52.58.22.247 is the public IP address of your remote instance – those two will be different for you of course. After running the SSH login, you will be greeted by Amazon Linux:



From now on, Docker installation process doesn't differ much from any Linux setup. At first, it's good to upgrade the operating system software to ensure are bug fixes are in place. To do this on Amazon Linux, execute the following:

```
sudo yum update -y
```

Next, install Docker using the yum package manager and add your user to the docker group:

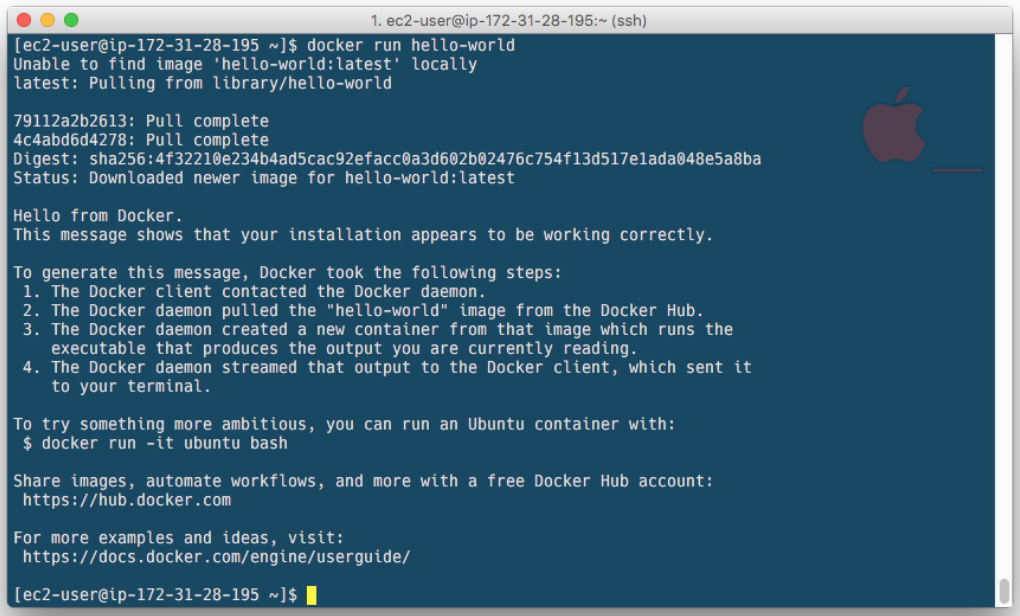
```
sudo yum install -y docker
sudo usermod -a -G docker ec2-user
```

After adding your user to the docker Linux group, you will need to logout and login again, to be able to run Docker as a normal, non-root user.

After installing Docker, let's check directly if it's running or not. We will do it by firing up the **hello-world** image. Execute the following command:

```
docker run hello-world
```

Docker will fetch the image from the Docker Hub and then run it:



The famous **Hello from Docker** message simply says, that now you are successfully running a Docker container in the AWS cloud!

Now we are after the installation process, and hopefully you were able to run the sample hello-world image on the operating system of your choice. Let's dive a little bit deeper into the world of containerization and learn more about containers and images in the next chapter.

Summary

3
Understanding Images and Containers

In previous chapter we learned out how to install Docker on Windows, Mac OS, Linux and Amazon EC2 cloud. So far you should have Docker running on your machine and be able to run the hello-world image in a container. In this chapter we will dive deeper into the world of images and containers. Later, we will also cover the image distribution related terms, like Docker repository, registry and index.

Please be advised that we will mainly use the shell (or command prompt in Windows) to execute Docker commands. You can always execute `docker help` to get the description of available commands. Executing docker help with a name of specific command, like `docker help pull` for example, will display information about this given command with a brief description of available options:

Let's start by explaining in details what images, layers and containers are.

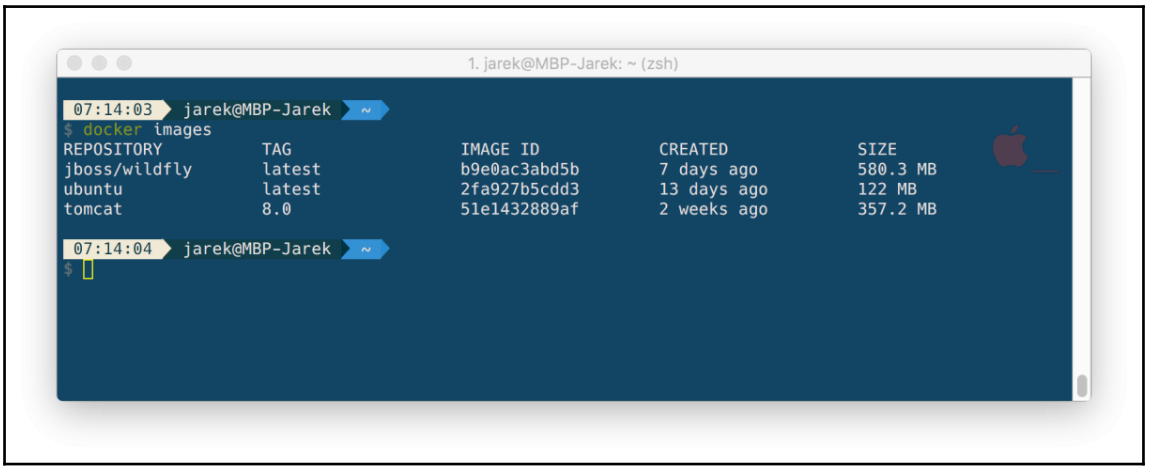
Images

You can think of an image as a read only template which is a base foundation to run a container on. It's like a template, which contains everything your application needs to operate. It can be Ubuntu Linux with a web server and your web application installed. Every image starts from a base image, for example ubuntu, a base Linux image. You can create images yourself – images are created using a series of commands (called "instructions"), described in the Dockerfile. It is an ordered collection of root filesystem changes (like running a command, adding a file or directory, creating environmental variables) 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 of an image, executes the instructions one by one, and returns a final image. Each instruction creates a new layer in the image. We will cover the process in the Chapter 6 – Building Images.

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. It's important to know, that Docker uses image to run your code, not the Dockerfile. The Dockerfile is used to create the image when you run `build` command – we will also get back to it in the Chapter 6 – Building Images. Also, if you publish your image to the Docker Hub, you publish a resulting image, not a source Dockerfile. We will describe the process later in this book, in Chapter

Publishing Images.

Local images you have on your machine can be listed by running `docker images` command:



The images command will display a table with the following columns:

- `REPOSITORY` – The name of the repository. We will explain it in detail at the end of this chapter.
- `TAG` – This is kind of a label attached to the image, similar to Git or other version control systems tag. It represents a specific set point in the repositories' commit history. You can have multiple tags when building the image. There's even a special tag called "latest" which represents the latest version. The full form of a tag is [REGISTRYHOST/][USERNAME/]NAME[:TAG], but the TAG column is just the [:TAG] part of the full tag. We will cover tagging in details later in the Chapter 6 – Building Images.
- `IMAGE ID` – This is the identifier for the image (actually it's the first 12 characters of the true identifier for an image). You may use it to refer to a specific image when executing image commands, but you can also use the image's name.
- `CREATED` – The date represents the time the repository was created. You can use it to verify how fresh the build of image is.
- `SIZE` – The size of the image.

To remove all images you have on your system, execute the following:

```
docker rmi $(docker images -q)
```

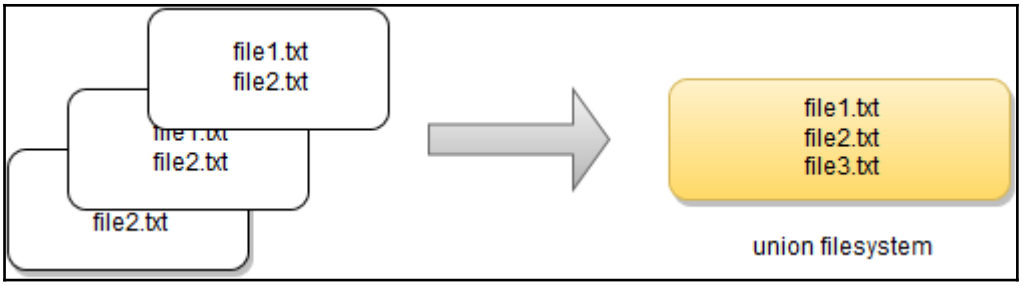
To remove all un-tagged docker images, use the list of images and a filter:

```
docker rmi $(docker images -q -f dangling=true)
```

Alternatively from creating your image from the scratch, you can pick already prepared image from a hundreds available on the Internet. Also, you can publish an image in your private hub, so other people in your organization can pull it and reuse. We will cover the process of looking for images in the Chapter 5 – Finding Images. The downloaded images can be updated and extended freely, so downloading base image is great way to get a serious speed boost when developing one by yourself. It's a very common practice to download ready-to-run image, like a webserver or database for example, and build on top of it. You can have for example a base Apache image you could use this as the base of all your web application images. This is possible due to the internal nature of an image – layers, which Docker images are composed from. We have said a while ago, that every instruction in the Dockerfile creates a new layer. Let's explain now what they are.

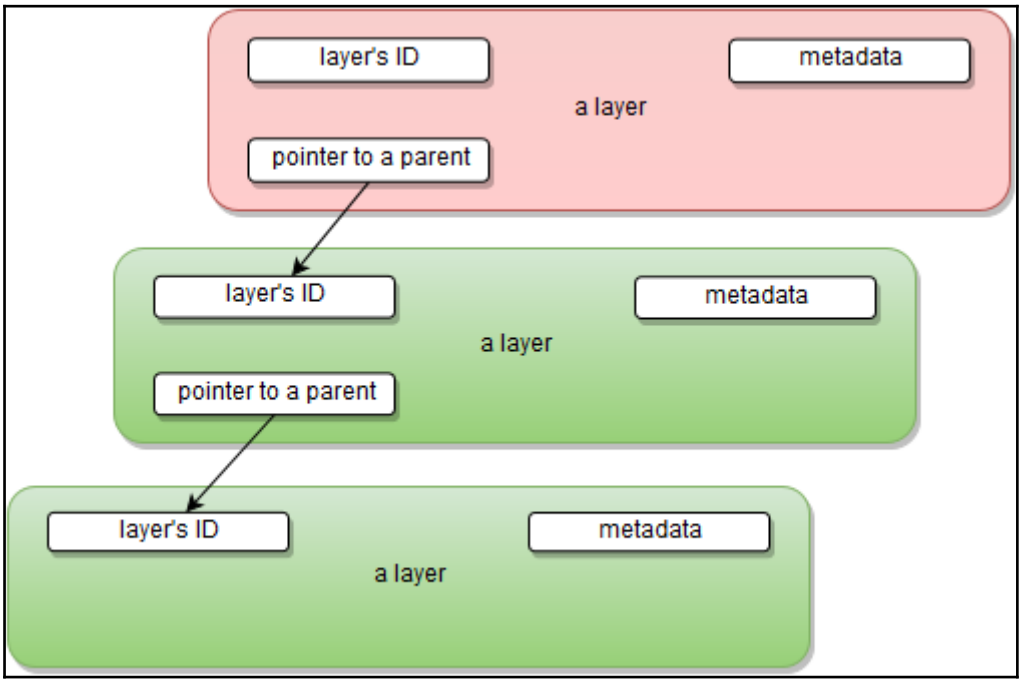
Layers

Each image consists of a series of layers which are stacked one on another. By using the union filesystem, Docker combines all these layers into a single image entity. Union file system allows transparent overlaying files and directories of separate file systems, giving a single, consistent filesystem as a result, as you can see on the diagram:

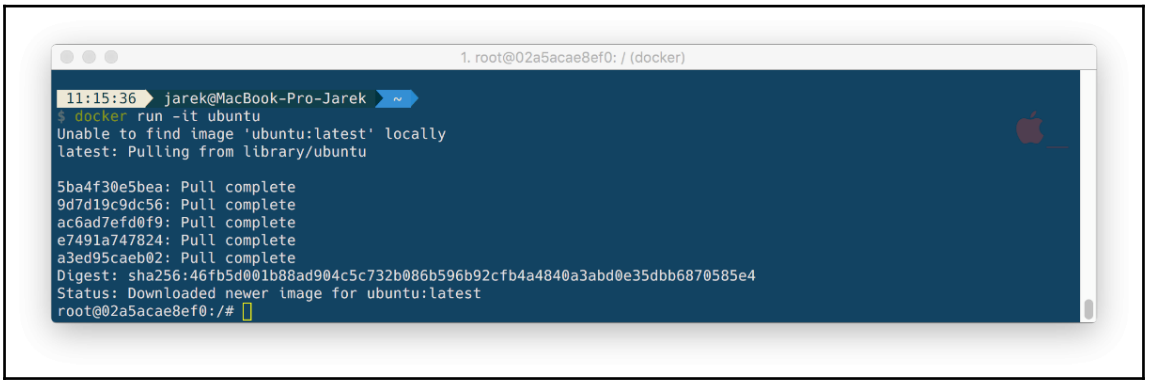


Contents and structure of directories which have the same path within these separate file systems will be seen together in a single merged directory, within the new, virtual-like filesystem. In other words, the filesystem structure of the top layer will merge with the structure of the layer beneath. 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 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 on the following diagram:

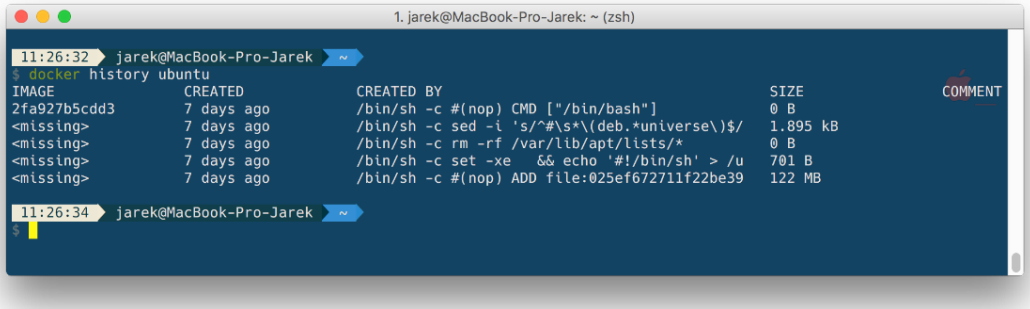


As you pull the image from Docker Hub, you actually can see the progress of each dependent layer being downloaded. Here's an example for the latest Ubuntu Linux:



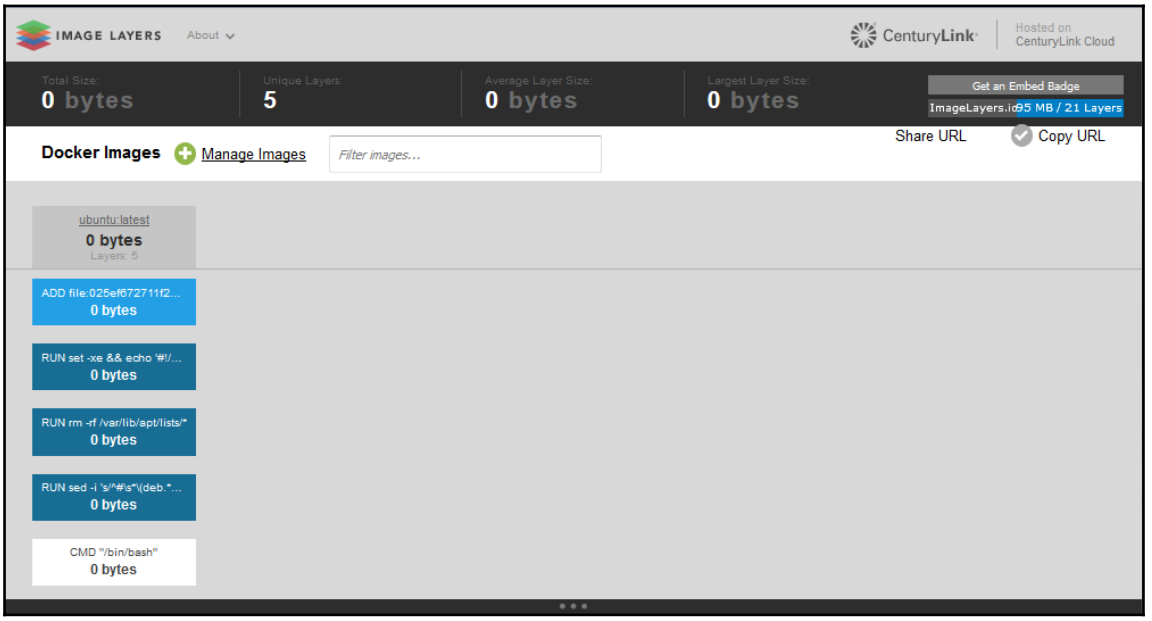
Another gain of using layers is the persistence of history. Layers can provide a history of how a specific image was built. Once all the layers are finished downloading, you can list the layers in the specific image using the `history` command:

```
docker history ubuntu
```



Each line in the `history` command's output corresponds to a commit to a filesystem. The values in the `SIZE` column add up to the corresponding `SIZE` column for the image in docker image.

You can also see the graphical representation of the image using the ImageLayers web application available at `https://imagelayers.iron.io`:



Layers have some interesting features. First – they are reusable. If two different images will have a common part, let's say it will be a Linux shell 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 remember, layers are read-only. When downloading the second image, the layer will be reused and only the difference will be pulled from them 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 bumping the version of your application, only the single layer gets modified. Instead of distributing the whole image, you push just the update, making 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 layers reuse feature, the whole process is a lot faster. Because of layers, Docker is lightweight in comparison to full virtual machines, which doesn'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.

Layers are additive – in result image can get quite large.

For example, 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 be still present in the layer history. Imagine this scenario: you pull the base Ubuntu image, install the Wildfly application server. Then you change your mind, uninstall the Wildfly and install Tomcat instead. All those removed files 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 file-system can make a big difference in the size of your images. Growing 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.

To "flatten" the image, you can export it to a TAR file, using the `export` command:

```
docker export <CONTAINER ID> > /home/docker/myImage.tar
```

Exporting the image to TAR will not preserve its history.

Exported file can then be imported back, using the import command:

```
cat /home/docker/myImage.tar | docker import – some-name:latest
```

If the free disk space is really an issue, you can pipe the output stream of export into the input stream of import:

```
docker export <CONTAINER ID> | docker import – exampleimagelocal:new
```

Alternatively, you can use the `docker-squash` utility, available at GitHub `https://github.com/jwilder/docker-squash`, to make your images smaller. It will quash multiple Docker layers into one in order to create an image with fewer and smaller layers. Squashed images work the same as they were originally built, because this utility retains Dockerfile commands like `PORT` or `ENV`. In addition, deleted files in later layers are actually removed from the image when squashed.

If there's a need, you can also extract data files from the finished container with `cp` (from "copy") command:

```
docker cp <CONTAINER ID>:/path/to/find/files /path/to/put/copy
```

Layers and images are closely related to each other. Docker allows dealing with images and their layers by a couple of commands and we have been using most of them already. Let's summarize them now:

Image related command	Description
images	List images
build	Build an image from a Dockerfile
history	Show the history of an image
import	Create new filesystem image from the contents of a TAR archive
load	Load an image from a TAR archive
rmi	Remove one or more images
save	Save an image contents to a TAR archive
inspect	Return low-level information on an image

Layers are a great feature in a container world. When used wisely, can be a great help when creating images. But, they also have a limitation. At present the `AUFS limit of 42 layers`. It means that you should group similar commands where it is possible – it will result with just one single layer.

As we have said before, Docker images are stored as series of read-only layers. This means that, once the container image has been created, it does not change. But having all the file system read-only would not have a lot of sense. What about modifying an image? 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 read-write 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. To run a container, use the same command we were using in the previous chapter when we have been testing our installation: `docker run`:

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

There are a lot of run command options that can be used. Some of them include the network configuration for example (we will explain Docker's networking in the next chapter), the `-it` (from "interactive") option tells Docker to make the container interactive and to attach a

terminal to its output and input. We will cover all of them in details in the Chapter 7 – Running the Software. Now, let's just focus on the idea of the container to better understand the whole picture. Let's try a simple command to start a new container using the latest version of Ubuntu. As a result of interactive run, once this container starts, you will get a bash prompt shell where you can execute Ubuntu's commands, like in a normal, ordinary shell:

```
docker run –it ubuntu:latest /bin/bash
```

So what happens under the hood when we run this command?

The image which is the "`ubuntu:latest`" in our case, will be pulled down from a "`ubuntu`" repository", unless it's already available on your local machine.

The Docker engine takes the image and adds a read-write layer on top of the layers stack, then initializes image name, ID and resource limits (like CPU and memory). In this phase, Docker will also sets up an 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: `"/bin/bash"` in our case – which starts a shell where you can log in. 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, downloading Node.js libraries using `npm` and so on. All of this will modify the filesystem of the top, writable layer. If you then execute the `commit` command, a new image containing all of your changes will be created and ready to run later.

Sometimes we can tell Docker, that we will not need a container after it is stopped. For this purpose, there is -rm option available for the run command. For example running:

```
docker run –i –t –rm ubuntu:latest /bin/bash
```

will pull and start the latest Ubuntu container and present us an interactive bash shell. As soon as we finish our work and stop the container, it will be deleted from the filesystem releasing some space on a drive.

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, read-write layer), but all processes will be stopped and you will lose anything in memory. This is what differentiates a stopped container from a Docker image. Sometimes you need to stop all of the running containers, so this command may come in handy:

```
docker stop $(docker ps -a -q)
```

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

```
docker ps -a
```

As a result, 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 (assigned by you or the funny name Docker has picked for you):

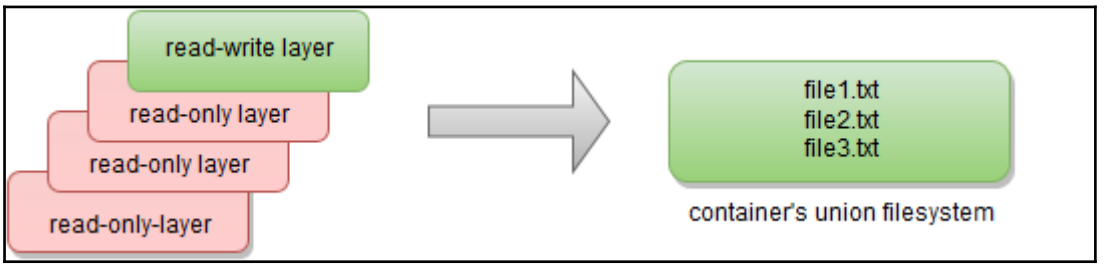
As you can see, the output will contain detailed information about the container status and uptime and a command used to start in the container.

To remove a container, you can just use the `rm` command. If want to remove couple of them at once, you can use the list of containers (given by the `ps` command) and a filter:

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

Saving changes to a container

Although an image is always read-only and immutable, we can actually make changes to a running container – the top layer of a container stack is always the read-write (writable) layer. This can be adding or modifying files, like installing a software package, configuring the operation system and so on.



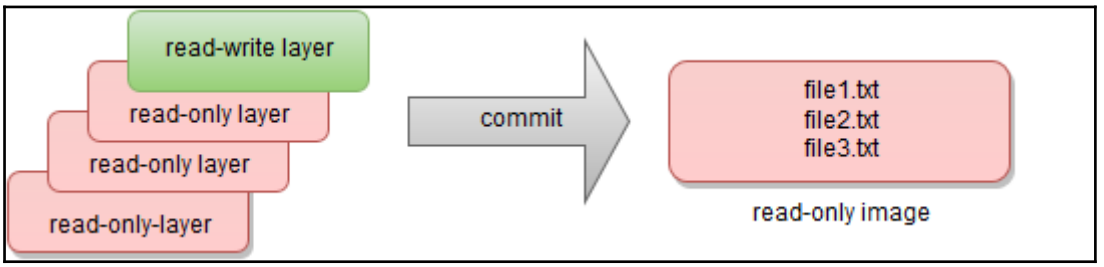
If you modify a file in the running container, the file will be copied out of the underlying read-only layer and into the top, read-write layer. Your changes will be performed only in the top layer, and the union filesystem will hide the underlying file. The original file will not be destroyed – it still exists in the underlying, read-only layer. If you delete the container, and relaunch the same image again, Docker will start a fresh container without any of the changes made in the previously running container.

In other words, your changes to the filesystem will not affect the base image. However, you can create a new image from a running container (and all it changes) using the commit command:

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

To save changes you have made to the container, you must commit them.

During runtime, if the process in a container makes changes to its filesystem, a "diff" is made between the current container filesystem and the filesystem of the image from which the container was created. If you run the `docker commit` command, the diff becomes a new read-only image, from which you can create new containers. Otherwise, if you remove the container, the diff will disappear. You can make updates to a container, but a series of updates will engender a series of new container images, so system rollbacks are easy. Take a look what happens after you do a commit:



Let's see an example. First, we pull a base image called busybox. Busybox combines tiny versions of many common UNIX utilities into a single small executable. It provides replacements for most of the utilities you usually find in GNU like file utilities, shell tools and so on.

```
docker pull busybox
```

Now we make changes to a container of this image in this case we make a new folder:

```
docker run busybox mkdir /home/test
```

At the moment, we can get a busybox container ID using the command:

```
docker ps -a
```

Let's commit this changed container – this will create a new image called "busybox_modfied":

```
docker commit <CONTAINER ID> busybox_modified
```

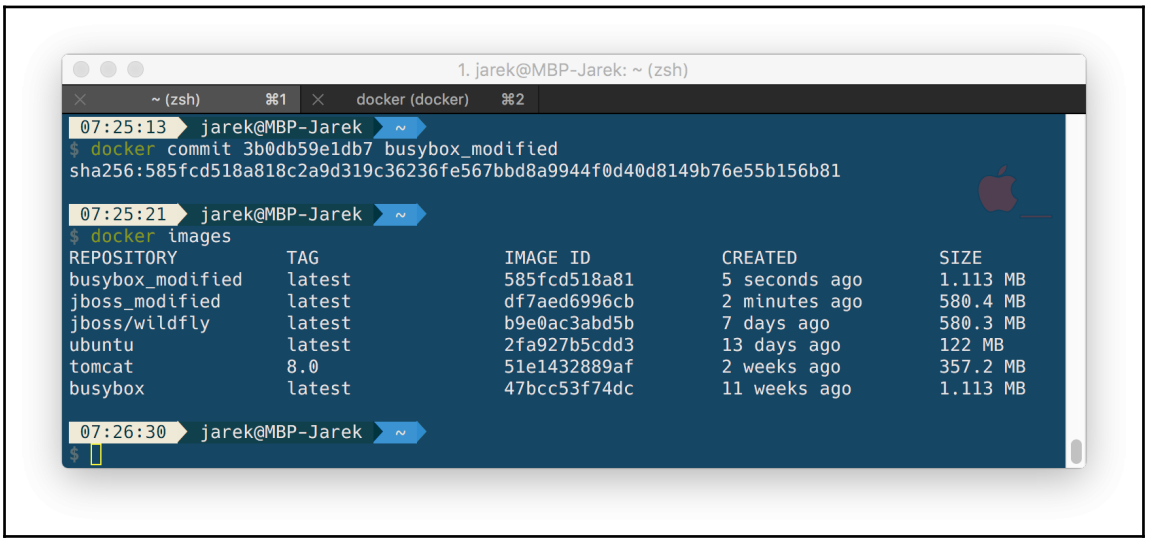
In the response of successful commit, Docker will just output the full ID of newly generated image.

To avoid data corruption or inconsistency, Docker will pause a container you are committing changes into. Although it's not recommended to do so, you have an option to disable this behavior, setting `--pause` option to `false`.

If we list images we have now, both "busybox" and "busybox_modified" should be present on the list. To see them, execute the `images` command:

```
docker images -a
```

As you can see, the new "busybox_modified" is present on the list of images available locally:



To see the difference between both images we can use the following check for folders:

```
    docker run busybox [ -d /home/test ] && echo 'Directory found' || echo
'Directory not found'
    docker run busybox_modified [ -d /home/test ] && echo 'Directory found'
|| echo 'Directory not found'
```

Now we have two different images ("busybox" and "busybox_modified") and we have a container made from "busybox" which also contains the change (the new folder `/home/test`). The 'commit' command takes a container's top-level read-write layer and turns it into a read-only layer. In effect, the container (no matter if it's running or stopped) becomes new, read-only, immutable image.

When the container is deleted the writable layer is also deleted.

Creating images by altering the top writable layer in the container is useful when debugging and experimenting, but it's usually better to use Dockerfile to manage your images in a documented and maintainable way. We will do it in the Chapter 6 – Building Images.

A container is a stateful instantiation of an image.

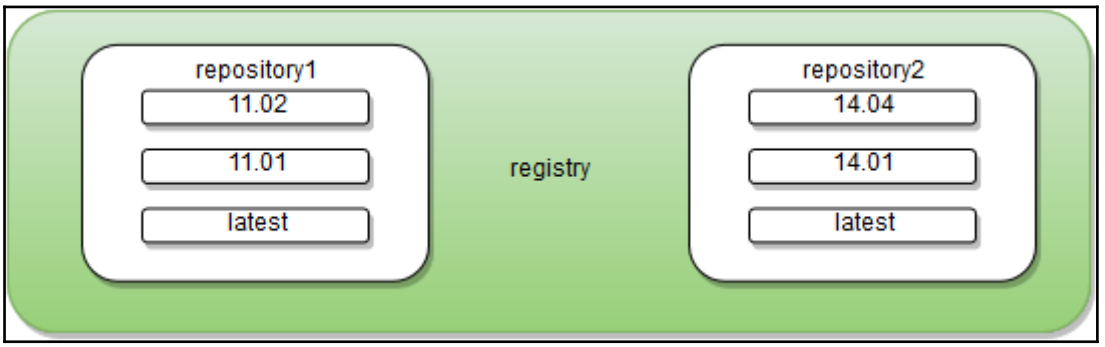
We have been using a couple of container – related commands in this sections, let's summarize them in a table:

Container related command	Description
attach	Attach to a running container
commit	Build an image from a Dockerfile
cp	Show the history of an image
create	Create new filesystem image from the contents of a TAR archive
diff	Load an image from a TAR archive
exec	Remove one or more images
inspect	Save an image contents to a TAR archive
kill	Return low-level information on an image
`start` / `stop` / `restart` / `pause` / `unpause`	Manage the container's run status
logs	Fetch the logs of a container
port	List port mappings or a specific mapping for the container
rename	Rename a container
run	Run a command in a new container
stats	Display a live stream of container(s) resource usage statistics
top	Display the running processes of a container
update	Update configuration of one or more containers
wait	Block until a container stops, then print its exit code

We have now learned about the build (images) and run (containers) pieces of our containerization world. We still are missing the last element – the distribution component. The distribution component of Docker consists of Docker Registry, Index and Repository. Let's focus on them now to have a complete picture.

Docker Registry, REPOSITORY and index

Docker utilizes a hierarchical system for storing images, shown at the diagram below:



The first component in this system is the registry. Images which you build will be stored in a remote registry for others to use. **Docker registry is a service (an application, in fact) that is storing your Docker images.** Docker Hub is example of the publicly available registry – it's free and serves a huge, constantly growing collection of existing images. There are of course other registries available on the Internet like Artifactory (`https://www.jfrog.com/artifactory`), Google Container Registry (`https://cloud.google.com/container-registry`) and Quay (https://quay.io).

Repository, on the other hand is a collection (namespace) of related images, usually providing different versions of the same application or service – in other words **is a collection of different docker images with same name and different tags.** If your app is named "hello-world" and your username (or namespace) for the Registry is "`developingWithDocker`" then your image will placed in the "`developingWithDocker/hello-world`" repository. You can tag an image, and store multiple versions of that image with different IDs in a single named repository, access different tagged versions of an image with a special syntax like username/image_name:tag. Docker repository isquite similar to a Git repository – like in Git, a Docker repository is identified by a URI and can either be public or private. The URI looks like:

```
{registryAddress}/{namespace}/{repositoryName}:{tag}
```

The Docker Hub is the default registry and Docker will pull images from the Docker Hub if you do not specify a registry address.

The registry address can be omitted for repositories hosted with Docker Hub.

We have mentioned tags earlier in this chapter – we will get back to them in details in

the Chapter 6 – Building Images. A registry typically hosts multiple Docker repositories.

The difference between Registry and repository can be confusing at the beginning, so let's describe what will happen if you execute the following command:

```
docker pull ubuntu:14.04
```

The command downloads the image tagged `14.04` within the `ubuntu` repository from the Docker Hub registry. Ubuntu repository doesn't use username, so the namespace part is omitted in this example.

Let's summarize repository related commands in a table:

Repository related command	Description
login	Log in to a Docker registry
logout	Log out from a Docker registry
pull	Pull an image or a repository from a registry
push	Push an image or a repository to a registry
search	Search the Docker Hub for images

Although the Docker Hub is public, you get one private repository for free with your Docker Hub user account, but it's not usable for organizations you're a member of. If you need more accounts you can upgrade your `Docker Hub` plan, which will not be free of charge. There are a couple of payment plans based on the number of private repositories you need. We will cover the Docker Hub in detail in the Chapter 5 – Finding Images and Chapter 8 – Publishing Images.

Private registries, on the other hand, can be setup just for you or other users in your organization, in your company's own network.

You can create the private registry behind your company's firewall.

To run your totally private registry, you can use the Docker Hub itself. It's an open source application, and is also available as a Docker image. The simplest case is just running the command:

```
docker run -d -p 5000:5000 --name registry registry:2
```

As a result, you will start private registry on your own machine.

Last, but not least component you should be aware of is an Index. An *Index* manages search and tagging but also user accounts and permissions. In fact, the registry delegates authentication to the *index*. When executing remote commands like push or pull, the *index* first will look at the name of the image and then check to see if it has a corresponding *repository*. If so, the index verifies if you are allowed to access or modify the image. If you are, the operation is approved and registry takes or sends the image.

Summary

Let's summarize what we have learned so far:

- The Dockerfile is the source code of the Image. It contains ordered instructions how to build an image.
- An image is a specific state of a filesystem: a read-only, frozen immutable snapshot of a live container.
- An image is composed of layers representing changes in the filesystem at various points in time; layers are a bit like the commit history of a git repository
- Containers are runtime instances of an image. They can be running or stopped.
- You can make changes to the filesystem on a container and commit them to make them persisted. Commit always creates a new image.
- Only changes on the filesystem can be committed – memory changes will be lost.
- A r*egistry* holds a collection of named *repositories*, which themselves are a *collection of images* tracked by their IDs. Registry is like a Git repository: you can push and pull containers.

After reading this chapter you should have an understanding the nature of images with their layers and containers. But Docker provides another way of extending and opening containers to the external world: networking and persistent storage. We are going to cover this subject in the next chapter.