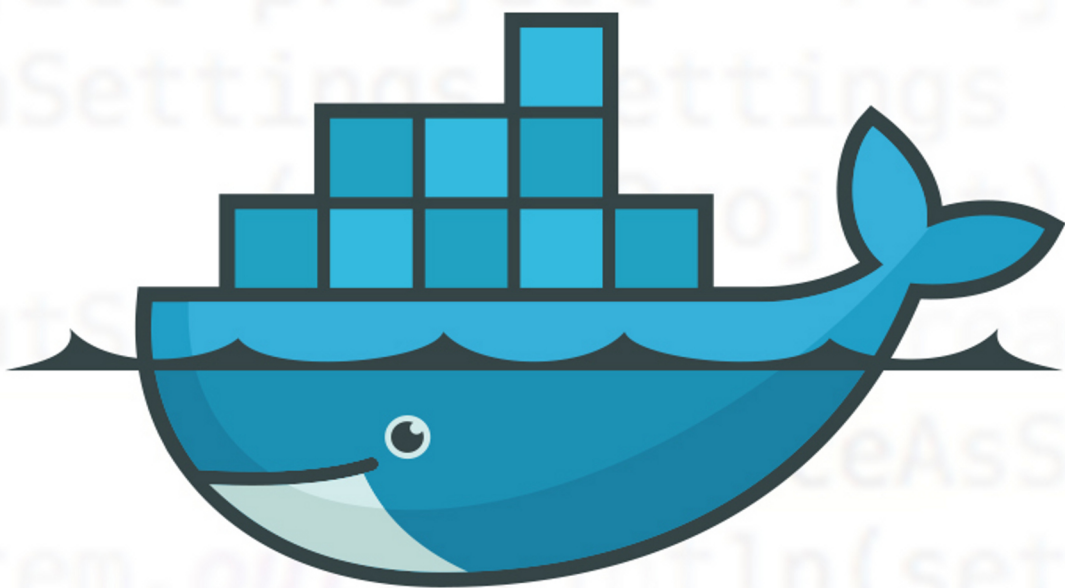


DOCKER CONTAINERIZATION COOKBOOK

Hot Recipes for Docker Automation



docker

JAVA CODE GEEKS



Java Code Geeks
JAVA 2 JAVA DEVELOPERS RESOURCE CENTER

Docker Containerization Cookbook

Contents

1	Docker Tutorial for Beginners	1
1.1	Why do we need Docker?	1
1.2	Machines, Images and Containers	2
1.2.1	Installing docker and docker-machine	3
1.3	Understanding the commands	4
1.3.1	Creating a machine	5
1.3.2	Creating an image	5
1.3.3	Creating a container	5
1.3.4	Cleaning the environment	6
1.4	Running integration tests with Maven	6
1.4.1	Maven configuration	6
1.4.2	Creating integration tests	8
1.4.3	Running the tests	9
1.5	Download the source code	10
2	Install Docker on Ubuntu Tutorial	11
2.1	Requisites	11
2.2	Installation	11
2.2.1	Fixing "Cannot connect to the Docker daemon" error	12
2.3	Basic usage	12
2.3.1	Pulling images	13
2.3.2	Creating images from Dockerfiles	13
2.3.3	Creating containers	13
2.4	Summary	14
3	Docker Kernel Requirements	15
3.1	Introduction	15
3.2	Docker engine dependencies from the Linux kernel	15
3.3	Resource constraining dependencies	16
3.3.1	3.1 Control groups a.k.a cgroups	16
3.3.2	Namespaces	17

3.4	Security dependencies	17
3.4.1	AppArmor	17
3.4.2	Security Enhanced Linux a.k.a SELinux	17
3.4.3	Posix capabilities a.k.a Capabilities	17
3.4.4	Secure Computing Mode a.k.a seccomp	17
3.5	Networking dependencies	18
3.5.1	Netfilter	18
3.5.2	IPTables	18
3.5.3	Netlink	18
3.6	File system dependencies	18
3.6.1	Device mapper	18
3.7	Other non-kernel dependencies	19
3.7.1	LibContainer a.k.a RunC	19
3.7.2	LXC	19
3.8	Summary	19
4	Docker Hello World Example	20
4.1	Introduction	20
4.2	Run Docker Hello World image provided by Docker	20
4.3	Get a "Hello, world" Printed from Another Basic Docker Image	21
4.4	Write a Simple "Hello, World" Program in Java and Run it Within a Docker Container	22
4.4.1	Create HelloWorld.java	22
4.4.2	Create a Dockerfile	23
4.4.3	Create Docker Hello World Image and Run it	23
4.5	Execute a "Hello, World" Program in Java Using Gradle and Docker	25
4.5.1	Initialize a New Java Project and Create HelloWorld.java	25
4.5.2	Create a Dockerfile	26
4.5.3	Include Gradle-Docker Plugin	26
4.5.4	Configure the Docker Tasks in Gradle Build File	26
4.5.5	Create Docker Container and Run it	27
4.6	Summary	29
4.7	Download the Source Code	29
5	Docker as a Service	30
5.1	Why Docker as a Service?	30
5.2	Container as a Service (CaaS)	31
5.3	Docker CaaS	32
5.4	Summary	32

6	Docker Build Example	33
6.1	Introduction	33
6.2	Installation & Usage	33
6.3	Context definition	34
6.3.1	Build with PATH	35
6.3.2	Build with URL	35
6.3.3	Build with -	36
6.4	Dockerfile location	36
6.5	Example	36
6.6	Conclusion	36
7	Docker Compose example	37
7.1	Introduction	37
7.2	Docker Toolbox setup	38
7.3	Docker compose example: An HTTP server connected to Redis	38
7.3.1	Creating the docker-compose.yml file	39
7.3.2	Creating the web server	39
7.3.3	Lauching Docker Compose	42
7.3.4	Testing the web server	42
7.4	Download the complete source code	42
8	Configuring DNS in Docker	43
8.1	Introduction	43
8.2	Basics of Networking Configurations in Docker Engine	43
8.2.1	2.1. Inspect Networks	43
8.2.2	2.2. Create User Defined Networks	44
8.2.3	2.3. Connect Containers Within a Network	45
8.3	Setup Container DNS in Bridge Network	46
8.4	Setup Container DNS in a User Defined Network	47
8.5	Summary	48
9	Docker Start Container Example	49
9.1	Installation	49
9.2	Pulling a sample image	49
9.3	Creating a container from an image and running it: run	50
9.3.1	Running containers in detached mode	50
9.3.2	Giving a name to the container	50
9.3.3	Mapping container ports to host ports	51
9.3.4	Run a container and get the command line	51
9.3.5	Specifying a username	51
9.4	Starting an existing container: restart	51
9.5	Summary	52

10 Docker List Containers Example	53
10.1 Installation	53
10.2 Setting up some containers	53
10.2.1 Pulling a sample image	53
10.2.2 Creating sample containers	53
10.3 Listing containers	54
10.3.1 Formatting the output	54
10.3.2 Saving format templates in Docker configuration file	55
10.3.3 Using filters	56
10.4 Summary	56

Copyright (c) Exelixis Media P.C., 2016

All rights reserved. Without limiting the rights under copyright reserved above, no part of this publication may be reproduced, stored or introduced into a retrieval system, or transmitted, in any form or by any means (electronic, mechanical, photocopying, recording or otherwise), without the prior written permission of the copyright owner.

Preface

Docker is the world's leading software containerization platform. Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, system libraries - anything that can be installed on a server. This guarantees that the software will always run the same, regardless of its environment. (Source: <https://www.docker.com/what-docker>)

Docker provides an additional layer of abstraction and automation of operating-system-level virtualization on Linux. Docker uses the resource isolation features of the Linux kernel such as cgroups and kernel namespaces, and a union-capable file system such as OverlayFS and others to allow independent "containers" to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines (Source: [https://en.wikipedia.org/wiki/Docker_\(software\)](https://en.wikipedia.org/wiki/Docker_(software))).

In this ebook, we provide a compilation of Docker examples that will help you kick-start your own automation projects. We cover a wide range of topics, from installation and configuration, to DNS and commands. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

JCGs (Java Code Geeks) is an independent online community focused on creating the ultimate Java to Java developers resource center; targeted at the technical architect, technical team lead (senior developer), project manager and junior developers alike.

JCGs serve the Java, SOA, Agile and Telecom communities with daily news written by domain experts, articles, tutorials, reviews, announcements, code snippets and open source projects.

You can find them online at <https://www.javacodegeeks.com/>

Chapter 1

Docker Tutorial for Beginners

In this article we are going to explain the main features of Docker. We are going to briefly explain why containers are necessary, which are the main commands to use and how to use Docker in our integration tests. The following table shows an overview of the whole article:

1.1 Why do we need Docker?



Figure 1.1: docker

Docker is a tool to avoid the usual headaches of conflicts, dependencies and inconsistent environments, which is an important problem for distributed applications, where we need to install or upgrade several nodes with the same configuration.

Docker is a container manager, which means that it is able to create and execute containers that represent specific runtime environments for your software. In contrast with virtual machines like VirtualBox, Docker uses resource isolation features of the Linux kernel to allow independent “containers” to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines. A computer with Docker can run multiple containers at the same time.

Therefore, the automation of Docker mainly offers facilities for integration tests and continuous delivery.

1.2 Machines, Images and Containers

In order to use Docker, it is important to have clear some vocabulary. Docker distinguishes three important concepts: Docker machines, Docker images and containers.

Docker containers are executions of a specific runtime environment. For example, to simulate a machine with a specific database up. These runtime environments are called Docker images, which are the result of executing the set of commands that appear in a specific script-like file called `Dockerfile`. Therefore, Docker allows having multiple containers of a specific Docker image. These could be compared with the concepts of program and process. Indeed, containers like processes, can be created, stopped, died, or running.

Docker machines are local (and virtual) machines or remote machines (e.g. in a cloud such as Amazon AWS or DigitalOcean) with Docker running. Like physical machines, Docker machines have a specific IP address. Each Docker machine can manage multiple Docker images and containers. From our own personal computer, the `docker-machine` command, allows us connecting to all our Docker machines to manage their containers and images.

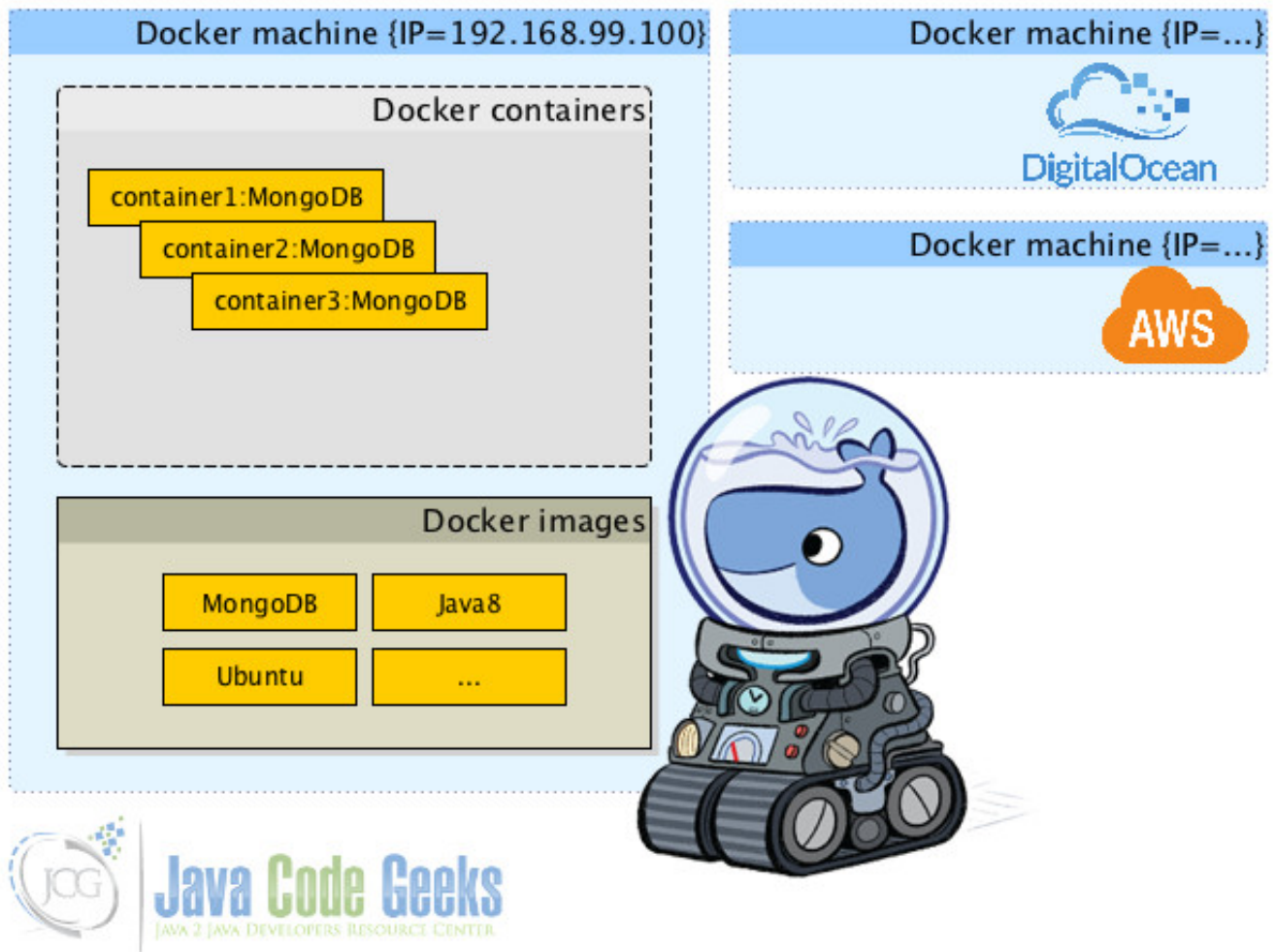


Figure 1.2: docker concepts

1.2.1 Installing docker and docker-machine

Follow the next steps to install docker and docker machine

Download and install the docker binary file from [here](#). Once, you have followed the installation instructions, you should be able to run the following command.

Docker installation test

```
$ docker run ubuntu:14.04 /bin/echo 'It works!'
It works!
```

Download the Docker Machine binary and update your PATH.If you are running OS X or Linux:

Unix Docker Machine installation

```
$ curl -L https://github.com/docker/machine/releases/download/v0.6.0/docker-machine-`uname -s`-`uname -m` > /usr/local/bin/docker-machine && \
chmod +x /usr/local/bin/docker-machine
```

If you are running Windows with git bash

Windows Docker Machine installation

```
$ if [[ ! -d "$HOME/bin" ]]; then mkdir -p "$HOME/bin"; fi && \  
curl -L https://github.com/docker/machine/releases/download/v0.6.0/docker-machine-Windows- ←  
x86_64.exe > "$HOME/bin/docker-machine.exe" && \  
chmod +x "$HOME/bin/docker-machine.exe"
```

Otherwise, simply use the installer from the [Docker official releases](#).

1.3 Understanding the commands

There are many docker commands. In fact, if we run simply `docker` from our terminal, we can see all the available commands.

Docker commands

```
Commands:  
attach      Attach to a running container  
build       Build an image from a Dockerfile  
commit      Create a new image from a container's changes  
cp          Copy files/folders from a container to a HOSTDIR or to STDOUT  
create      Create a new container  
diff        Inspect changes on a container's filesystem  
events      Get real time events from the server  
exec        Run a command in a running container  
export      Export a container's filesystem as a tar archive  
history     Show the history of an image  
images      List images  
import      Import the contents from a tarball to create a filesystem image  
info        Display system-wide information  
inspect     Return low-level information on a container or image  
kill        Kill a running container  
load        Load an image from a tar archive or STDIN  
login       Register or log in to a Docker registry  
logout      Log out from a Docker registry  
logs        Fetch the logs of a container  
pause       Pause all processes within a container  
port        List port mappings or a specific mapping for the CONTAINER  
ps          List containers  
pull        Pull an image or a repository from a registry  
push        Push an image or a repository to a registry  
rename      Rename a container  
restart     Restart a running container  
rm          Remove one or more containers  
rmi         Remove one or more images  
run         Run a command in a new container  
save        Save an image(s) to a tar archive  
search      Search the Docker Hub for images  
start       Start one or more stopped containers  
stats       Display a live stream of container(s) resource usage statistics  
stop        Stop a running container  
tag         Tag an image into a repository  
top         Display the running processes of a container  
unpause     Unpause all processes within a container  
version     Show the Docker version information  
wait        Block until a container stops, then print its exit code
```

Run `'docker COMMAND --help'` for more information on a command.

In this section, we will see how to manage docker machines, how to pull a docker image using an specific docker machine and how to create a container for such docker image.

1.3.1 Creating a machine

First of all, we will list the set of available docker machines with the following command:

List Docker Machines

```
docker-machine ls
```

In case of not having any machine, we are going to create a new one as follows:

Docker installation test

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

Notice that in order to create a docker machine, we need to specify a driver, which will determine if it is a virtual machine, in this case virtual box, or if it is a connection to docker running in an external machine.

1.3.2 Creating an image

Docker images are created in an specific docker machine. In order to define the docker machine that we want to use, we need to run the following command:

Select docker machine

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

In order to check the effects, list the available docker machines again. At this moment, a new machine should appear with a new IP address (192.168.99.100) and contains an asterisk in the ACTIVE column.

Active Docker Machine

NAME	ACTIVE	DRIVER	STATE	URL	SWARM
default	*	virtualbox	Running	tcp://192.168.99.100:2376	

Docker images are the binary files of runtime environments. The contents of a docker image are the result of executing the set of instructions that appear in a plain text file called `Dockerfile`. However, in order to avoid building docker images from `Dockerfiles` over and over again, there is an online repository for docker images called [Docker Hub](#).

For example, in order to pull a MongoDB image, we only need to execute the docker pull command as follows:

Docker pull command

```
docker pull mongo
```

In order to see the installed docker images in our docker machine, use the following command. Notice that a new entry appears with the Mongo image. Each image has a unique identifier of the docker image for our machine called `IMAGE ID`.

List Docker Images

```
docker images
```

If you are interested to understand how to create a `Dockerfile`, we strongly recommend follow [this](#) example.

1.3.3 Creating a container

Docker containers are executions of docker images. In order to create a container from an image, we use the command `docker run`. For example:

List Docker Images

```
docker run mongo
```

After that, we can check the list of containers, with the following instruction.

List docker containers

```
docker ps -a
```

Notice that our container has an identifier too, called `CONTAINER ID`. We will use it later.

The IP address to connect to this container from our application is the IP of our docker machine (192.168.99.100).

1.3.4 Cleaning the environment

At this point we have downloaded a docker machine and we have started a docker container. In order to stop and remove the container and the image we need to execute the following instructions:

Stop the container:

Stop Docker Container

```
docker stop #containerId
```

Remove the container:

Remove Docker Container

```
docker rm #containerId
```

Remove the docker image:

Remove Docker Image

```
docker rmi #imageId
```

1.4 Running integration tests with Maven

There are different approaches to design integration tests with Docker. In this example, we will design a Maven project, which before executing JUnit tests, it automatically starts a MongoDB Docker container.

1.4.1 Maven configuration

First of all, copy this `pom.xml` file. We are going to use the `docker-maven-plugin` to automatically load the defined Docker images and create a container before running the integration tests. In order to support integration tests, we use the `maven-failsafe-plugin`.

`pom.xml`

```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>javacodegeeks</groupId>
  <artifactId>javacodegeeks</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```

```
<project.build.resourceEncoding>UTF-8</project.build.resourceEncoding>
<maven.compile.encoding>UTF-8</maven.compile.encoding>
</properties>

<build>
  <plugins>

    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <encoding>UTF-8</encoding>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-failsafe-plugin</artifactId>
      <version>2.19.1</version>
      <configuration>
        <includes>
          <include>*/IntegrationTest.java</include>
        </includes>
      </configuration>
      <executions>
        <execution>
          <id>integration-test</id>
          <goals>
            <goal>integration-test</goal>
          </goals>
        </execution>
        <execution>
          <id>verify</id>
          <goals>
            <goal>verify</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
    <!-- Tell surefire to skip test, we are using the failsafe plugin -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.10</version>
      <configuration>
        <skip>true</skip>
      </configuration>
    </plugin>
    <plugin>
      <groupId>io.fabric8</groupId>
      <artifactId>docker-maven-plugin</artifactId>
      <version>0.15.0</version>
      <configuration>
        <logDate>default</logDate>
        <autoPull>true</autoPull>
        <dockerHost>tcp://192.168.99.100:2376</dockerHost>
        <certPath>${user.home}/.docker/machine/certs/</certPath>
        <images>
          <image>
            <name>mongo</name>
          </image>
        </images>
      </configuration>
    </plugin>
  </plugins>
</build>
```



```

        <ports>
            <port>27017:27017</port>
        </ports>
    </run>
</image>

</images>
</configuration>
<!-- Hooking into the lifecycle -->
<executions>
    <execution>
        <id>start</id>
        <phase>pre-integration-test</phase>
        <goals>
            <goal>start</goal>
        </goals>
    </execution>
    <execution>
        <id>stop</id>
        <phase>post-integration-test</phase>
        <goals>
            <goal>stop</goal>
        </goals>
    </execution>
</executions>
</plugin>
</plugins>
</build>
<dependencies>
    <dependency>
        <groupId>org.mongodb</groupId>
        <artifactId>mongo-java-driver</artifactId>
        <version>3.2.2</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
</project>

```

1.4.2 Creating integration tests

Integration tests are executed after creating a MongoDB container by the `maven-docker-plugin`. We are going to create a test to validate the following `Zoo` component, which only has one operation called `addAnimal` that inserts an animal to MongoDB.

`Zoo.java`

```

package javacodegeeks;

import org.bson.Document;

import com.mongodb.MongoClient;

public class Zoo {

    private MongoClient client;

```

```
public Zoo(MongoClient client) {
    this.client = client;
}

public void addAnimal(String name, String type) {
    Document doc = new Document("name", "lion").append("type", "mammal");
    client.getDatabase("mydb").getCollection("animals").insertOne(doc);
}
}
```

To test the Zoo component, we just validate, with an empty database, that after invoking the addAnimal operation, MongoDB contains a new animal.

IntegrationTest.java

```
package javacodegeeks;

import org.bson.Document;
import org.junit.AfterClass;
import org.junit.Assert;
import org.junit.BeforeClass;
import org.junit.Test;
import com.mongodb.MongoClient;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoDatabase;

public class IntegrationTest {

    private static MongoClient client;

    @BeforeClass
    public static void start() {
        client = new MongoClient("192.168.99.100");
        MongoDatabase db = client.getDatabase("mydb");
        db.createCollection("animals");
    }

    @Test
    public void testMongoInsert() {

        Zoo zoo = new Zoo(client);
        zoo.addAnimal("lion", "mammal");

        FindIterable it = client.getDatabase("mydb").getCollection("animals")
            .find(new Document("name", "lion"));
        Assert.assertNotNull(it.first());
    }

    @AfterClass
    public static void stop(){
        client.close();
    }
}
```

Notice that the IP address of our MongoDB database is our Docker machine.

1.4.3 Running the tests

To run our integration tests, we simply need to execute the `mvn verify` command.

Run integration tests

```
mvn verify
```

Which will produce these contents:

Mvn verify output

```
[INFO] Building jar: /Users/rpau/workspace/javacodegeeks/target/javacodegeeks-1.0-SNAPSHOT. ↵
  jar
[INFO]
[INFO] --- docker-maven-plugin:0.15.0:start (start) @ javacodegeeks ---
[INFO] DOCKER> [mongo] : Start container a2a9d37cbbfd
[INFO]
[INFO] --- maven-failsafe-plugin:2.19.1:integration-test (integration-test) @ javacodegeeks ↵
  ---

  T E S T S
Running javacodegeeks.IntegrationTest
may 15, 2016 6:11:10 PM com.mongodb.diagnostics.logging.JULLogger log
INFORMATION: Cluster created with settings {hosts=[192.168.99.100:27017], mode=SINGLE, ↵
  requiredClusterType=UNKNOWN, serverSelectionTimeout='30000 ms', maxWaitQueueSize=500}
may 15, 2016 6:11:10 PM com.mongodb.diagnostics.logging.JULLogger log
INFORMATION: No server chosen by WritableServerSelector from cluster description ↵
  ClusterDescription{type=UNKNOWN, connectionMode=SINGLE, all=[ServerDescription{address ↵
  =192.168.99.100:27017, type=UNKNOWN, state=CONNECTING}]]. Waiting for 30000 ms before ↵
  timing out
may 15, 2016 6:11:10 PM com.mongodb.diagnostics.logging.JULLogger log
INFORMATION: Opened connection [connectionId{localValue:1, serverValue:1}] to ↵
  192.168.99.100:27017
may 15, 2016 6:11:10 PM com.mongodb.diagnostics.logging.JULLogger log
INFORMATION: Monitor thread successfully connected to server with description ↵
  ServerDescription{address=192.168.99.100:27017, type=STANDALONE, state=CONNECTED, ok= ↵
  true, version=ServerVersion{versionList=[3, 2, 6]}, minWireVersion=0, maxWireVersion=4, ↵
  maxDocumentSize=16777216, roundTripTimeNanos=1473362}
may 15, 2016 6:11:10 PM com.mongodb.diagnostics.logging.JULLogger log
INFORMATION: Opened connection [connectionId{localValue:2, serverValue:2}] to ↵
  192.168.99.100:27017
may 15, 2016 6:11:10 PM com.mongodb.diagnostics.logging.JULLogger log
INFORMATION: Closed connection [connectionId{localValue:2, serverValue:2}] to ↵
  192.168.99.100:27017 because the pool has been closed.
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.601 sec - in ↵
  javacodegeeks.IntegrationTest

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[WARNING] File encoding has not been set, using platform encoding UTF-8, i.e. build is ↵
  platform dependent! The file encoding for reports output files should be provided by the ↵
  POM property ${project.reporting.outputEncoding}.
[INFO]
[INFO] --- docker-maven-plugin:0.15.0:stop (stop) @ javacodegeeks ---
[INFO] DOCKER> [mongo] : Stop and remove container a2a9d37cbbfd
[INFO]
[INFO] --- maven-failsafe-plugin:2.19.1:verify (verify) @ javacodegeeks ---
```

1.5 Download the source code

Download

You can download the full source code of this example here: [DockerTutorialForBeginners](#)

Chapter 2

Install Docker on Ubuntu Tutorial

Docker is, without any doubt, on of the most popular software at the moment, being used by almost every web developer and systems administrator. In this tutorial, we will see how to install it in Ubuntu systems.

For this tutorial, we will use Linux Mint 18 (an Ubuntu based Linux distribution) 64-bit version.

2.1 Requisites

According to Docker documentation, we must meet the following prerequisites for installing Docker:

- 64-bit distro. Obviously, we need a 64-bit microprocessor.
- At least, 3.10 kernel version. In Ubuntu, these versions are present since 13.10, so it shouldn't be a problem.

Just in case, we can check the kernel version with the following command:

```
uname -r
```

Which will show the kernel release. For this case, the output is:

```
4.4.0-21-generic
```

Now, let's see how we can install it.

2.2 Installation

We can install Docker simply via `apt-get`, without the need of adding any repository, just installing the `docker.io` package:

```
sudo apt-get update
sudo apt-get install docker.io
```

We can check that Docker has been successfully installed with executing the following:

```
docker -v
```

In this case, the following version has been installed:

```
Docker version 1.11.2, build b9f10c9
```

We can check if the Docker service was started:

```
docker.service - Docker Application Container Engine
  Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
  Active: active (running) since Thu 2016-09-22 12:39:12 CEST; 4min 13s ago
    Docs: https://docs.docker.com
 Main PID: 18883 (docker)
  CGroup: /system.slice/docker.service
          18883 /usr/bin/docker daemon -H fd://
          18891 containerd -l /var/run/docker/libcontainerd/docker-containerd.sock -- ↵
              runtime runc --start-timeout 2m
```

Finally, we can pull an image to check that Docker is working properly. For example, we will pull a BusyBox image:

```
docker pull busybox
```

And it should start to download the layers for the image.

Perhaps you got an error similar to the following:

```
Warning: failed to get default registry endpoint from daemon (Cannot connect to the Docker ↵
 daemon. Is the docker daemon running on this host?). Using system default: https://index ↵
 .docker.io/v1/

Cannot connect to the Docker daemon. Is the docker daemon running on this host?
```

If you got this error, don't worry, we will see how to fix it in the following subsection.

2.2.1 Fixing "Cannot connect to the Docker daemon" error

Actually, is not an error itself, is just that the user we are trying to execute Docker with, has no permission to access the Docker service.

This means that, if we try the same command with `sudo` permissions, we won't get the error:

```
sudo docker pull busybox
```

You will see that the image is being downloaded properly.

If we want to get rid of the `sudo` permissions for using Docker, we just have to add the user in question to the `docker` group, created by Docker during the installation. We can check that, effectively, the user we are working with is not in the group:

```
groups $(whoami)
```

And we will see that the `docker` group is not in the list. To add it, we can simply execute:

```
sudo usermod $(whoami) -G docker -a
```

(The `-G` option is for secondary groups; and the `-a` is for appending to the existing ones).

If we check again the groups, we will see that now the `docker` group is in the list. But, before anything, we have to restart the session, to make the change work.

After re-logging in again, we will see that we can run Docker without the need of root permissions:

```
docker pull busybox # It works!
```

2.3 Basic usage

Now that we have Docker running, we will see the most basic things we can do with it.

2.3.1 Pulling images

The Docker community is so big. There is a huge number of preconfigured images, many of them maintained by the official developers of the software in question (e.g. MySQL, Nginx, Jenkins, and a large etc.).

These images can be found at [Docker Hub](#). You don't have to create an account to pull the images from there, but you will need one if you want to host your images.

We already have seen how to pull images. For those official images, we just have to specify the name of the repository:

```
docker pull <repository>
```

For images of other users, we have to specify the username:

```
docker pull <username>/<repository>
```

By default, the latest image in the repository will be pulled (you have probably already the "Using default tag: latest" message if you have pulled the BusyBox image). But we can also pull a specific image, specified by a version tag. For that, we just have to append the tag in question after a colon:

```
docker pull <username>/<repository>:<tag>
```

For example:

```
docker pull busybox:1.25.0
```

Note: to see the images you have, you can use the `images` command:

```
docker images
```

2.3.2 Creating images from Dockerfiles

Creating files known as Dockerfiles is the way of creating our custom images, basing on an existing one.

It only consists on writing the steps to execute in a file.

2.3.3 Creating containers

We have seen how to pull images and create our own ones, but not how to use them.

When we use an image, i.e., we **instantiate** it, we create a container. We can have several containers from the same image. The concept is almost the same as with class and objects in Object Oriented Programming.

For creating containers, we use the `run` command. The easiest way is to execute it just specifying the image:

```
docker run busybox
```

Nothing seemed to happen. But, behind the scenes, Docker has created a container from the image, execute it, and then end it, as we didn't specify nothing to do.

We can execute a command inside the container:

```
docker run busybox echo "Hello world!"
```

Not very impressive, right? But imagine doing so with a virtual machine, and compare how much time takes each one.

We can also set an interactive shell with `-it` option:

```
docker run -it busybox
```

To see a more complete example, we are going to pull the Nginx image:

```
docker pull nginx
```

And then we are going to run it with the following command:

```
docker run -p 8080:80 -d nginx
```

And, if we follow `localhost:8080` in our browser, we will see that we have a dockerized Nginx web server!



Figure 2.1: Running Nginx in a Docker container.

Let's explain what we have done:

- With `-p 8080:80`, we have binded to the host's port 8080, the container's port 80.
- With `-d` option, we have detached the container to run it in the background, receiving its id.

With this example, you already should have noticed the powerfulness of Docker.

Note: To stop a container, we can execute the `stop` command followed by the container id:

```
docker stop <container-id>
```

2.4 Summary

In this tutorial we have seen the installation of Docker in Ubuntu and its very basic configuration. Apart from that, we have also seen the basic usage of Docker, pulling images from the Docker Hub and creating containers from these images.

Chapter 3

Docker Kernel Requirements

3.1 Introduction

Docker is a containerization technology that provides OS level virtualization to applications. It isolates processes, storage, networking, and also provide security to services running within it's containers. To enable this, Docker depends on various features of the Linux Kernel. Let us get introduced to these Docker kernel requirements in this post.

3.2 Docker engine dependencies from the Linux kernel

The dependencies on the Linux kernel can be broadly categorized into 4 classes - resource constraining, security, networking, and storage. Resource constraining features allow container creators to place restrictions on container environments like memory usage, cpu, etc.,. Security features allow security policies to be applied on containers. Networking features allow for the SDN networking features provided by Docker. Storage features allow Docker to support volumes, and various storage backends.

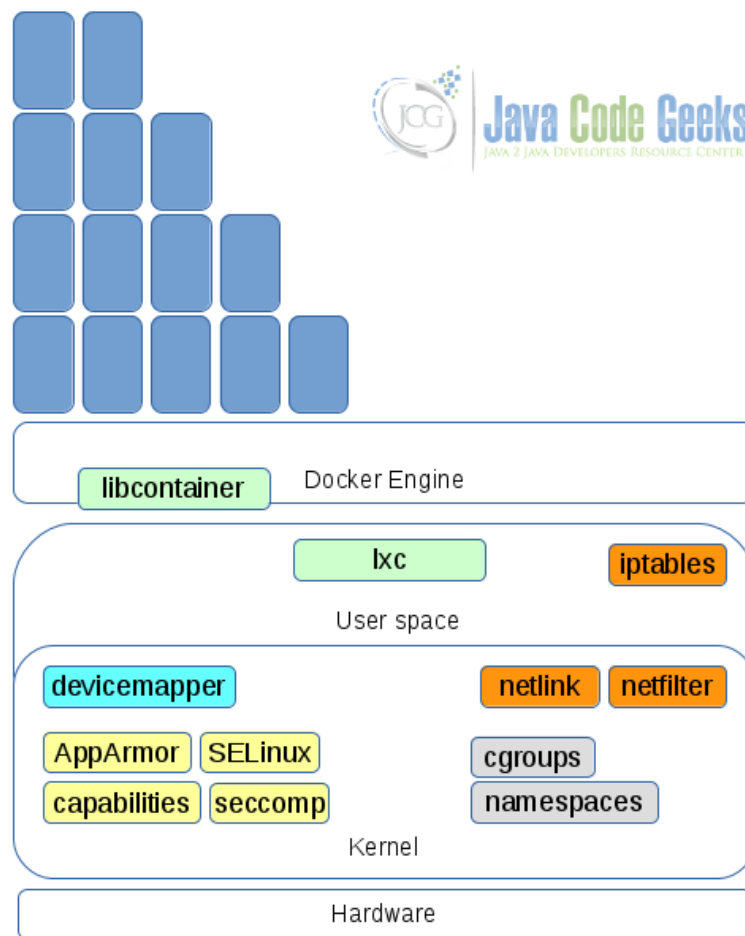


Figure 3.1: Kernel dependencies for Docker

Let us now examine each of these dependencies in brief.

3.3 Resource constraining dependencies

3.3.1 3.1 Control groups a.k.a cgroups

Control groups, or **cgroups**, is a kernel feature to constrain the resource usage of a process or a set of processes. This provides Docker with 4 main features:

- Limit resources (CPI, memory, network, disk I/O, ...) to user-defined processes.
- Prioritize resources to processes (a set of processes will get more resources than another set).
- Measure resource usage for billing purposes.
- Control a group of processes.

The `docker run` command is used to manipulate resources allocated to a container. For instance, `docker run --cpu-shares=<value>` sets the cpu share allocated to a container (every container gets 1024 shares by default). `docker run --cpuset-cpus=<value>` sets the CPU core on which the container would be run. Do look at [this insightful article](#) for some examples of manipulating cgroups settings for Docker containers.

3.3.2 Namespaces

Namespaces is a kernel feature that provides lightweight process virtualization to containers. This helps Docker to isolate these resources for a container - process IDs, hostnames, user IDs, network access, IPC and filesystems. Docker combines namespaces and cgroups to isolate resources for containers and place resource usage constraints. These namespaces are used to isolate containers - Process ID (pid), Network (net), Mount (mnt), Hostname (uts), Shared Memory (ipc).

- A pid namespace provides processes running within containers with separate pids isolated from other containers./li>
- A net namespace creates separate network interfaces, IP addresses and such for each container.
- A mnt namespace creates isolated mounts for each container. Mount points from host OS may be carried into the container but any any additions to the container mounts are not propagated back to the host.
- An uts namespace creates containers with their own hostnames without affecting other containers or the rest of the system.
- An ipc namespace creates isolated shared memory space for each container and prevents access between shared memory of different computers.

3.4 Security dependencies

3.4.1 AppArmor

AppArmor is a **Mandatory Access Control (MAC)** tool to restrict programs to a limited set of resources. Restriction policies are set in a simple text file to administer storage, networking, capabilities of a program. A policy can run in enforcement or complain mode. A policy running in enforcement mode will enforce the policy and report violations. A policy running in complain mode will not enforce restrictions but only report violations.

Docker installs a default AppArmor profile - `/etc/apparmor.d/docker` - during installation. This profile is applied to all Docker containers. To apply a specific AppArmor profile to a container use the option `docker run -it <container-name> --security-opt=apparmor=<profile-name>`.

Read [this page](#) for more details about Docker's usage of AppArmor.

3.4.2 Security Enhanced Linux a.k.a SELinux

SELinux, like AppArmor, enforces MAC policies on other subsystems of the Linux kernel. When compared to AppArmor, SELinux follows a more elaborate multi-level security policy control. This is currently developed and maintained by **RedHat**.

3.4.3 Posix capabilities a.k.a Capabilities

Capabilites as implemented in Linux (known as "**Posix Capabilities**") partitions the root user's privileges into distinct smaller units called "capabilities". These capabilities are enabled/disabled as a unit and assigned to individual threads. This allows a thread/process to perform some privileged operation with a minimal set of capabilities but without assuming superuser permissions. See `man capabilities` in any Linux system for more details on capabilities. Docker uses capabilities to restrict the actual capabilities of the container while providing all possible features to the service within it. A root user within a Docker container may not have all privileges as a root user in the actual host OS.

Read [this post](#) for more explanation on Docker's support of capabilities.

3.4.4 Secure Computing Mode a.k.a seccomp

Secure Computing Mode, also called **seccomp**, provides a facility to place filters on the system calls available to a user-defined process. This is combined with other tools to **provide a secure computing sandbox** to filter a thread from all available system calls. When seccomp is applied to a thread, the thread can perform only 4 system calls - `read()`, `write()`, `sigreturn()` and `exit()`. The kernel will kill the process if it uses any other system call.

A seccomp profile is set to a Docker container with the `security-opt` option of `docker run` like so:

```
$ docker run -it <container-name> --security-opt <value>
```

Read [this](#) doc for more information about Docker's use of seccomp.

3.5 Networking dependencies

3.5.1 Netfilter

Netfilter is a framework provided by the Linux Kernel that allows network packets flowing through the machine to be manipulated. Features include stateless and stateful packet filtering of IPv4 and IPv6 packets, Network address translation, port address translation, extensible APIs for 3rd party app developers. Docker uses Netfilter through its userspace counterpart IPTables.

3.5.2 IPTables

iptables is the user-space utility counterpart for netfilter. It interacts with netfilter and allows a system administrator to define tables of firewalling rules for packet filtering, network address translation (NAT), and so on. The Docker daemon automatically appends rules firewalling rules to iptables if it sees it installed in the system. For example when we expose a container's port to the outside world Docker adds a corresponding rule to iptables. To disable iptables, start Docker daemon with the option `iptables` set to `false` like so:

```
$ dockerd --iptables=false
```

See [this blog post](#) for a few examples of how Docker uses iptables.

3.5.3 Netlink

Netlink as a tool provides a mechanism for communication between kernel and userspace components using a socket interface. Even userspace components can use this to communicate among one another. This is an alternative to `ioctl` and reduces dependence on direct system calls, `ioctl` calls, and such. Docker implements its netlink libraries to talk to the kernel's netlink interface to create and configure network devices.

[This excellent post](#) has more details about how Docker uses Netlink.

3.6 File system dependencies

Docker supports several storage drivers with a plug-in architecture. One can choose a storage driver to run the Docker daemon with. However, Docker Engine can support only one active storage driver at a time. A change in the storage drive will need the Docker daemon to restart.

3.6.1 Device mapper

The **devicemapper framework** is provided by the kernel to map physical block devices as virtual devices devices. It provides the foundation for features such as logical volume management, device encryption, copy-on-write files, etc.,. Docker uses this framework to support copy-on-write files in containers.

3.7 Other non-kernel dependencies

3.7.1 LibContainer a.k.a RunC

libcontainer (Now called **opencontainers/RunC**) - This is not exactly a kernel feature. Docker developed this as an execution engine that exposes a consistent standardized Go API to work with Linux namespaces, cgroups, capabilities, AppArmor, security profiles, network interfaces, firewalls and firewalling rules. RunC has replaced LXC as the default execution driver of the Docker Engine.

3.7.2 LXC

Like libcontainer, **LXC** provides a userspace interface for the Linux Kernel's container supporting features. LXC was the initial execution engine before Docker moved to RunC.

3.8 Summary

In this post we were introduced to the key kernel features on which Docker depends and builds to enable containerization. Each of the Kernel features in itself can be pursued further and understood more deeply to **improve container security** in Docker. The **Docker docs** also contain more information about kernel level enablers.

Chapter 4

Docker Hello World Example

In this example we will explore different ways of running basic "Hello, World" containers in Docker.

4.1 Introduction

Creating a Docker Hello World container and getting it to work verifies that you have the Docker infrastructure set up properly in your development system. We will go about this in 4 different ways:

- Run Docker hello world image provided by Docker
- Get a "Hello, world" printed from another basic Docker image
- Write a Simple "Hello, World" Program in Java and Run it Within a Docker Container
- Execute a "Hello, World" Program in Java Using Gradle and Docker

You should have Docker installed already to follow the examples. Please go to the Docker home page [to install](#) the Docker engine before proceeding further, if needed. On Linux, please do ensure that your user name is added into the "docker" group while installing so that you need not invoke sudo to run the Docker client commands. You should also have Java and Gradle installed to try examples 3 and 4. Install your favorite J2SE distribution to get Java and install Gradle from [gradle.org](#).

So let us get started!

4.2 Run Docker Hello World image provided by Docker

This is the simplest possible way to verify that you have Docker installed correctly. Just run the [hello-world](#) Docker image in a terminal.

```
$ docker run hello-world
```

This command will download the `hello-world` Docker image from the Dockerhub, if not present already, and run it. As a result, you should see the below output if it went well.

```
File Edit View Search Terminal Help
sh-4.2$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world

c04b14da8d14: Pull complete
Digest: sha256:0256e8a36e2070f7bf2d0b0763dbabdd67798512411de4cdc9431a1feb60fd9
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
https://hub.docker.com

For more examples and ideas, visit:
https://docs.docker.com/engine/userguide/

sh-4.2$
```




Figure 4.1: Run Docker Hello-World Image

4.3 Get a "Hello, world" Printed from Another Basic Docker Image

This too is simple. Docker provides a few **baseimages** that can be used directly to print a "Hello, World". This can be done as shown below and it verifies that you have a successful installation of Docker.

```
$ docker run alpine:latest "echo" "Hello, World"
```

This command downloads the **Alpine** baseimage the first time and creates a Docker container. It then runs the container and executes the echo command. The echo command echoes the "Hello, World" string. As a result, you should see the output as below.

```
File Edit View Search Terminal Help
sh-4.2$ docker run alpine:latest "echo" "Hello, World"
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine

Digest: sha256:1354db23ff5478120c980eca1611a51c9f2b88b61f24283ee8200bf9a54f2e5c
Status: Downloaded newer image for alpine:latest
Hello, World
sh-4.2$
```




Figure 4.2: Use the Alpine Docker Image to Print Hello World

4.4 Write a Simple "Hello, World" Program in Java and Run it Within a Docker Container

Let us take it up a notch now. Let us write a simple hello world program in Java and execute it within a Docker container.

4.4.1 Create HelloWorld.java

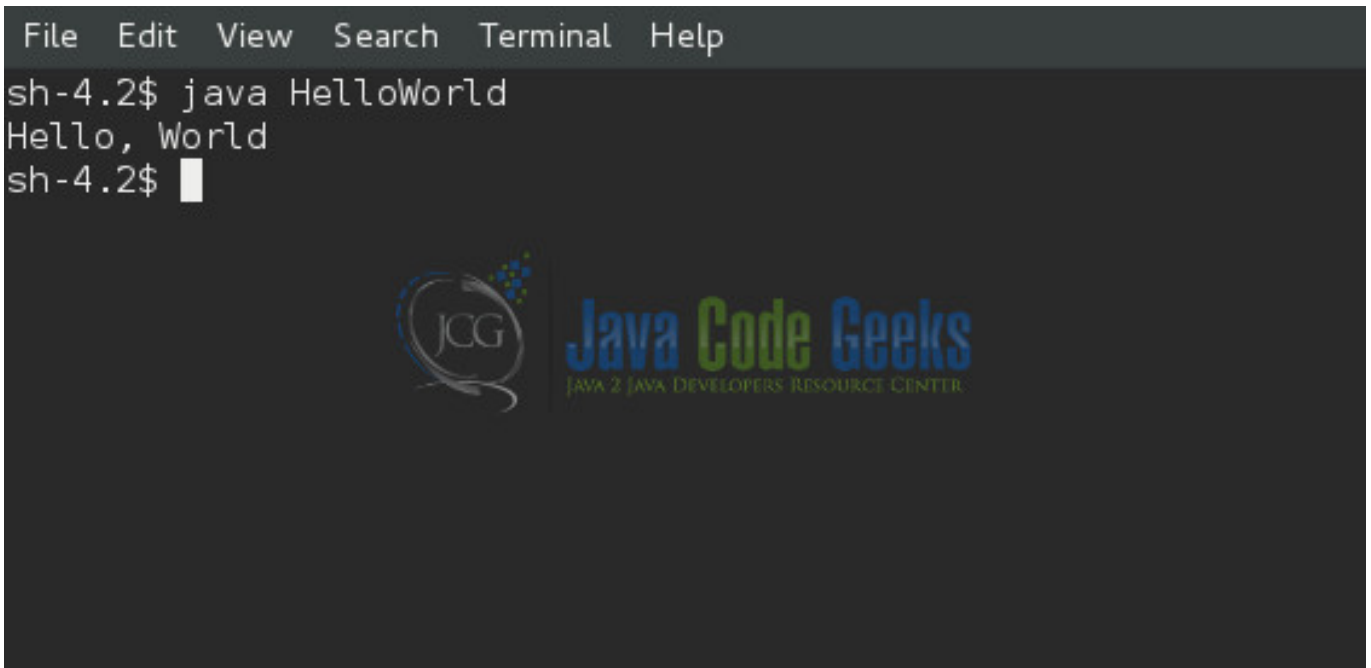
First of all, let us create a simple Java program that prints "Hello, World". Open up your favorite text editor and enter the following code:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

This is a standard HelloWorld program in Java very closely resembling the one [provided in the Official Java tutorial](#). Save the file as `HelloWorld.java`. Now, compile this file using the Java compiler.

```
$ javac HelloWorld.java
```

This should create the class file `HelloWorld.class`. Normally, we would use the `java` command to execute HelloWorld as below



```
File Edit View Search Terminal Help
sh-4.2$ java HelloWorld
Hello, World
sh-4.2$
```

The image shows a terminal window with a dark background. At the top, there is a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. Below the menu bar, the terminal prompt 'sh-4.2\$' is followed by the command 'java HelloWorld'. The output of the command is 'Hello, World'. The prompt 'sh-4.2\$' is shown again with a cursor. Below the terminal window, there is a logo for 'Java Code Geeks' which includes the letters 'JCG' in a circle and the text 'Java Code Geeks' and 'JAVA 2 JAVA DEVELOPERS RESOURCE CENTER'.

Figure 4.3: Run HelloWorld.main() in Your Native OS

But we want to run this from within a Docker container. To accomplish that we need to create a Docker image. Let us do that now.

4.4.2 Create a Dockerfile

To create a new Docker image we need to create a Dockerfile. A Dockerfile defines a docker image. Let us create this now. Create a new file named `Dockerfile` and enter the following in it and save it.

```
FROM alpine:latest
ADD HelloWorld.class HelloWorld.class
RUN apk --update add openjdk8-jre
ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom", "HelloWorld"]
```

The complete Dockerfile syntax can be found from [Docker docs](#) but here is briefly what we have done. We extend our image from the [Alpine baseimage](#). We next added `HelloWorld.class` into the image with the same name. Later we installed a JRE environment using [OpenJDK](#). Finally, we gave the command to execute when this image is run - that is to run our HelloWorld in the JVM.

4.4.3 Create Docker Hello World Image and Run it

Now, build an image from this Dockerfile by executing the below command.

```
$ docker build --tag "docker-hello-world:latest" .
```

As a result of this you should see the following output


```
File Edit View Search Terminal Help
sh-4.2$ docker build --tag "docker-hello-world:latest" .
Sending build context to Docker daemon 4.096 kB
Step 1 : FROM alpine:latest
--> baa5d63471ea
Step 2 : ADD HelloWorld.class HelloWorld.class
--> Using cache
--> 44a4cc613c43
Step 3 : RUN apk --update add openjdk8-jre
--> Using cache
--> 392eefc0aad3
Step 4 : ENTRYPOINT java -Djava.security.egd=file:/dev/./urandom HelloWorld
--> Running in 051e53de6d98
--> 7d363cc7ff0b
Removing intermediate container 051e53de6d98
Successfully built 7d363cc7ff0b
sh-4.2$ █
```




Figure 4.4: Build Docker Image Called docker-hello-world

Finally, run the Docker image to see the "Hello, World" printed.

```
$ docker run docker-hello-world:latest
```

As a result of this you should see the below output

```
File Edit View Search Terminal Help
sh-4.2$ docker run docker-hello-world:latest
Hello, World
sh-4.2$ █
```




Figure 4.5: Run the Docker Image docker-hello-world

That's it. This verifies that you can create your own custom Docker images too and run them.

4.5 Execute a "Hello, World" Program in Java Using Gradle and Docker

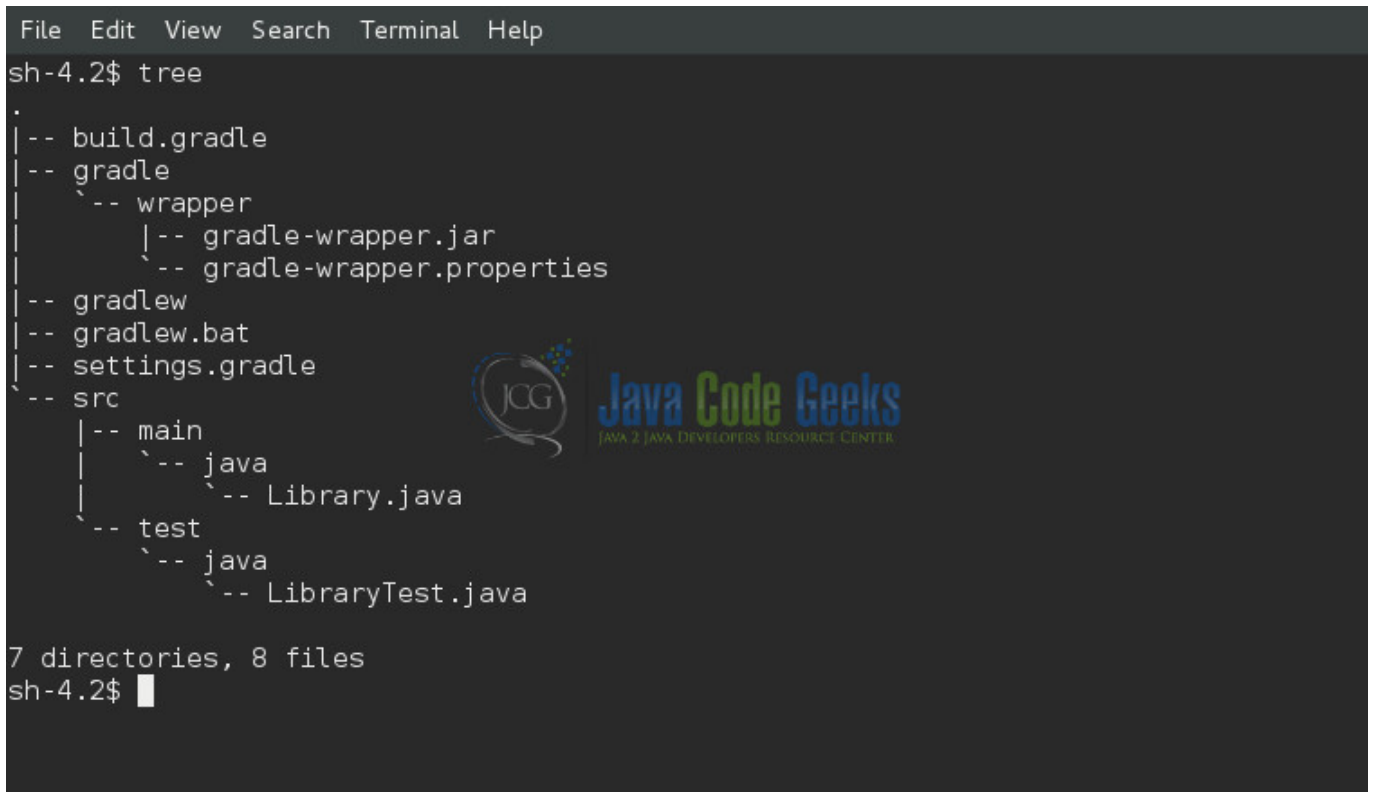
Let us take it up another notch now. Real world programs are more complex than the ones shown above. So let us create another hello world Java program and run it in a Docker container using a few best-practices of the Java ecosystem. Furthermore, let us set up a Java project using Gradle, add the Docker plugin for Gradle into it, create a Docker image and run it using Gradle.

4.5.1 Initialize a New Java Project and Create HelloWorld.java

First of all, Create a new folder called "docker-hello-world-example", open a terminal and change into this folder. Next, use Gradle to initialize a new Java project.

```
$ gradle init --type java-library
```

Once done you should see the following folder structure created. You can freely delete the files `Library.java` and `LibraryTest.java`. We will not be needing them.



```
File Edit View Search Terminal Help
sh-4.2$ tree
.
|-- build.gradle
|-- gradle
|   |-- wrapper
|   |   |-- gradle-wrapper.jar
|   |   |-- gradle-wrapper.properties
|-- gradlew
|-- gradlew.bat
|-- settings.gradle
|-- src
|   |-- main
|   |   |-- java
|   |   |-- Library.java
|   |-- test
|   |   |-- java
|   |   |-- LibraryTest.java
7 directories, 8 files
sh-4.2$
```

Figure 4.6: Folder structure for the Sample Java Project Created by Gradle

Now, create a new file `src/main/java/com/javacodegeeks/examples/HelloWorld.java` and enter the following in it. Save it once done.

```
package com.javacodegeeks.examples;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}
```

4.5.2 Create a Dockerfile

Next, create a Docker file in `src/main/resources` called `"src/main/resources/Dockerfile"` and enter the following. Save and close the file once done.

```
FROM alpine:latest
RUN apk --update add openjdk8-jre
COPY docker-hello-world-example.jar app.jar
ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/./urandom", "-jar", "app.jar"]
```

Let us quickly go through what we did in the above Dockerfile. We derive our image from the Alpine Linux base image. Next, we installed J2SE from OpenJDK. Next, we copied the jar file generated by the build process as `app.jar` into the Docker image. Finally, we set the command to be run when the Docker container is run which is to execute the `app.jar` file.

4.5.3 Include Gradle-Docker Plugin

Next, make the following changes into the `build.gradle` file to update the Jar manifest.

```
mainClassName = 'com.javacodegeeks.examples.HelloWorld'

jar {
    manifest {
        attributes(
            'Main-Class': mainClassName
        )
    }
}
```

We set the `mainClassName` to `com.javacodegeeks.examples.HelloWorld` and set the same attribute to be inserted into the manifest file of the jar that will be created.

Next, insert the following changes at the very top of `build.gradle` file to configure and add the Gradle plugins for Docker.

```
buildscript {
    repositories {
        maven {
            url "https://plugins.gradle.org/m2/"
        }
    }
    dependencies {
        classpath "gradle.plugin.com.palantir.gradle.docker:gradle-docker:0.9.1"
    }
}

plugins {
    id 'com.palantir.docker' version '0.9.1'
    id 'com.palantir.docker-run' version '0.9.1'
}

apply plugin: 'com.palantir.docker'
```

Here we basically added the Gradle plugins called `docker` and `docker-run` provided by **Palantir**. These plugins enable us to create Docker containers and run them using Gradle. We added the URL to tell where to find the `gradle-docker` plugin (`https://plugins.gradle.org/m2/`). We also added the plugins into the classpath as build dependencies.

4.5.4 Configure the Docker Tasks in Gradle Build File

Next, insert the following changes at the bottom of `build.gradle` file to configure the details of the Docker image.

```
docker {  
    name 'com.javacodegeeks.examples/docker-hello-world:1'  
    tags 'latest'  
    dockerfile 'src/main/docker/Dockerfile'  
    dependsOn tasks.jar  
}
```

Here we configured the details of the docker image. We assigned the image name `com.javacodegeeks.examples/docker-hello-world` with version 1. We assigned it the tag `latest` and gave the path where to find the Docker file. Finally, we made it depend on the Gradle task `jar` so that the output of the `jar` task will be fed to this task.

Finally, insert the following changes at the bottom of `build.gradle` file to configure the details of the Docker container.

```
dockerRun {  
    name 'docker-hello-world-container'  
    image 'com.javacodegeeks.examples/docker-hello-world:1'  
    command 'java', '-Djava.security.egd=file:/dev/./urandom', '-jar', 'app.jar'  
}
```

Here we configured the details of the docker container. We assigned the container name `"docker-hello-world-container"`. We assigned it the image from where to create the container and gave it a command to execute when running the container.

4.5.5 Create Docker Container and Run it

Now we are ready to build and run this. Use Gradle to build and run this program

```
$ ./gradlew clean build docker dockerRun
```

You will notice that the code was built, jar was created, Docker image and containers were created, and the container was run too. However, there is no "Hello, World" actually printed on the screen here.

```
File Edit View Search Terminal Help
sh-4.2$ ./gradlew clean build docker dockerRun
Starting a Gradle Daemon (subsequent builds will be faster)
:clean
:compileJava
:processResources UP-TO-DATE
:classes
:jar
:startScripts
:distTar
:distZip
:dockerfileZip
:assemble
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:check UP-TO-DATE
:build
:dockerClean UP-TO-DATE
:dockerPrepare
:docker
Sending build context to Docker daemon 245.8 kB
Step 1 : FROM alpine:latest
--> baa5d63471ea
Step 2 : RUN apk --update add openjdk8-jre
--> Using cache
--> 2b4b741d1723
Step 3 : COPY docker-hello-world-example.jar app.jar
--> da7f4534b8f0
Removing intermediate container 896d167763e9
Step 4 : ENTRYPOINT java -Djava.security.egd=file:/dev/./urandom -jar app.jar
--> Running in b981f0cb206a
--> c074699e538d
Removing intermediate container b981f0cb206a
Successfully built c074699e538d
:dockerRun
0a68b3942fcd6333f4bc13e013931402ea3f3e31d0214d58c28f0d589c1fd05b
:dockerRunStatus
Docker container 'docker-hello-world-container' is RUNNING.

BUILD SUCCESSFUL

Total time: 13.492 secs
sh-4.2$
```




Figure 4.7: Run the Gradle Tasks clean, build, docker, and dockerrun

The last line simply says that the Docker container *docker-hello-world-container* is running. To see if it printed "Hello, World" successfully, we need to check the logs created by the Docker container. This can be checked by executing the following command.

```
$ docker logs docker-hello-world-container
```

```
File Edit View Search Terminal Help
sh-4.2$ docker logs docker-hello-world-container
Hello, World
sh-4.2$
```

The image shows a terminal window with a dark background. At the top, there is a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. Below the menu bar, the terminal prompt is 'sh-4.2\$'. The user has entered the command 'docker logs docker-hello-world-container'. The output of the command is 'Hello, World'. Below the terminal output, there is a logo for 'Java Code Geeks' which includes the text 'JCG' in a circle and 'Java Code Geeks' in a stylized font, with 'JAVA 2 JAVA DEVELOPERS RESOURCE CENTER' written below it.

Figure 4.8: Check the Docker Container Logs to Check if Hello world was Printed

There it is!

4.6 Summary

In this example we learned how to create Docker Hello world type containers in 4 different ways:

- First, we learned how to run the hello-world image provided by Docker. This printed Hello World as soon as it was run
- Next, we learned how to run the Alpine image and print a Hello World using the echo shell command
- Next, we learned how to get the basic Hello World program into a Docker image and run it to print Hello World. The source code for this can be downloaded from the links given below.
- Finally, we learned how to set up a structured Java project using Gradle and generate Docker images and Containers using Gradle plugins. We used this structure to create a Hello World Java project and got it to run in a container. Later, we verified the Docker logs for the container to see that Hello World was printed when the container was run. The source code for this also can be downloaded from the links given below.

4.7 Download the Source Code

Download

The source code for example 3 can be downloaded here: [docker-hello-world.zip](#)

The source code for example 4 can be downloaded here: [docker-hello-world-example.zip](#)

Chapter 5

Docker as a Service

In the previous posts we learned about [Docker](#), [containers](#), and [working with containers](#), In this post we will learn about Containers as a Service (CaaS) in general and how Docker realizes CaaS.

5.1 Why Docker as a Service?

Take a typical developer work-flow involving Docker. It will be typically as below. This is based on the reference workflow provided in the [Docker docs](#).

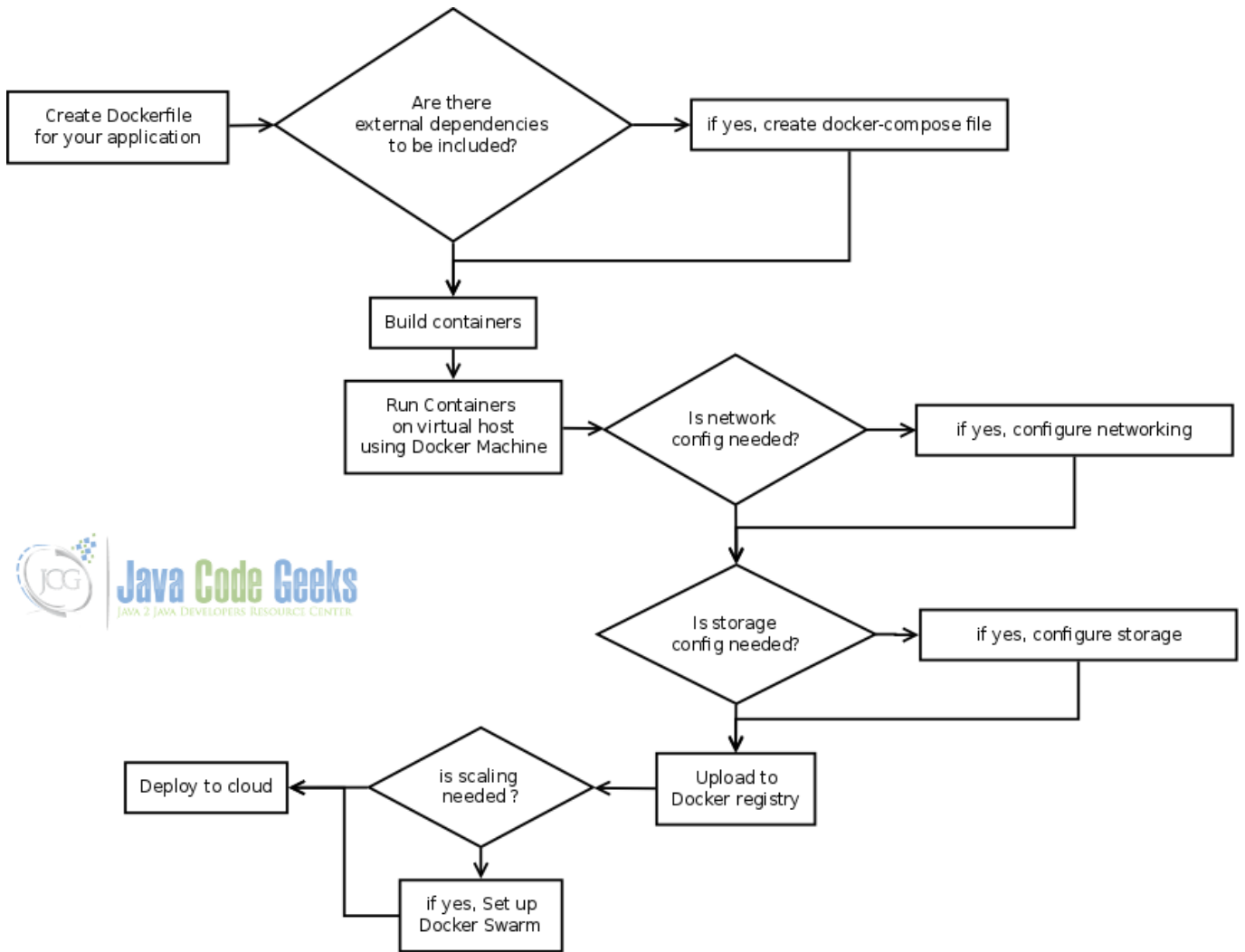


Figure 5.1: A typical Docker workflow

As you can see from the above workflow, there is quite some IT activity to be performed if Docker is used for app development at scale in an organization.

- Infrastructure to host Docker containers and all the other related tools provided by Docker.
- Provide compute, storage and networking for the containers as needed.
- Application lifecycle support and tooling.
- Monitor utilization of compute, storage and networking and stop/start containers as needed.
- Address scaling needs of the containers. Handle orchestration of containers at enterprise scale.

These issues are significant at enterprise scale. The existing solution has been for administrators to use infrastructure-as-a-service (IaaS) like Amazon’s AWS EC2 and handle the above issues manually or through home-built tools. However, Docker now provides the **Docker container-as-a-service** to address the above issues.

5.2 Container as a Service (CaaS)

So, what is CaaS? It is a cloud service model that enables users to order and use containerization infrastructure from a cloud provider using a pay-as-you-go model. Docker CaaS is an example. **Amazon EC2 Container Service (ECS)** by AWS is another

example.

As a cloud service model, CaaS probably sits as a subset of IaaS in that it enables a container-based infrastructure for its users. A CaaS offering usually seeks to provide the following features to its users.

- Manage containers through an API or web interface.
- Monitor compute, storage and network resources used by the containers.
- Provide cluster management and orchestration capabilities for containers.
- Provide security and governance controls.
- Optionally, be cloud agnostic so that the CaaS can be shifted across cloud providers or from in-premises to cloud.

5.3 Docker CaaS

Docker's CaaS offering called [Docker Datacenter](#) packages Docker's tools to provide developers and IT with a consistent container infrastructure. This is a commercial offering aimed at enterprises and provides the following features.

- Brings well known open source Docker tools under a CaaS umbrella - [Docker Engine](#), [Docker Compose](#), [Docker registry](#), [Docker Swarm](#).
- Adds commercial offerings - Universal Control Pane and Trusted Registry.
- Ability to deploy the CaaS in-premises or cloud.
- A private trusted docker registry for managing images.
- Cluster management and orchestration capabilities through Swarm and Universal Control Pane.
- Tools for the application development lifecycle, continuous integration and deployment.
- Provides security in the application lifecycle all the way from from dev to production stages.

Refer to the [Docker Datacenter](#) pages by Docker for more information about providing Docker as a service.

5.4 Summary

In this short post, we understood the need for a "Docker-as-a-service" solution to handle containerized application development needs at scale for an organizations. We briefly understood the features provided by a CaaS service in general. Then we briefly discussed Docker's CaaS offering called the Docker Datacenter that offers Dockers tools as a service to developers and IT administrators.

Chapter 6

Docker Build Example

6.1 Introduction

Docker is a tool to avoid the usual headaches of conflicts, dependencies and inconsistent environments, which is an important problem for distributed applications, where we need to install or upgrade several nodes with the same configuration.

Docker is a container manager, which means that is able to create and execute containers that represent specific runtime environments for your software. An specific runtime environment is called *Docker image* and containers are specific executions of a Docker images. It is the same concept that classes and objects/instances, which allow to define, for example, different network configurations for each container.

Dockerhub is a repository of docker images. Docker users can upload their docker images and share them with the community. It allows that users of software providers do not need to deal with installation guides. They just need to download the image and create as many containers they need. In fact, it is specially useful when system administrators need to scale the same software in a cluster of machines.

The purpose of this tutorial is to explain how to build docker images using the **docker build** command and the supported building options. As we will see below, docker allows to build images using different approaches.

6.2 Installation & Usage

The docker build command is available from your docker installation, which depends on your operative system and are widely explained ([here](#) for mac, [here](#) for windows and [here](#) for ubuntu). Once, you have followed the installation instructions, you should be able to run the docker build command.

Docker usage command

```
docker build --help
```

It prints the following output:

```
bash-3.2$ docker build --help
```

```
Usage: docker build [OPTIONS] PATH | URL | -
```

```
Build a new image from the source code at PATH
```

```
-c, --cpu-shares=0          CPU shares (relative weight)
--cgroup-parent=           Optional parent cgroup for the container
--cpu-period=0             Limit the CPU CFS (Completely Fair Scheduler) period
--cpu-quota=0             Limit the CPU CFS (Completely Fair Scheduler) quota
--cpuset-cpus=            CPUs in which to allow execution (0-3, 0,1)
--cpuset-mems=            MEMs in which to allow execution (0-3, 0,1)
--disable-content-trust=true Skip image verification
-f, --file=               Name of the Dockerfile (Default is 'PATH/Dockerfile')
--force-rm=false          Always remove intermediate containers
--help=false              Print usage
-m, --memory=            Memory limit
--memory-swap=           Total memory (memory + swap), '-1' to disable swap
--no-cache=false         Do not use cache when building the image
--pull=false             Always attempt to pull a newer version of the image
-q, --quiet=false        Suppress the verbose output generated by the containers
--rm=true                Remove intermediate containers after a successful build
-t, --tag=               Repository name (and optionally a tag) for the image
--ulimit=[]              Ulimit options
```



Figure 6.1: dockerbuild

The `docker build` command builds Docker images from a Dockerfile and a *context*. In order to specify a context you need to reference a local file directory or a URL. The contents of this context can be referenced from ADD instructions of the Dockerfile to reference a file in the context.

6.3 Context definition

The context of a `docker build` command could be a local directory (`PATH`), a Git repository (`URL`) or the standard input (`-`). This section explains the usage scenarios of these different contexts.



Figure 6.2: docker-build-alternatives

6.3.1 Build with PATH

Probably, you are assuming that the docker daemon is always running in your machine. However, when you are building docker images for a remote machine, this command is sending to the daemon a tar file with the contents of the local directory you have specified to proceed with the build command. For example:

Docker build from path[source.java]

```
docker build .
```

Sends the contents of the execution directory to the docker daemon.

6.3.2 Build with URL

URLs are specially designed for Git repositories. In this case, the build command clones the Git repository and use the cloned repository as context. The Dockerfile at the root of the repository is used as Dockerfile. Alternatively to http or https, you can specify an arbitrary Git repository by using the `git://` or `git@` schema. For a GitHub repository, we can use docker build as follows:

Docker build from git

```
docker build github.com/myuser/project
```

6.3.3 Build with -

This will read a Dockerfile from STDIN without context. It is commonly used to send a compressed file. The Supported formats are: bzip2, gzip and gz. For example:

Docker build from the standard output

```
docker build - < context.tar.gz
```

6.4 Dockerfile location

The dockerfile location, is by default in the root directory of the specified context and it is called *Dockerfile*. However, you can specify another one with the `-f` option, but always it must be located inside the build context. For example:

Docker build with an specific dockerfile

```
docker build -f Dockerfile.debug .
```

6.5 Example

Now, let's build an Ubuntu image with Java from a Github project used in a [previous JavaCodeGeeks example](#). Run the following command:

Docker build command with a real project

```
docker build -t javacodegeeks/buildsample github.com/rpau/docker-example4j
```

Now, in order to check if the image is available, execute `docker images`. This command shows the available images for your docker daemon. It should show something similar to the following output:

```
bash-3.2$ docker images
REPOSITORY          TAG          IMAGE ID          CREATED          VIRTUAL SIZE
javacodegeeks/buildsample  latest      3c33d4cb5c8f     6 minutes ago   961.2 MB
```

The logo for Java Code Geeks, featuring a circular emblem with 'JCG' and the text 'Java Code Geeks' and 'JAVA 2 JAVA DEVELOPERS RESOURCE CENTER' below it.

Figure 6.3: dockerimages

6.6 Conclusion

This tutorial explains what is a docker image and how to build it using the command `docker build`. This command requires to specify a context, which is the location of the files that we need to add into the image and also the Dockerfile. Finally, our example shows how to build an image from a Github project and how to verify what are the available images in our docker daemon with the `docker images` command.

Remember that creating docker images is extremely useful to set up an specific execution environment that needs to be installed in a cluster of machines. Therefore, it allows to avoid problems caused by differences between production and development environments.

Chapter 7

Docker Compose example

7.1 Introduction

Docker Compose is a tool for defining and running multi-container Docker applications. With Compose, you define in a configuration file the set of docker containers that application requires on a specific machine. Then, using a single command, you create and start all the services in a single host.

Docker Compose is specially useful for the following use case scenarios:

- **Create a development environment** with all the required services started only using your own machine. In other words, stop to prepare your development environment manually over and over again in your own or different machines.
- **Automate test environments** with the same characteristics than the production environments and run integration tests.
- **Single hosts deployments**- that is use a machine with different services as docker containers. The most recommended way to do so is using the Docker machine or Docker swarm because you can launch docker compose remotely from your laptop in a safe way without requiring SSH.

To use Docker Compose, you just need to create configuration file called `docker-compose.yml`, which contains the specification to create and run a set of docker containers. Let's try to create an example with a web application that uses a redis database server.



```
$ docker-compose up
Pulling image redis...
Building web...
Starting composable_redis_1...
Starting composable_web_1...
redis_1 | [8] 02 Jan 18:43:35.576 # Server started,
Redis version 2.8.3
web_1 | * Running on http://0.0.0.0:5000/
$|
```

The image shows a terminal window with the output of the `docker-compose up` command. The output indicates that the Redis image is pulled, the web application is built, and both containers are started. The Redis container logs show the server starting and the version (2.8.3). The web container logs show it is running on `http://0.0.0.0:5000/`. To the right of the terminal is a cartoon octopus holding several blue blocks. At the bottom left is the Java Code Geeks logo.

Figure 7.1: docker-compose

7.2 Docker Toolbox setup

Docker compose is one of the included tools into the [Docker toolbox](#). To install Docker toolbox, you just need to download the appropriate installer from [here](#) for your operative system and execute it.

Docker Toolbox contains the following tools:

- Docker Compose : to create runtime environments with multiple docker containers in one machine.
- Docker Machine : to dynamically run docker commands to remote machines or in the local/default one.
- Docker Kitematic : to Build and run containers through a graphical user interface (GUI).
- VirtualBox : to run Docker.
- Docker client: to create and run docker containers.

Once you have finished the installation procedure, you should be able to run `docker-compose --help`, which prints the accepted subcommands.

7.3 Docker compose example: An HTTP server connected to Redis

This example is about a web application for create a TODO list using Redis with basically a three steps:

- Define how to run the web and database docker images in `docker-compose.yml` so they can be run together in an isolated environment.
- Create the web application and its `Dockerfile` to create the runtime environment anywhere.
- Lastly, run `docker-compose up` and Compose will start and run your entire app.

7.3.1 Creating the `docker-compose.yml` file

Create an empty project called `todos` and copy the following `docker-compose.yml` inside the project directory.

`docker-compose.yml`

```
web:
  build: .
  ports:
    - "8000:8000"
  links:
    - redis
redis:
  image: redis
```

This file specifies two containers: `web`, which contains our Java code to launch a web server and `redis` server. Notice that in the first case, appears the `build` property, whereas the second case, the `image` property. It means that in order `web` container needs to build a Docker image that appears in our local file system. However, for the `redis` server, the image is downloaded from the DockerHub repository.

With this `docker-compose.yml` file, Compose simulates two machines with different IP addresses in a same local network because they are linked (i.e `link` property). However, only the `web` is externally through the 8000 of our docker-machine and Docker creates a binding between this port and the container 8000 port.

7.3.2 Creating the web server

Now, let's create the `Dockerfile` that Compose requires in the same directory than the `docker-compose.yml` file with the following contents.

`Dockerfile`

```
FROM ubuntu:14.04
MAINTAINER javacodegeeks

RUN apt-get update && apt-get install -y python-software-properties software-properties-↵
  common
RUN add-apt-repository ppa:webupd8team/java

RUN echo "oracle-java8-installer shared/accepted-oracle-license-v1-1 boolean true" | ↵
  debconf-set-selections

RUN apt-get update && apt-get install -y oracle-java8-installer maven

ADD . /usr/local/todolist
RUN cd /usr/local/todolist && mvn assembly:assembly
CMD ["java", "-cp", "/usr/local/todolist/target/todolist-1.0-jar-with-dependencies.jar", "↵
  com.javacodegeeks.todolist.TODOServer"]
```

This `Dockerfile` builds our `todolist` project with Maven and starts an specific Java class. Thus, it is necessary to convert our `todolist` project into a Maven project. To do so, copy the following `pom.xml` in the project directory (the same place than `Dockerfile` and `docker-compose.yml`).

`pom.xml`


```
<project xmlns="https://maven.apache.org/POM/4.0.0" xmlns:xsi="https://www.w3.org/2001/ ↵
  XMLSchema-instance"
  xsi:schemaLocation="https://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd ↵
    /maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.javacodegeeks</groupId>
  <artifactId>todolist</artifactId>
  <version>1.0</version>
  <dependencies>
    <dependency>
      <groupId>biz.paluch.redis</groupId>
      <artifactId>lettuce</artifactId>
      <version>3.3.2.Final</version>
    </dependency>
    <dependency>
      <groupId>commons-io</groupId>
      <artifactId>commons-io</artifactId>
      <version>2.4</version>
    </dependency>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.6.3</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>2.6</version>
        <configuration>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</ ↵
              descriptorRef>
          </descriptorRefs>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

This POM file uses lettuce (a Java Redis client) and jackson and commons-io as a utility libraries for coding basic HTTP server in only one Java class. Create this class with the following code:

src/main/java/com/javacodegeeks/todolist/ToDoServer.java

```
package com.javacodegeeks.todolist;

import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.List;

import org.apache.commons.io.IOUtils;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.lambdaworks.redis.RedisClient;
import com.lambdaworks.redis.RedisConnection;
import com.lambdaworks.redis.ValueScanCursor;
import com.sun.net.httpserver.HttpExchange;
```

```

import com.sun.net.httpserver.HttpHandler;
import com.sun.net.httpserver.HttpServer;

public class TodoServer {

    public static void main(String[] args) throws Exception {
        HttpServer server = HttpServer.create(new InetSocketAddress(8000), 0);
        server.createContext("/", new MyHandler(System.getenv("REDIS_PORT")));
        server.setExecutor(null); // creates a default executor
        server.start();
    }

    static class MyHandler implements HttpHandler {

        private RedisClient redisClient;
        private RedisConnection connection;
        private ObjectMapper mapper;

        public MyHandler(String redisURL) throws MalformedURLException {

            String hostPortURL = redisURL.substring("tcp://".length());
            int separator = hostPortURL.indexOf(':');
            redisClient = new RedisClient(hostPortURL.substring(0, separator),
                Integer.parseInt(hostPortURL.substring(separator + ←
                1)));
            connection = redisClient.connect();
            mapper = new ObjectMapper();
        }

        public void handle(HttpExchange t) throws IOException {
            String method = t.getRequestMethod();
            OutputStream os = t.getResponseBody();
            String response = "";

            if (t.getRequestURI().getPath().equals("/todos")) {
                if (method.equals("GET")) {
                    ValueScanCursor cursor = connection.sscan("todos");
                    List tasks = cursor.getValues();
                    response = mapper.writeValueAsString(tasks);
                } else if (method.equals("PUT")) {

                    connection.sadd("todos", IOUtils.toString(t. ←
                    getRequestBody()));
                }
            }

            t.sendResponseHeaders(200, response.length());
            os.write(response.getBytes());
            os.close();
        }

        @Override
        public void finalize() {
            connection.close();
            redisClient.shutdown();
        }
    }
}

```

Notice that the `TodoServer` is assuming that the Redis connection URL appears in an environment variable called `REDIS_PORT`.

However, how to know the names of the available environment variables? Docker defines an standard way to do so [here](#).

7.3.3 Lauching Docker Compose

All the elements for the example are ready now. So, open your prompt, go to the project directory and run:

Docker-Compose up command

```
docker-compose up
```

Voila! At this moment, we have an HTTP server connected with Redis in our default docker-machine. In order to discover the IP of the default docker-machine, run the following command.

Docker-Machine ip command

```
$docker-machine ip default
192.168.99.100
```

7.3.4 Testing the web server

Using the output of the previous command, we can test the web server using the `curl` command. The available options are:

Create new tasks:

CURL command to create new tasks

```
curl -X PUT --data "new task" https://192.168.99.100:8000/todos
```

List all tasks:

CURL command to List all tasks

```
curl -X GET https://192.168.99.100:8000/todos
```

7.4 Download the complete source code

This is an example of how running a Java program with Docker Compose.

Download

You can download the full source code of this example here: [docker-compose example](#)

Chapter 8

Configuring DNS in Docker

8.1 Introduction

This post introduces Docker Engine's network feature in general and specifically introduces configuring DNS in containers. This post assumes that you have the Docker Engine installed and that you know the [basics](#) of working with [containers](#). We will not discuss service discovery from other networks here since that needs a deeper knowledge of Docker tools like Docker Swarm. Since these posts focus on the basics of Docker we will focus on networking containers within the same host. So let's get started with understanding the basics of networking in Docker.

8.2 Basics of Networking Configurations in Docker Engine

The Docker Engine provides 3 networks by default - `bridge`, `host` and `none`. The command `docker network ls` lists out all networks created by Docker. So on a default installation you will see something like this.

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
36d51e62e518       bridge             bridge             local
18e8e68644ea       host               host               local
1f87f168df62       none               null               local
```

The `bridge` network is mapped to the `docker0` network bridge on the Docker Engine's host. The `ip address` command can be used to check the details of `docker0`.

```
$ ip address show label docker0
7: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 02:45:01:c0:ba:c7 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:1ff:fec0:bac7/64 scope link
        valid_lft forever preferred_lft forever
```

8.2.1 2.1. Inspect Networks

Every container created by the Docker engine is added to the default `bridge` network. The command `docker network inspect` can be used to inspect the network like so...

```
$ docker network inspect bridge
[
  {
```

```

    "Name": "bridge",
    "Id": "36d51e62e518d5430c798ce6eb68dc2737e9ad0555dd45097b04ef7597bce644",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "172.17.0.0/16",
          "Gateway": "172.17.0.1"
        }
      ]
    },
    "Internal": false,
    "Containers": {},
    "Options": {
      "com.docker.network.bridge.default_bridge": "true",
      "com.docker.network.bridge.enable_icc": "true",
      "com.docker.network.bridge.enable_ip_masquerade": "true",
      "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
      "com.docker.network.bridge.name": "docker0",
      "com.docker.network.driver.mtu": "1500"
    },
    "Labels": {}
  }
]

```

Note the "Containers" section in the above picture. It lists the containers attached to the bridge network. This section will be empty in a fresh Docker installation or when all containers are stopped.

8.2.2 2.2. Create User Defined Networks

Apart from the default networks, users can create new networks with the `docker network create` command. Let us create a new network called `example-network` and inspect its contents now.

```

$ docker network create example-network && docker network inspect example-network
301ffd3fbc1e323d40d2bc687f2e5c6341c16011e91d57151577fa08c914a157
[
  {
    "Name": "example-network",
    "Id": "301ffd3fbc1e323d40d2bc687f2e5c6341c16011e91d57151577fa08c914a157",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.18.0.0/16",
          "Gateway": "172.18.0.1/16"
        }
      ]
    },
    "Internal": false,
    "Containers": {},
    "Options": {},
    "Labels": {}
  }
]

```

```
]

```

Let us now add a container to this network and inspect again. The command `docker run --network=` can be used to attach a container to a network of our choice, like below:

```
$ docker run -itd --name=infinite_loop --network=example-network enhariharan/infinite-loop
8da6157240e09d69c7490c1af4ce483b30ff6f7ba7804b45818c0975e7b8ba25

```

The container is added into the network `example-network`. Let us now investigate the network settings again.

```
$ docker network inspect example-network
[
  {
    "Name": "example-network",
    "Id": "301ffd3fbc1e323d40d2bc687f2e5c6341c16011e91d57151577fa08c914a157",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "172.18.0.0/16",
          "Gateway": "172.18.0.1/16"
        }
      ]
    },
    "Internal": false,
    "Containers": {
      "8da6157240e09d69c7490c1af4ce483b30ff6f7ba7804b45818c0975e7b8ba25": {
        "Name": "infinite_loop",
        "EndpointID": "↔
          ccb0318b24dc3b0f1676f5aa16e5ea839adb715baa6bcd4ad16cdadb4aabd973",
        "MacAddress": "02:42:ac:12:00:02",
        "IPv4Address": "172.18.0.2/16",
        "IPv6Address": ""
      }
    },
    "Options": {},
    "Labels": {}
  }
]

```

You can see that the "Containers" section now has the container `infinite_loop` created from the image.

8.2.3 2.3. Connect Containers Within a Network

All containers are added to the `bridge` network by default. Containers within the same Docker network can connect to each other. Let us see that through a simple example. We will create 2 simple containers from the image `enhariharan/infinite-loop` into the custom network `example-network` created earlier. Then we'll inspect the network to see that the containers were added as expected.

```
$ docker run -itd --name=infinite_loop_1 --network=example-network enhariharan/infinite- ↔
loop
3c23fb845274e24ac8bdaa3216affc3b6d2755b58e6a8a4424c4008ca3c8022c
$
$ docker run -itd --name=infinite_loop_2 --network=example-network enhariharan/infinite- ↔
loop
9bd946658d4456618f60413e0cb41f6b4502eb60cfbff325abd7f218bffdd45

```

```
$
$ docker network inspect example-network
...
  "Containers": {
    "3c23fb845274e24ac8bdaa3216affc3b6d2755b58e6a8a4424c4008ca3c8022c": {
      "Name": "infinite_loop_1",
      "EndpointID": "0 ↔
        c5fc22252e5d88ae746fb07d9dc9827ab6840a40db956eff2ed8bcfa042a099",
      "MacAddress": "02:42:ac:12:00:02",
      "IPv4Address": "172.18.0.2/16",
      "IPv6Address": ""
    },
    "9bd946658d4456618f60413e0cb41f6b4502eb60cfbfff325a1bd7f218bffdd45": {
      "Name": "infinite_loop_2",
      "EndpointID": "1 ↔
        eecb3c8414026bfeaf63cfb23fb5319ca068c2c3c220ad486f19de02b2225bd",
      "MacAddress": "02:42:ac:12:00:03",
      "IPv4Address": "172.18.0.3/16",
      "IPv6Address": ""
    }
  },
  },
  ...
```

Containers `infinite_loop_1` and `infinite_loop_2` were created in detached mode. They have been added into `example-network` with IP address `172.18.0.2` and `172.18.0.3` respectively, as confirmed above. If the `--network=` is not provided then containers are added to the bridge network by default. Now, let us open a session into the container `infinite_loop_1` and try to ping `infinite_loop_2`.

```
$ docker exec -it infinite_loop_1 sh
/ # ping -c 5 172.18.0.3
PING 172.18.0.3 (172.18.0.3): 56 data bytes
64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.165 ms
64 bytes from 172.18.0.3: seq=1 ttl=64 time=0.124 ms
64 bytes from 172.18.0.3: seq=2 ttl=64 time=0.115 ms
64 bytes from 172.18.0.3: seq=3 ttl=64 time=0.115 ms
64 bytes from 172.18.0.3: seq=4 ttl=64 time=0.116 ms

--- 172.18.0.3 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.106/0.117/0.137 ms
```

So as you can see, containers added into the same network can automatically find each other through their IP addresses. Next, let us see how to setup DNS in Docker so that we need use the IP addresses to communicate.

8.3 Setup Container DNS in Bridge Network

Docker engine uses these 3 files within every container to configure it's DNS.

- `/etc/hostname` - This file maps the container IP address to a name.
- `/etc/hosts` - This file maps other container's IP addresses to names.
- `/etc/resolv.conf` - This file contains the IP addresses of other DNS servers to refer to if the container cannot resolve a name to IP address.

These files can be setup and manipulated using the `docker run` command. The hostname for a container is set into `/etc/hostname` using the `--hostname=` option.

```
$ docker run -itd --name=infinite_loop_1 --hostname=container1 enhariharan/infinite-loop
986d6b95a283493edd6d331d3c9f0c31595b9ec2b4ffae737ad99f5b1b090c46
$
$ docker exec -it infinite_loop_1 cat /etc/hostname
container1
```

The entry of `container1` can be put into the `/etc/hosts` file of `container2` using the `--link=` option of `docker run`. Let us try one more example now, we will link `infinite_loop_1` to `infinite_loop_2` and ping `container1` from within `container2` using the hostname of `infinite_loop_1`.

```
$ docker run -itd --name=infinite_loop_1 --hostname=container1 enhariharan/infinite-loop
c97fe6577f1e8b2ca90b04157a299c45becadea5a5ccd61564c8d279d98c3719
$
$ docker run -itd --name=infinite_loop_2 --hostname=container2 --link=infinite_loop_1 ↔
enhariharan/infinite-loop
2a26adfae9fb06038a823e4fea9a5118875abdb8e023502eda96c91d4c79d6c
$
$ docker exec -it infinite_loop_2 cat /etc/hosts
127.0.0.1      localhost
::1          localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.3    infinite_loop_1 container1
172.17.0.4    container2
$
$ docker exec -it infinite_loop_2 ping -c 5 container1
PING container1 (172.17.0.3): 56 data bytes
64 bytes from 172.17.0.3: seq=0 ttl=64 time=0.175 ms
64 bytes from 172.17.0.3: seq=1 ttl=64 time=0.118 ms
64 bytes from 172.17.0.3: seq=2 ttl=64 time=0.115 ms
64 bytes from 172.17.0.3: seq=3 ttl=64 time=0.123 ms
64 bytes from 172.17.0.3: seq=4 ttl=64 time=0.118 ms

--- container1 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.115/0.129/0.175 ms
```

But the fun with discovering containers in the bridge network sort of stops here. To connect to containers without using their IP addresses and without linking to them at `docker run` as above we need to do user-defined networks of Docker. Let us see the basics of that next.

8.4 Setup Container DNS in a User Defined Network

There are some important differences between connecting containers in user-defined networks and within bridge network. One such point is that the `--link` option of `docker run` will not work in user-defined networks. Let us see how we can get the same functionality as above using user-defined networks. We will add the containers `infinite_loop_1` and `infinite_loop_2` into a user-defined network called `example_network` and ping `infinite_loop_2` from within `infinite_loop_1`.

```
$ docker run -itd --name=infinite_loop_1 --hostname=container1 --network=example_network ↔
enhariharan/infinite-loop
d91757ae6e4dab01d9a37e73ec9a314c15ed195e5763fb6d9898ceb45dbc0964
$
$ docker run -itd --name=infinite_loop_2 --hostname=container2 --network=example_network ↔
enhariharan/infinite-loop
3ab4dad71bad7114025db7330686cb23114c76f705f9e3489a2763aa74970f2b
```



```
$
$ docker exec -it infinite_loop_2 ping -c 4 infinite_loop_1
PING infinite_loop_1 (172.20.0.2): 56 data bytes
64 bytes from 172.20.0.2: seq=0 ttl=64 time=0.111 ms
64 bytes from 172.20.0.2: seq=1 ttl=64 time=0.122 ms
64 bytes from 172.20.0.2: seq=2 ttl=64 time=0.119 ms
64 bytes from 172.20.0.2: seq=3 ttl=64 time=0.131 ms

--- infinite_loop_1 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.111/0.120/0.131 ms
$
$ docker exec -it infinite_loop_2 ping -c 4 container1
ping: bad address 'container1'
```

From the above snippet, it is seen that containers are visible in the same network by their container names but not their host names since ping by the hostname `container1` does not work.

The recommended way is to add a network-alias to each container apart from the hostname. This network-alias can then be used to connect by other containers. Let us try the above steps again but by setting a network-alias this time.

```
$ docker run -itd --name=infinite_loop_1 --hostname=container1 --network=example_network -- ←
network-alias=container1_alias enhariharan/infinite-loop
c80675335e3a235b84e317d5a160324e7add9421d197524a26ef2c461a679f1c
$
$ docker run -itd --name=infinite_loop_2 --hostname=container2 --network=example_network -- ←
network-alias=container2_alias enhariharan/infinite-loop
532b2d3d2694ed41c3abe44ae32671619b312b9a5c33604f8eaacd620b1d2874
$
$ docker exec -it infinite_loop_2 ping -c 4 container1_alias
PING container1_alias (172.20.0.2): 56 data bytes
64 bytes from 172.20.0.2: seq=0 ttl=64 time=0.127 ms
64 bytes from 172.20.0.2: seq=1 ttl=64 time=0.115 ms
64 bytes from 172.20.0.2: seq=2 ttl=64 time=0.119 ms
64 bytes from 172.20.0.2: seq=3 ttl=64 time=0.113 ms

--- container1_alias ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.113/0.118/0.127 ms
```

So this is how Docker Engine's embedded DNS can be configured so that containers can communicate with other containers within the same host and within the same network bridge. To talk among other networks or to talk among other hosts an [overlay network](#) must be configured. Also, a cluster of Docker hosts can be created using tools like [Docker Swarm](#). These topics are outside the scope of this post and will be taken up in a future post.

8.5 Summary

In this post, we were introduced to the basics of Docker networking. We learned the types of networks supported and how to create a user-defined `bridge` type network. We also understand how to add containers into specific networks. Then we understood how containers can communicate with one another by configuring the embedded DNS server provided by Docker Engine.

Chapter 9

Docker Start Container Example

A container, in Docker terminology, is an instance of an image. A typical simile used for this, is the classes and objects in Object Oriented Programming.

This example will show how to start containers, that is, making run images instances.

For this example, Linux Mint 18 and Docker version 1.12.1 have been used.

9.1 Installation

Note: Docker requires a 64-bit system with a kernel version equal or higher to 3.10.

We can install Docker simply via `apt-get`, without the need of adding any repository, just installing the `docker.io` package:

```
sudo apt-get update
sudo apt-get install docker.io
```

For more details, you can follow the [Install Docker on Ubuntu Tutorial](#).

9.2 Pulling a sample image

If you have used Docker before, you (should) already know that a container is an instance of an image, so, first, we need one.

The easiest way to obtain Docker images is from Docker Hub, pulling them. We could pull, for example, an Nginx image:

```
docker pull nginx
```

You can check that you have the image on your system executing:

```
docker images
```

Which should show something similar to:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
nginx	latest	abf312888d13	2 weeks ago	181.5 MB

9.3 Creating a container from an image and running it: run

Now that we have a Docker image, we can easily create a container from it.

As said before, a container is just an instance of an image. We can have several different containers from a single image, running at the same time.

The `run` docker command creates the container, and then runs it. So, running a container from an image is as easy as executing:

```
docker run <image-name|image-id>[:tag]
```

That means that we can create our container executing:

```
docker run nginx
```

And, also:

```
docker run abf312888d13
```

Additionally, we can also specify the tag:

```
docker run nginx:latest
```

For the previous examples, Docker would create the container from the image that already exists in our disk. But, we can also directly run containers for which we don't have the image in the disk; Docker is clever enough to know that it has to pull it, if it cannot be found in the disk. So, we could execute:

```
docker run nginx:1.10
```

Which is an image that doesn't exist, and Docker will do a `pull` followed by the `run`, after saying that it was unable to find the image:

```
Unable to find image 'nginx:1.10' locally
```

9.3.1 Running containers in detached mode

If you ran the container, you would see that the container is being ran in the console, and, that if you terminate the process, the container is stopped.

Usually, we want to run the containers in background or detached. For that, we have to use the `-d` option:

```
docker run -d nginx
```

With this option, Docker will show the container ID, and then detach the container.

9.3.2 Giving a name to the container

The random names generated by Docker do not say anything about the container, so, we should always give a meaningful name to the container. This is achieved with the `pull` option, e.g.:

```
docker run -d --name=nginx1 nginx
```

9.3.3 Mapping container ports to host ports

An essential option for most of the containers is the port publishing. If we are running a web server, as in this example, is obviously essential.

For this, we have to use the `-p` (`--publish`) option, which follows this format:

```
docker run -p <host-port>:<container-port> <image>
```

So, for example, if we would want to map the port 80 of the container to the port 80 of the host, we would execute:

```
docker run -d --name=nginx2 -p 80:80 nginx
```

Now, if we follow `localhost` (or `127.0.0.1`) in a browser, we should see the Nginx welcome page. That is, we would be accessing the web root of the web server running in the container.

Of course, we can use any (free) port in the host:

```
docker run -d --name=nginx3 -p 8080:80 nginx
```

9.3.4 Run a container and get the command line

For some occasions, we just want to get the command line of a container. For this, we have to use two options: `-i` (`--interactive`) and `-t` (`--tty`) and also specify the shell:

```
docker run -i -t <image> </path/to/shell>
```

For example:

```
docker run --name=nginx4 -i -t nginx /bin/bash
```

And we will run the container interactively.

For exiting the container, we can execute `exit` inside it.

We can also combine both options in just one with `-it`:

```
docker run --name=nginx5 -it nginx /bin/bash
```

9.3.5 Specifying a username

Exists the possibility of specifying the username the container has to be ran with, with the `-u` (`--user`) option:

```
docker run -u <username> <image>
```

Applied to the previous example, for running the container not with `root` user, but with `nginx`:

```
docker run --name=nginx6 -u nginx -it nginx /bin/bash
```

9.4 Starting an existing container: restart

Let's suppose that we stop one of the containers:

```
docker stop nginx1
```

Now, the `nginx1` container is stopped. If we would try to create a container with the same name:

```
docker run --name=nginx1 nginx
```

Docker would throw an error, and the container wouldn't be started:

```
docker: Error response from daemon: Conflict. The name "/nginx1" is already in use by ↵  
container 4e3a240cd6c20d6968afdb91be14860be72ed6ca652e6d52ac8546ee5d6ce16d. You have to ↵  
remove (or rename) that container to be able to reuse that name.
```

The message is quite explanatory: we cannot create a container with the name of another container (that's why we have used different names for the containers in this example), even if stopped. For that, we would have to delete or rename it.

If what we want to do is to run that container, that is, that **specific instance of the nginx image that we created before**, we don't have to use the start command, but restart:

```
docker restart <container-name|container-id>
```

So, for restarting our stopped container, we just have to execute:

```
docker restart nginx1
```

And it will be running again, keeping the port mapping, if specified.

9.5 Summary

This example has shown how to run Docker containers, that is, an instance of a Docker image. We have seen several options for these. Apart from that, we have also seen how to restart existing containers, which is slightly different from creating them.

Chapter 10

Docker List Containers Example

This example will show you how to list containers with Docker, one of the most popular software of the moment. Apart from just listing the containers, we will see the useful options that this command is provided with.

For this example, Linux Mint 18 and Docker version 1.12.1 have been used.

10.1 Installation

Note: Docker requires a 64-bit system with a kernel version equal or higher to 3.10.

We can install Docker simply via `apt-get`, without the need of adding any repository, just installing the `docker.io` package:

```
udo apt-get update
sudo apt-get install docker.io
```

For more details, you can follow the [Install Docker on Ubuntu Tutorial](#).

10.2 Setting up some containers

10.2.1 Pulling a sample image

As you should know, before creating containers, we need Docker images.

The easiest way to obtain a Docker image is to pull one from the Docker Hub. You don't have to have an account for pulling images.

Let's pull a Busybox image:

```
<code class="plain">docker pull busybox</code>
```

10.2.2 Creating sample containers

Let's create several containers:

```
docker run busybox
docker run busybox echo "Hello world"
docker run --name=mybusybox busybox
docker run -it busybox
```

Note: after the last container execution, open a new terminal, since the have set the interactive mode for it.

10.3 Listing containers

The Docker command for listing containers is `ps` (yes, not a very good name). So, let's try it:

```
docker ps
```

For which will return an output like the following:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
e723bc39fdd9	busybox	"sh"	3 minutes ago	Up 3
minutes		jolly_bhabha		

The first thing that probably will come to our minds is that we are just seeing a single container, when we actually instantiated several of them. **This is because `ps` command just shows the active, running containers.** And, actually, we only have a single container running: the one we instantiated interactively.

If we want to show all the containers, we have to pass the `-a` (`--all`) option to `ps`:

```
docker ps -a
```

In this case, the output would be:

CONTAINER ID	IMAGE	PORTS	COMMAND	NAMES	CREATED	STATUS
e723bc39fdd9	busybox		"sh"	jolly_bhabha	3 minutes ago	Up 3
minutes						
642aec18d638	busybox		"sh"	mybusybox	3 minutes ago	Exited
(0) 3 minutes ago						
fe574b4fcca5	busybox		"sh"	grave_ramanujan	3 minutes ago	Exited
(0) 3 minutes ago						
0583be5c5598	busybox		"echo 'Hello world'"	pedantic_hoover	3 minutes ago	Exited
(0) 3 minutes ago						
e4ceead785b3	busybox		"sh"	cranky_murdock	3 minutes ago	Exited
(0) 3 minutes ago						

10.3.1 Formatting the output

You may find the `ps` output not very pretty, since the lines can be wrapped if the width of the terminal window is not big enough. Fortunately, this command allows custom formatting. This is achieved with the `--format` option. The format is specified in "Go Template", which is pretty easy:

```
docker ps --format '{{.<column-name1>}}[ {{.<column-nameN>}} ]'
```

Specifying as many columns as we want. Of course, the formatting is compatible with `-a` option.

For example, if we would just want to see the names, we could execute:

```
docker ps -a --format '{{ .Names }} {{ .Status }}'
```

Which would show:

```
jolly_bhabha Exited (0) 5 minutes ago
mybusybox Exited (0) 5 minutes ago
grave_ramanujan Exited (0) 5 minutes ago
pedantic_hoover Exited (0) 5 minutes ago
cranky_murdock Exited (0) 5 minutes ago
```

The name of the fields are case sensitive, and for some cases the field name we have to use with `--format` is not very intuitive. So this is the list of the field name we have to use for each column:

- CONTAINER ID: ID
- IMAGE: Image
- COMMAND: Command
- CREATED: RunningFor
- STATUS: Status
- PORTS: Ports
- NAMES: Names

Note that the fields which were named CONTAINER ID and CREATED, use completely different names for the formatting. If we want to show the header for each column, we can add the table identifier before we specify the format, for example:

```
docker ps -a --format 'table {{ .ID }} {{ .Names }}'
```

Which would make the output look like:

```
CONTAINER ID      NAMES
e723bc39fdd9     jolly_bhabha
642aec18d638     mybusybox
fe574b4fcca5     grave_ramanujan
0583be5c5598     pedantic_hoover
e4ceead785b3     cranky_murdock
```

And, to align each column, we can use the tabulation t character between each one:

```
docker ps -a --format 'table {{ .ID }}\t{{ .Names }}'
```

Having a nicer formatting:

```
CONTAINER ID      NAMES
e723bc39fdd9     jolly_bhabha
642aec18d638     mybusybox
fe574b4fcca5     grave_ramanujan
0583be5c5598     pedantic_hoover
e4ceead785b3     cranky_murdock
```

10.3.2 Saving format templates in Docker configuration file

We have seen how useful can be the custom formatting for container outputting. But, on the other hand, it requires to write quite a lot, which perhaps make thing that it's not worth.

Fortunately, Docker allows to define a default formatting configuration for the ps command, so, every time we execute it, it will show the output as defined, without the need of specifying the --format option.

For this, we just have to open the config.json file with our favorite editor, located in the .docker/ folder, in the home directory:

```
vim ~/.docker/config.json
```

And add a --table value for the psFormat entry, e.g.:

```
{
  "psFormat": "table {{ .ID }}\t{{ .Names }}\t{{ .Status }}"
}
```

Now, just executing the following Docker command:


```
docker ps -a
```

The output would be:

CONTAINER ID	NAMES	STATUS
e723bc39fdd9	jolly_bhabha	Exited (0) 10 minutes ago
642aec18d638	mybusybox	Exited (0) 10 minutes ago
fe574b4fcca5	grave_ramanujan	Exited (0) 10 minutes ago
0583be5c5598	pedantic_hoover	Exited (0) 10 minutes ago
e4ceead785b3	cranky_murdock	Exited (0) 10 minutes ago

10.3.3 Using filters

The remaining interesting feature for container listing is the filtering of the results. This is useful if we are dealing with several containers, and we just want information about some of them.

For filtering the output, the option `-f` (`--format`) is used. The format is the following:

```
docker ps -f "key1=value1" -f "key2=value2" -f "keyN=valueN"
```

As you can see, we can specify as many filters as we want, but, for each of them, we have to specify the filter option.

For example, we could filter by the container name:

```
docker ps -a -f "name=mybusybox"
```

Showing the following output:

CONTAINER ID	NAMES	STATUS
642aec18d638	mybusybox	Exited (0) 15 minutes ago

(Keeping the configuration as we saw in previous section).

The filter also supports regular expressions, for example:

```
docker ps -a -f "name=/*_"
```

Looking for container names with an underscore, outputting:

CONTAINER ID	NAMES	STATUS
e723bc39fdd9	jolly_bhabha	Exited (0) 15 minutes ago
fe574b4fcca5	grave_ramanujan	Exited (0) 15 minutes ago
0583be5c5598	pedantic_hoover	Exited (0) 15 minutes ago
e4ceead785b3	cranky_murdock	Exited (0) 15 minutes ago

10.4 Summary

In this tutorial, we have seen how to list containers created with Docker. But we have gone further than that: we have seen the options that the command for listing containers provide, to make the output prettier and more suitable for our needs, seeing also how to save the configuration to use it as default.