## What is Docker?

Docker allows an application and all its dependencies to be packaged in a Container that will always run the same regardless of the environment it's deployed in. The term "dependencies" here refers to the components an application needs to run, like Java, OS packages or libraries

# DOCKER Install Docker

Docker is available in two editions: **Community Edition (CE)** and **Enterprise Edition (EE)**. **Docker Community Edition (CE)** is ideal for developers and small teams looking to get started with Docker and experimenting with container-based apps. **Docker Enterprise Edition (EE)** is designed for enterprise development and IT teams who build, ship, and run business critical applications in production at scale.

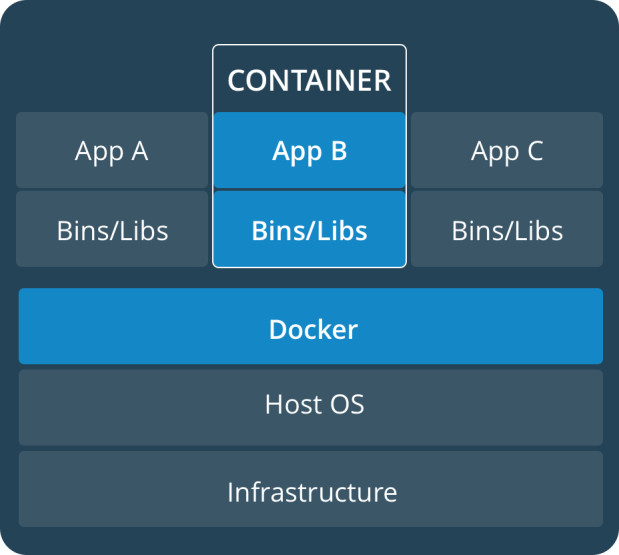
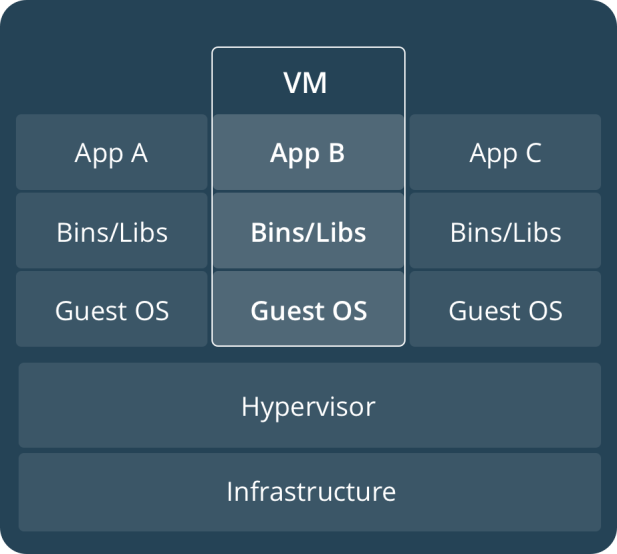
### Supported Cloud Env.

# Amazon Web Services

# <https://docs.docker.com/docker-cloud/cloud-swarm/link-aws-swarm/#create-a-dockercloud-swarm-role-role-with-an-embedded-policy>

# Microsoft Azure Cloud

# <https://docs.docker.com/docker-for-azure/>

### CONTAINERS

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), and start almost instantly.

### VIRTUAL MACHINES

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, one or more apps, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

### A brief explanation of containers

An **image** is a lightweight, stand-alone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and config files.

A **container** is a runtime instance of an image—what the image becomes in memory when actually executed. It runs completely isolated from the host environment by default, only accessing host files and ports if configured to do so.

Containers run apps natively on the host machine’s kernel. They have better performance characteristics than virtual machines that only get virtual access to host resources through a hypervisor. Containers can get native access, each one running in a discrete process, taking no more memory than any other executable.

### Define a container with a Dockerfile

**Dockerfile** will define what goes on in the environment inside your container. Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system, so you have to map ports to the outside world, and be specific about what files you want to “copy in” to that environment. However, after doing that, you can expect that the build of your app defined in this Dockerfile will behave exactly the same wherever it runs.

### Dockerfile

Create an empty directory. Change directories (cd) into the new directory, create a file called Dockerfile, copy-and-paste the following content into that file, and save it. Take note of the comments that explain each statement in your new Dockerfile.

# Use an official Python runtime as a parent image

FROM python:2.7-slim

# Set the working directory to /app

WORKDIR /app

# Copy the current directory contents into the container at /app

ADD . /app

# Install any needed packages specified in requirements.txt

RUN pip install -r requirements.txt

# Make port 80 available to the world outside this container

EXPOSE 80

# Define environment variable

ENV NAME World

# Run app.py when the container launches

CMD ["python", "app.py"]

### requirements.txt

Flask

Redis

## Share your image

# To demonstrate the portability of what we just created, let’s upload our built image and run it somewhere else.

### Tag the image

# The notation for associating a local image with a repository on a registry is username/repository:tag

docker tag image username/repository:tag

For example:

docker tag friendlyhello john/get-started:part2

### Publish the image

Upload your tagged image to the repository:

docker push username/repository:tag

### Pull and run the image from the remote repository

From now on, you can use docker run and run your app on any machine with this command:

docker run -p 4000:80 username/repository:tag

## Services

# In a distributed application, different pieces of the app are called “services.” For example, if you imagine a video sharing site, it probably includes a service for storing application data in a database, a service for video transcoding in the background after a user uploads something, a service for the front-end, and so on.

# Services are really just “containers in production.” A service only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

## docker-compose.yml file

A docker-compose.yml file is a YAML file that defines how Docker containers should behave in production.

### docker-compose.yml

Save this file as docker-compose.yml wherever you want. Be sure you have [pushed the image](https://docs.docker.com/get-started/part2/#share-your-image) you created in [Part 2](https://docs.docker.com/get-started/part2/) to a registry, and update this .yml by replacing username/repo:tag with your image details.

version: "3"

services:

web:

# replace username/repo:tag with your name and image details

image: username/repo:tag

deploy:

replicas: 5

resources:

limits:

cpus: "0.1"

memory: 50M

restart\_policy:

condition: on-failure

ports:

- "80:80"

networks:

- webnet

networks:

webnet:

This docker-compose.yml file tells Docker to do the following:

* Pull [the image we uploaded in step 2](https://docs.docker.com/get-started/part2/) from the registry.
* Run 5 instances of that image as a service called web, limiting each one to use, at most, 10% of the CPU (across all cores), and 50MB of RAM.
* Immediately restart containers if one fails.
* Map port 80 on the host to web’s port 80.
* Instruct web’s containers to share port 80 via a load-balanced network called webnet. (Internally, the containers themselves will publish to web’s port 80 at an ephemeral port.)
* Define the webnet network with the default settings (which is a load-balanced overlay network).

## Run your new load-balanced app

Before we can use the docker stack deploy command we’ll first run:

docker swarm init

**Note**: We’ll get into the meaning of that command in [part 4](https://docs.docker.com/get-started/part4/). If you don’t run docker swarm init you’ll get an error that “this node is not a swarm manager.”

Now let’s run it. You have to give your app a name. Here, it is set to getstartedlab:

docker stack deploy -c docker-compose.yml getstartedlab

Our single service stack is running 5 container instances of our deployed image on one host. Let’s investigate.

Get the service ID for the one service in our application:

docker service ls

You’ll see output for the web service, prepended with your app name. If you named it the same as shown in this example, the name will be getstartedlab\_web. The service ID is listed as well, along with the number of replicas, image name, and exposed ports.

## Scale the app

You can scale the app by changing the replicas value in docker-compose.yml, saving the change, and re-running the docker stack deploy command:

docker stack deploy -c docker-compose.yml getstartedlab

Docker will do an in-place update, no need to tear the stack down first or kill any containers.

Now, re-run docker container ls -q to see the deployed instances reconfigured. If you scaled up the replicas, more tasks, and hence, more containers, are started.

### Take down the app and the swarm

* Take the app down with docker stack rm:
* docker stack rm getstartedlab
* Take down the swarm.

docker swarm leave --force

## Understanding Swarm clusters

A swarm is a group of machines that are running Docker and joined into a cluster. After that has happened, you continue to run the Docker commands you’re used to, but now they are executed on a cluster by a **swarm manager**. The machines in a swarm can be physical or virtual. After joining a swarm, they are referred to as **nodes**.

The basic concept is simple enough**: run docker swarm init** to enable swarm mode and make your current machine a swarm manager, then **rundocker swarm join** on other machines to have them join the swarm as workers.

## Spring Boot & Docker

<https://spring.io/guides/gs/spring-boot-docker/>

Let’ say we have rest service exposed.

## Containerize It

Docker has a simple [Dockerfile](https://docs.docker.com/reference/builder/) file format that it uses to specify the "layers" of an image. So let’s go ahead and create a Dockerfile in our Spring Boot project:

Dockerfile

FROM openjdk:8-jdk-alpine

VOLUME /tmp

ADD target/gs-spring-boot-docker-0.1.0.jar app.jar

ENV JAVA\_OPTS=""

ENTRYPOINT exec java $JAVA\_OPTS -Djava.security.egd=file:/dev/./urandom -jar /app.jar

This Dockerfile is very simple, but that’s all you need to run a Spring Boot app with no frills: just Java and a JAR file. The project JAR file is ADDed to the container as "app.jar" and then executed in the ENTRYPOINT.

|  |  |
| --- | --- |
|  | We added a VOLUME pointing to "/tmp" because that is where a Spring Boot application creates working directories for Tomcat by default. The effect is to create a temporary file on your host under "/var/lib/docker" and link it to the container under "/tmp". This step is optional for the simple app that we wrote here, but can be necessary for other Spring Boot applications if they need to actually write in the filesystem. |
|  | To reduce [Tomcat startup time](https://wiki.apache.org/tomcat/HowTo/FasterStartUp#Entropy_Source) we added a system property pointing to "/dev/urandom" as a source of entropy. |

|  |  |
| --- | --- |
|  | if you are using boot2docker you need to run it **first** before you do anything with the Docker command line or with the build tools (it runs a daemon process that handles the work for you in a virtual machine). |

## Maven Build Plugin

<build>

<plugins>

<plugin>

<groupId>com.spotify</groupId>

<artifactId>**dockerfile-maven-plugin**</artifactId>

<version>1.3.4</version>

<configuration>

<repository>${docker.image.prefix}/${project.artifactId}</repository>

</configuration>

</plugin>

</plugins>

</build>

The configuration specifies 2 things:

* The repository with the image name, which will end up here as springio/gs-spring-boot-docker
* Optionally, the image tag, which ends up as latest if not specified. It can be set to the artifact id if desired.

**You can build a tagged docker image using the command line like this:**

* $ ./mvnw install dockerfile:build
* And you can push the image to dockerhub with ./mvnw dockerfile:push.

<https://medium.com/@hudsonmendes/docker-spring-boot-microservice-with-gradle-9785087e7992>

<http://www.baeldung.com/dockerizing-spring-boot-application>

## Docker Concepts

### Docker Image

A Docker Image acts as a blueprint or template for a Docker Container. For example docker run ubuntu will create a **new** Docker Container based on the **ubuntu** Docker Image. The container does not modify the image in any way unless changes are explicitly committed. New images can easily be created by inheriting from existing images, online repositories like [Docker Hub](https://hub.docker.com/)makes it easy to share and find images created by the community.

### Dockerfile

The preferred way to create a Docker Image is with a script known as a Dockerfile. Alternatively the required changes can be made through running a shell on a container and then committing it to an image. Dockerfiles have the advantage of providing the ability to automate image creation and the syntax is simple, clear and self explanatory. The below example will create an image based on **ubuntu** and will run the echo "Hello docker!" command when a container is created from this image.

#Sample Dockerfile

FROM ubuntu

CMD "echo" "Hello docker!"

### Docker Container

A Docker Container [is a running instance](http://stackoverflow.com/questions/23735149/docker-image-vs-container) of a Docker Image and there can be many running instances of the same image. A Docker Volume can be used to persist changes to the file system in a Docker Container. Every time a docker run <image> command is run a **new** container is created from the given image. Docker Containers that expose ports can be mapped to a port on the host where the Docker Daemon is running with docker run -p 8080:80 <name|id>.

### Docker Volumes

Changes an application makes to the filesystem (like writing log or database files) will not persist beyond the lifecycle of a Docker Container unless these files are written to a mounted Volume. A Volume remains available after a Docker Container that uses it is destroyed.

To create a data Volume for Mongo and then start it run:

docker volume create **--**name data**-**mongo

docker run **-**v data**-**mongo:/data/db mongo

Then to backup the Mongo data into the current working directory:

docker run **--**rm **-**v data**-**mongo:/data/mongo **-**v $(pwd):/backup busybox tar cvf /backup/mongo**-**data.tar /data

docker volume inspect <name|id> will show the mount point of the Volume. On

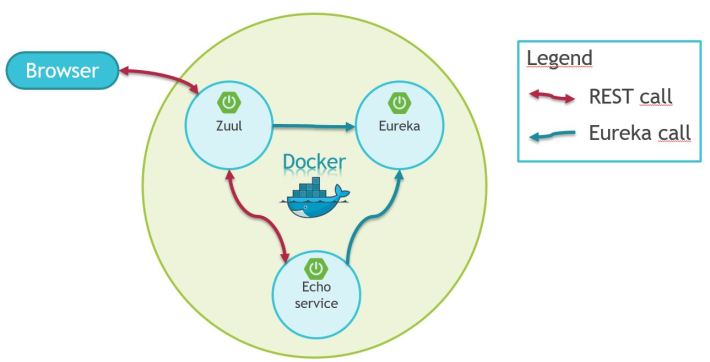
## Basic Docker Commands

* docker run <image> Creates a new Container each time it is run.
* docker start <name|id> Starts an existing Container.
* docker stop <name|id> Stops an existing Container.
* docker ps [-a include stopped containers] List all Containerscreated.
* docker rm <name|id> Remove a Container.
* docker rm $(docker ps -aq) Delete all Containers.
* docker images Lists built and dowloaded Images.
* docker rmi node remove Image named 'node'
* docker exec -t -i <name|id> /bin/bash Get a shell in a running Container.
* docker volume ls Lists the created data Volumes.
* docker build -t <name> . Build an Image from a Dockerfile in the current directory.

Docker is a powerful tool that will reduce issues caused by differences in development and production environments, by defining Docker Images that can be shared between the environments.

**Docker Good Example**

<https://exampledriven.wordpress.com/2016/06/24/spring-boot-docker-example/>



We will use a Zuul API Gateway and an Eureka service registry. Details of how they work are explained in other blog post of the series (links below), but what matters for this exercise is that they are regular Spring Boot applications. A very similar ecosystem could be build from any java application.

## Technology Used

* Spring boot 1.3.5.RELEASE
* Docker
* Docker Compose
* Docker maven plugin
* Zuul
* Eureka

This article is part of a Spring Cloud / Netflix OSS series

## Use docker-compose the start up multiple containers and set up a container network

Docker compose is a command line tool to start up multiple predefined containers. Container definitions are stored in a file named docker-compose.yml . The above example starts up three containers. The depends\_on defines the startup order and how containers see the other. The file also defines the exposed ports of the services.

|  |  |
| --- | --- |
| version: '2'  services:    echo-service:      image: echo-service      depends\_on:       - eureka-server    eureka-server:      image: eureka-server      ports:       - "8761:8761"    zuul-server:      image: zuul-server      depends\_on:       - eureka-server      ports:       - "9090:9090" |  |

The above file defines that

* Eureka will expose it’s 8761 port to the external network
* Zuul server will expose it’s 9090 port to the external network
* Echo-service will be only internally accessible

To start up the containers build all images then type

|  |  |
| --- | --- |
| ##use the -d option to start up in deamon mode |  |

docker-compose up

**With MYSQL**

version : '2'

services:

springappserver:

build:

context: .

dockerfile: springapp.dockerfile

ports:

- "8080:8080"

networks:

- net-spring-db

volumes:

- .:/vol/development

depends\_on:

- mysqldbserver

mysqldbserver:

build:

context: .

dockerfile: mysqldb.dockerfile

ports:

- "3306:3306"

networks:

- net-spring-db

environment:

MYSQL\_DATABASE: testdb

MYSQL\_USER: myuser

MYSQL\_PASSWORD: mypassword

MYSQL\_ROOT\_PASSWORD: myrootpassword

container\_name: mysqldbserver

networks:

net-spring-db:

driver: bridge

Now let's check the mysqldb.dockerfile:

FROM mysql/mysql-server

MAINTAINER Dursun KOC <dursunkoc@gmail.com>

# Copy the database initialize script:

# Contents of /docker-entrypoint-initdb.d are run on mysqld startup

ADD mysql/ /docker-entrypoint-initdb.d/