

Tetiana Tykhomyrova

Real world Docker applications

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

17 April 2018

Author Title	Tetiana Tykhomyrova Real world Docker applications
Number of Pages Date	25 pages 17 April 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Head of Department
<p>Docker is one of the fastest developing modern technologies. The purpose of this thesis is to showcase Docker use across multidisciplinary applications with intention of helping beginners to learn and understand more about this technology.</p>	
Keywords	Docker, React, Container

Contents

1	Introduction	1
1.1	Background	1
1.1.1	Unix v7	1
1.1.2	FreeBSD Jails	1
1.1.3	Linux VServer	2
1.1.4	Oracle Solaris Containers	2
1.1.5	Open VZ	2
1.1.6	Process Containers	2
1.1.7	LXC	2
1.1.8	Warden	3
1.1.9	LMCTFY	3
1.1.10	Kubernetes	3
1.2	History of Docker	3
1.3	Objective of the thesis	4
1.4	Installation	4
1.5	Windows vs Linux	4
2	Docker technology	5
1.1	Images	5
2.5	DockerHub	6
3	Docker use	7
3.2	VR	7
3.2.1	Problem:	7
3.2.2	Solution:	7
4.2	Visa	8
3.2.3	Problem	8
3.2.4	Solution	8
3.3	Kadaster	8
3.3.1	Problem	8
3.3.2	Solution	9
3.4	ADP	9
3.4.1	Problem	9

3.4.2	Solution	9
3.5	Cornell university	9
3.5.1	Problem	9
3.5.2	Solution	10
3.6	BBC News [9]	10
3.6.1	Problem	10
3.6.2	Solution	10
4	First Docker container	10
5	Usual Docker commands	12
6	Docker compose	15
6.2	Structure of a yemo file	16
7	Dockerizing React application	17
8	Writing Dockerfile for a React App	18
9	Pushing images into Docker Cloud	21
10	Docker Networks	24
11	Creating a custom network	25
12	Using Docker in production	27
13	Conclusion	28
	References	29

1 Introduction

Docker is an implementation of container-based virtualization technologies. Main differences between virtualization technology and pre-virtualization technology are cost, speed of deployment and ease of migration from one machine to another.

With hypervisor-based virtual machines it's a lot more cost efficient and easy to scale since no applications are stored locally. All the technology can be installed in the cloud which makes scaling a lot faster. Even though with all the benefits of hypervisor-based virtual machines over pre-virtualization technology there are still some limitations.

This is a time when Docker comes into the picture. [1]

1.1 Background

The concept of containers is not new, to be more precise the concept was born back in 1979. I would like to discuss about some of them to give clear picture of the topic and why I choice it. [2]

1.1.1 Unix v7

With the development of Unix V7 chroot (change root) system was introduced. It is an operation which changes root directory for the current processes and their children. [3]

1.1.2 FreeBSD Jails

FreeBSD is an operating system which used to power servers, desktops and embedded systems.

In early 2000's a shared-environment hosting provider came up with a concept of separation between customer environment and its own. FreeBSD Jails lets administrators to

partition a FreeBSD computer system into smaller systems with their own IP addresses and configuration.

1.1.3 Linux VServer

Similar to FreeBSD Jails it is a concept which allows partition of file systems, network addresses and memory. It became available in 2001 and it is implemented by patching the Linux kernel. It is still possible to use but last stable patch was released in 2006.

1.1.4 Oracle Solaris Containers

In 2004 Oracle introduced Solaris Container which provides system recourse controls and boundary separation which is provided by zones. Zones are completely isolated virtual servers running in the same operating system instance.

1.1.5 Open VZ

Open Virtuozzo (Open VZ) was not released as part of the official Linux kernel. It is an operating system virtualization technology for Linux which takes care of isolating and recourse management using patched Linux kernel. Technology was created in 2005.

1.1.6 Process Containers

Launched by Google in 2006 was introduced to limit and isolate usage of memory, CPU, network for a combination of processes. It was eventually merged to Linux kernel.

1.1.7 LXC

LinuX Containers (LXC) was first implementation of Linux container manager. It was introduced in 2008 using cgroups (previously called Process Containers) and Linux namespaces.

1.1.8 Warden

In 2011 CloudFoundry started Warden first using LXC and later on replacing it with their own implementation. Warden is able to isolate environments in any operating system and provides API for container management.

1.1.9 LMCTFY

Let Me Contain That For You (LMCTFY) started in 2013 as an open-source implementation of Google's container stack. Deployment stopped in 2015, when core parts started being used in libcontainer which is now part of Open Container Foundation.

1.1.10 Kubernetes

It is an open-source system for automating deployment and scaling of containerized applications. It supports very complex classes of applications. Kubernetes and Docker operate on different levels of stack and even can be used together.

1.2 History of Docker

Docker was released as an open-source project by dotCloud in 2013. Like previously mentioned technologies it relies on namespaces and cgroups to be able to take care of resource isolation. It allows to write an application in any language on any Linux distribution and then move it from a laptop to a test environment or production if needed. Before Docker was introduced you could never be sure that you could move your application without a problem from one environment to another.

According to Docker one month after launching a tutorial about 10000 developers tried it out. In a year Amazon started supporting Docker and by June 2014 Docker was downloaded more than 2 million times.

In 2014 CoreOS introduced another container technology called Rocket which is now the main competitor for Docker. [4]

1.3 Objective of the thesis

Docker provides great user interface and it is really easy to use. Although the technology has quite a downflow of not being really stable in production.

Docker's success made me very interested in the technology. While writing this thesis I would like to explore reasons for its popularity and to check with my own experience how easy and fast it is to use for a beginner.

1.4 Installation

The steps required to install Docker vary depending on the operating system you use. On Docker documentation page there is a lot of information which guides user through the process so it becomes rather straightforward. In the beginning of my project I was using VM, but later on I switched to local machine using Linux. Installation took about 10 minutes. Here are the steps taken in order to install Docker.[5]

1.5 Windows vs Linux

Docker works similarly on both operating systems but there are some differences which I would like to talk about. First of all installation is different on both systems. As user guide explains for Windows you can just download a user friendly installation wizard when for Linux you have to write commands. When you use Linux you install only Docker engine and management tools, no need to create virtual machine because docker containers will handle the setup for you. As for Windows Docker creates a virtual based Linux machine, but user doesn't have to worry about it since all the process is happening in the background. All Docker commands are same for both operating systems.

I tried to install Docker on Windows 10 Home edition, but I wasn't able to run it at all. I found out that home edition doesn't support Hyper-V, which was needed in order to run Docker, so I couldn't run it. I found a way around by using VirtualBox version , but there are so many forums which discuss problems connected to Docker that you can solve pretty much any problem you face with the help of fellow developers.

2 Docker technology

1.1 Images

There are two most important concepts of Docker technology : images and containers. Images are read only templates and they are used to create containers. They are created with the docker build command. Images are designed to be composed of layers of other images allowing a minimum amount of data to be sent when transferring data over a network. Images are stored in a Docker registry.[6]

2.2 Containers

Containers are runtime objects. They are lightweight and portable components of environment in which we run applications. Containers are created from images. Inside a container it has all dependencies which are needed to run the application.

2.3 Registry

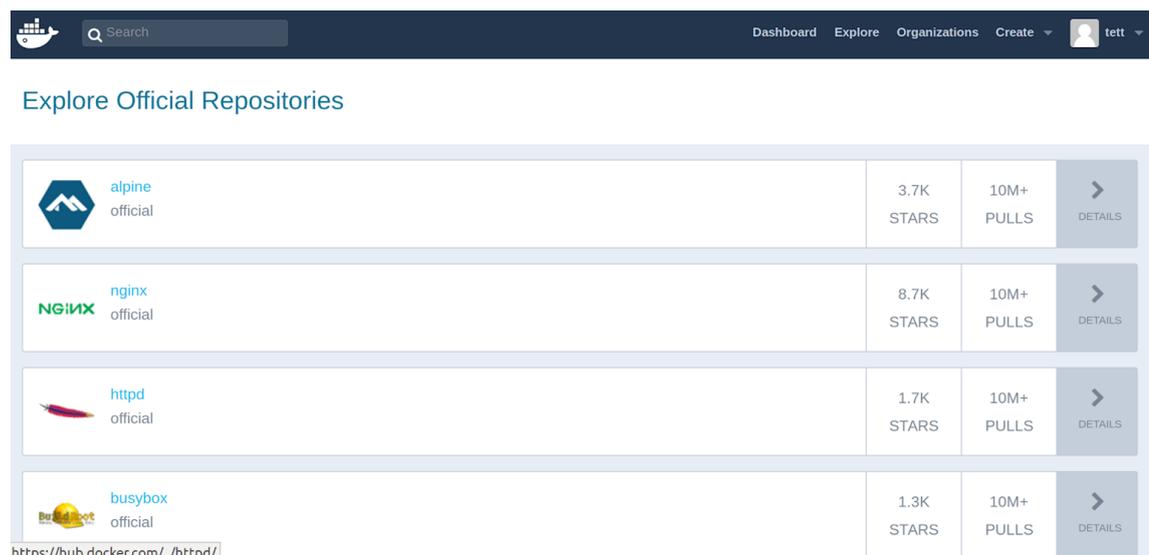
Registry is a place where we store images. User has a choice of hosting his own registry or using Docker's public registry called Docker Hub.

2.4 Repository

Inside a registry images are stored in repositories. Docker repository is a collection of different docker images with the same name but different tags. Usually each tag represents a different version of the image.

2.5 DockerHub

DockerHub is a registry service on the cloud which allows you to download Docker images or share yours with other users. [7]



The screenshot shows the Docker Hub interface. At the top, there is a navigation bar with a search bar and links for Dashboard, Explore, Organizations, Create, and a user profile (tett). Below the navigation bar, the page title is "Explore Official Repositories". The main content is a table listing four official repositories:

Repository Name	Stars	Pulls	Action
alpine official	3.7K STARS	10M+ PULLS	DETAILS
nginx official	8.7K STARS	10M+ PULLS	DETAILS
httpd official	1.7K STARS	10M+ PULLS	DETAILS
busybox official	1.3K STARS	10M+ PULLS	DETAILS

The URL at the bottom of the screenshot is https://hub.docker.com/_/httpd/.

Figure 1 Docker Hub interface

You can see how many stars are given to particular repository, how many pulls there are, and details of all repositories.

3 Docker use

I think it's always interesting to know which companies use Docker since it brings some kind of special vibe to the whole picture. To name a few companies:

VR Group - Finnish Railways, Paypal, General Electric, BBC News, Business Insider, Spotify, The New York Times, badoo. Next I would like to tell more about cases from different industries which shows how Docker can be used in different work environments.[8]

3.2 VR

Here I would like to focus on the case study of VR, since I am sure that almost everyone who ever been to Finland used their services.

3.2.1 Problem:

First of all VR had high operating cost since supporting many different platforms makes it complicated to make system efficient. Adding problems with quality delivery and simply how long it took for the applications to be ready made team look for alternatives.

3.2.2 Solution:

VR started working with Accenture to make a new common application platform based on Docker EE. Together they rewrote some applications and while that was happening they migrated already existing applications. Commuter services app went live in June 2017 and new reservation system went live in August 2017.

So what changes exactly happened? Operation cost savings went up to 50%, better visibility of the system which made quality go up, same platform for all the application in VR which makes it a lot easier and faster to develop and improve across the teams.

4.2 Visa

Second case study I would like to share is Visa since it is one of the most known brands in the world.

3.2.3 Problem

First of all company was making huge investments into virtualization but the outcome wasn't meeting the efforts. Maintenance of the system was taking way too much time. In order to improve its operations.

3.2.4 Solution

Visa started working with Docker by building two applications for payment processing. So what changes happened? Scalability increase. Visa has hundreds of thousands transaction going through its services every day, with two application and only 100 containers. In the peak time Visa can scale up to 800 containers in order to meet customer needs. Maintenance of the whole system became a lot faster and easier.

3.3 Kadaster

Kadaster is a registry governmental organization in Netherlands. They register property and land rights, aircrafts, ships.

3.3.1 Problem

Back in 2012 the biggest challenge they had was to be able to maintain different technology stack applications in the cloud.

3.3.2 Solution

Today there are about 250 Docker EE nodes which run Kadaster applications, removing the problem of not being able to use different technologies in the same space.

3.4 ADP

ADP is the largest company which provides human capital management solutions. It has more than half a million clients with more than 35 million users all over the world. Needless to say the company is quite big.

3.4.1 Problem

ADP faced problems with security, since they treat sensitive data like social security numbers. ADP is cloud based, which means they have to be able to scale their system fast in order to handle changes. Over the years company developed many applications which had thousands of lines of code.

3.4.2 Solution

Using Docker solved problem with security and workflow. In order to scale the application company relies on Universal Control Plane/Swarm. This way teams have the ability to do each part of the application in little Docker engine swarms rather than in one huge swarm per application.

3.5 Cornell university

Ivy league university which was founded in 1865.

3.5.1 Problem

Spending too much time and money on things that could be automated. They wanted to make sure they would be able to handle all the microservices in order.

3.5.2 Solution

Now university team is able to host their Docker images securely allowing only people who have access to use them. Docker ensures that even if an instance of application gets broken with the help of Docker Trusted Registry they can be always sure that applications are still available.

3.6 BBC News [9]

3.6.1 Problem

BBC had challenges with speed. Since BBC delivers news in about 30 languages with huge amount of information it was lagging behind.

3.6.2 Solution

Docker let BBC speed grew higher, eliminated waiting time and let jobs run simultaneously.

4 First Docker container

To make sure Docker is installed it is possible to run a command [10]

```
$ docker version
```

Docker can be run from terminal. Docker run command will create the container using the image user specifies, then spin up the container and run it.

Simple example of giving a command to run docker image can look like this:

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

When we use the image to run a container docker first looks through local box to find the image. If docker can't find image locally it will look for a download from remote registry. To check if there are any images we have locally we can type in

```
$docker images
```

The outcome of the command in my terminal looked like this:

```
root@ubuntu-512mb-fra1-01:~# docker images
REPOSITORY          TAG             IMAGE ID        CREATED
SIZE
hello-world         latest         1815c82652c0   2 months ago
1.84kB
root@ubuntu-512mb-fra1-01:~#
```

Figure 2 Docker images command

As seen on the picture, docker downloaded an image from repository “hello-world”, with a tag name “latest”. Images have unique ids.

The specifications for the command can be more precise in a form of

```
$docker run repository:tag command [arguments]
```

and the output for the command

```
$docker run busybox:1.24 echo "What's up" is:
```

```
root@ubuntu-512mb-fra1-01:~# docker run busybox:1.24 echo "What's up"
Unable to find image 'busybox:1.24' locally
1.24: Pulling from library/busybox
385e281300cc: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:8ea3273d79b47a8b6d018be398c17590a4b5ec604515f416c5b797db9dde3ad8
Status: Downloaded newer image for busybox:1.24
What's up
root@ubuntu-512mb-fra1-01:~#
```

Figure 3 Running busybox image

We can see that image wasn't found locally, and it was pulled from remote registry.

If we run the command again the execution will happen a lot faster since now the image is located locally.

The -i flag starts an interactive container. The -t flag creates a pseudo-TTY that attaches receiving or reading input such as stdin(standard in) and printing output such as stdout(standard output).

Implementing command:

```
$docker run -i -t busybox:1.24
```

Will put us right inside the container.

5 Usual Docker commands

- \$docker ps

Command is used to show all the running containers.

- \$docker ps -a

Shows all the containers which stopped as well.

In order to give a container specific name we can use a command

```
$docker run --name ello busybox:1.24
```

This will give a particular name to the container.

```

root@ubuntu-512mb-fra1-01:~# docker ps -a
CONTAINER ID   IMAGE          PORTS          COMMAND          NAMES          CREATED
STATUS        PORTS
3f972dfc6f60   busybox:1.24  "sh"          ello            43 seconds ago
  Exited (0) 43 seconds ago
9a7c42cd6d15   busybox:1.24  "sleep 1000"  infallible_wescoff  5 minutes ago
  Up 5 minutes
62ea281d7d15   busybox:1.24  "sh"          objective_borg    10 minutes ago
  Exited (0) 9 minutes ago
cfbfc231b0e4   busybox:1.24  "echo 'What's up'"  epic_banach      14 minutes ago
  Exited (0) 14 minutes ago
adc7678505b5   busybox:1.24  "echo 'What's up'"  lucid_minsky     18 minutes ago
  Exited (0) 18 minutes ago
9dffc05afd17   hello-world   "/hello"      determined_rosalind  36 minutes ago
  Exited (0) 35 minutes ago
e9b83d1c8bb3   hello-world   "/hello"      compassionate_joliot  36 minutes ago
  Exited (0) 36 minutes ago
271b551fae9a   hello-world   "/hello"      dreamy_agnesi     40 minutes ago
  Exited (0) 40 minutes ago

```

Figure 4 Container names

Docker containers can be run in a browser. This can be accessed by a command:

```
$docker run -it -p 8888:8080 tomcat:8.0
```

I had to use IP address given in Ubuntu console, since I was using cloud in order to access Docker and Linux distribution system.

```
$docker logs
```

This command will show any running containers.

When we create a container, we add a thin image layer on top of all underlying layers. It is called writable container layer. All changes made into the running containers will be written into the writable layer. When the container is deleted, the writable layer is also deleted, but the underlying image remains unchanged. This means that many containers can have different files but all of them have same underlying image.

There are two ways in which docker image can be built. One way is to commit changes made in a Docker container.

There are three main steps to be done in order to achieve the goal. We have to spin up a container from the base image, install Git package in the container and commit changes made in the container.

Docker commit is a command used to save the changes made to the Docker container file system to a new image.

Another is to write a Dockerfile. A Dockerfile is a document where user provides all the instructions to assemble the image. Each instruction creates new image layer to the image. Docker build command takes the path to the build context as an argument.

By default, docker would search for the Dockerfile in the build context path. Docker executes all the instructions written in the file. Then it creates a new container from the base image. Docker daemon runs each instruction in a different container. For example, for instruction, Docker daemon creates a container, runs the instruction, commits new layer to the image and removes the container.

Each run command will execute the command on the top writable layer of the container, then commit the container as a new image. As well as each run command will create a new image layer. It is important to remember to update packages alphanumerically. This way the process will be going faster. CMD instructions specify what command user wants to run when the container starts up, otherwise if CMD instructions are not specified in the Dockerfile, Docker will use default command defined in the base image.

If instructions do not change Docker will use same layer when building an image. ADD instruction let's not only to copy file but also allow to download file from internet and add it to the container.

All the dependencies are managed by Docker, so there is no need to install, for example, Python on the local machine in order to work with it.

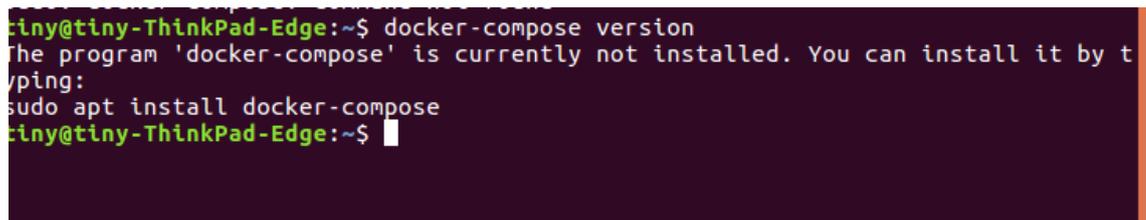
6 Docker compose

Container links allow containers locate each other and securely transfer information about one container to another container. When we set up a link we create a pipeline between the source container and recipient container. The recipient container can then access select data above the source container. In our case the rattus container is the source container and our container is the recipient container. The links are established by using container names.

The main use for docker container links is when we build an application with a micro-service architecture, we are able to run many independent components in different containers. Docker creates a secure tunnel between the containers that doesn't need to expose any ports externally on the container.

Docker compose is a very important component which is made in order to run multi-container Docker applications. With the help of Docker compose we can define all the containers in a single file called yemo file.

You can check if you have Docker compose installed simply by running a command:



```
tiny@tiny-ThinkPad-Edge:~$ docker-compose version
The program 'docker-compose' is currently not installed. You can install it by t
yping:
sudo apt install docker-compose
tiny@tiny-ThinkPad-Edge:~$
```

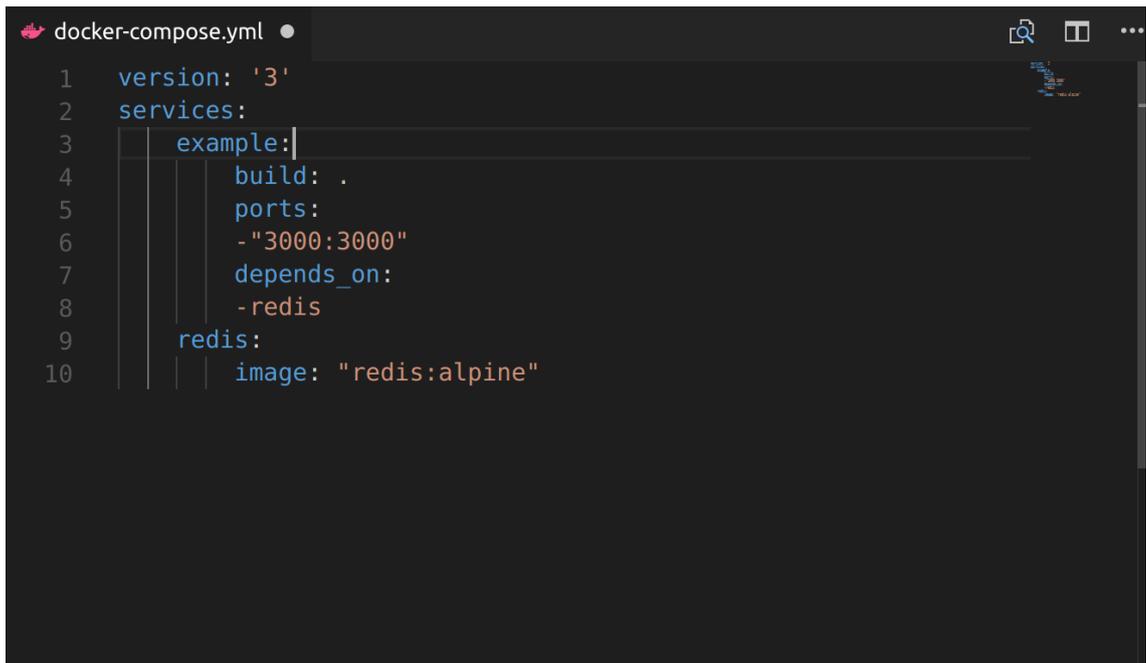
Figure 5 Checking if Docker-compose is installed

As we clearly see you will be told if installation is required. Next step is to create docker-compose.yml file.

6.2 Structure of a yemo file

There are three versions of docker compose file, version 1 which is legacy format which does not support volumes or networks, version 2 and version 3 which is the most up to date format and it is recommended to use it. Next we define services to make up our application for each service. We should provide instructions on how to build and run the container in the application. We have two services: example and redis.

The first instruction is the “build” instruction. The “build” instruction tells the path to the file which will be used to build docker image. Second instruction is “port” which defines what ports to show to external network. “Depends on” is the next part since in this example docker container is the client of redis and we need to start redis beforehand.



```
docker-compose.yml ●
1  version: '3'
2  services:
3    example:
4      build: .
5      ports:
6        - "3000:3000"
7      depends_on:
8        - redis
9    redis:
10   image: "redis:alpine"
```

Figure 6 Docker-compose file

7 Dockerizing React application

There are bunch of articles which can help building applications and deploy react applications, well basically any applications. The most important steps are explained below.[11]

Step 1

```
$ npm install -g create-react-app
```

Someone who has worked with React might see this steps as very familiar.

Step 2 (creating a folder)

```
$ mkdir myApp
```

Step 3 (get inside the folder)

```
$ cd myApp
```

Step 4 (create a react folder)

```
$ create-react-app frontend
```

Then we are creating a Dockerfile.

8 Writing Dockerfile for a React App

```
FROM node

ENV NPM_CONFIG_LOGLEVEL warn
ARG app_env
ENV APP_ENV $app_env

RUN mkdir -p /frontend
WORKDIR /frontend
COPY ./frontend ./

RUN npm install

CMD if [ ${APP_ENV} = production ]; \
    then \
    npm install -g http-server && \
    npm run build && \
    cd build && \
    hs -p 3000; \
    else \
    npm run start; \
    fi

EXPOSE 3000
```

Figure 7 Writing Dockerfile

So what does the code mean? [11]

FROM node means start from node base image

ENV NPM_CONFIG_LOGLEVEL warn means less messages during build

ARG app_env means that app environment can be set during build

ENV APP_ENV \$app_env means that an environment variable is set to *app_env* argument

RUN mkdir -p /frontend means that frontend folder is created

WORKDIR /frontend means that all commands will be run from this folder

COPY ./frontend ./ means that the code from the local folder is copied into container's working directory

RUN npm install installs dependencies

CMD if means *If the arg_env = production then* http-server will be installed, and then build. Otherwise used create-react-app hot reloading tool (basically webpack—watch)

EXPOSE 3000 means that app runs on port 3000 by default

Next commands in order to run the app are :

```
$ docker build ./
```

Type `$ docker images` to find out image id

After following command you should be able to run your container in localhost:3000

```
$ docker run -it -p 3000:3000 -v [put your path here]/frontend/src:/frontend/src [image id]
```

To build a production image run:

```
$ docker build ./ --build-arg app_env=production
```

To run the production image:

```
$ docker run -i -t -p 3000:3000 [image id]
```

And worry not if you make a mistake in your file, the image won't be built.

```
COPY ./my-app ./
RUN npm install
CMD if [ ${APP_ENV} = production ]; \
    then \
        npm install -g http-server && \
        npm run build && \
        cd build && \
        hs -p 3000; \
    else \
        npm run start; \
    fi
A
```

Figure 8 Letter A is a mistake in the file

Next is Docker build.

```

Sending build context to Docker daemon 293.5MB
Step 1/10 : FROM node
--> aa3e171e4e95
Step 2/10 : ENV NPM_CONFIG_LOGLEVEL warn
--> Using cache
--> cd87c0ed5e21
Step 3/10 : ARG app_env
--> Using cache
--> ae6f37ffe53b
Step 4/10 : ENV APP_ENV $app_env
--> Using cache
--> 4f8eb094fbfa
Step 5/10 : RUN mkdir -p /my-app
--> Running in 65a35aeff49a
Removing intermediate container 65a35aeff49a
--> 8184ca073a30
Step 6/10 : WORKDIR /my-app
Removing intermediate container fd6972d1a9ad
--> a1851e3a6ba5
Step 7/10 : COPY ./my-app ./
--> 54e440b26431
Step 8/10 : RUN npm install
--> Running in d3197fe3ed86

```

Figure 9 Docker build

Current Docker images situation.

```

tiny@tiny-ThinkPad-Edge:~/dilemmareact$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
<none>              <none>             6e41227bb689      3 minutes ago
937MB
<none>              <none>             5e32f21620e2      About an hour ago
804MB
<none>              <none>             8261e973e80a      About an hour ago
804MB

```

Figure 10 Docker images

Getting the image to run inside the container.

```

tiny@tiny-ThinkPad-Edge:~/dilemmareact$ sudo docker run -it -p 3000:3000 -v /home/
e/tiny/desktop/dilemmareact/my-app/src 6e41227bb689

```

Figure 11 Getting inside the container

This what your Terminal will show if everything goes fine.

```
Compiled successfully!  
  
You can now view my-app in the browser.  
  
Local:      http://localhost:3000/  
On Your Network: http://172.17.0.2:3000/  
  
Note that the development build is not optimized.  
To create a production build, use npm run build.
```

Figure 12 Successful Docker build

As a result React app is up.

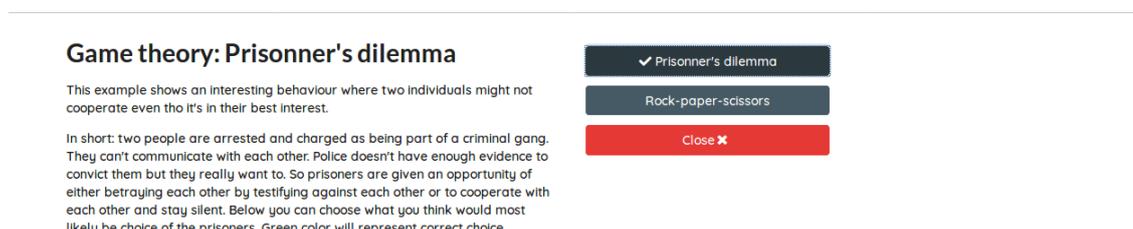


Figure 13 React app interface

9 Pushing images into Docker Cloud

Important thing to know from the beginning Docker Cloud does not provide cloud services. Docker cloud however has more added features than Docker Hub but it is built on top of Docker Hub. If you push an image to Docker Hub it will be automatically in Docker cloud. [12]

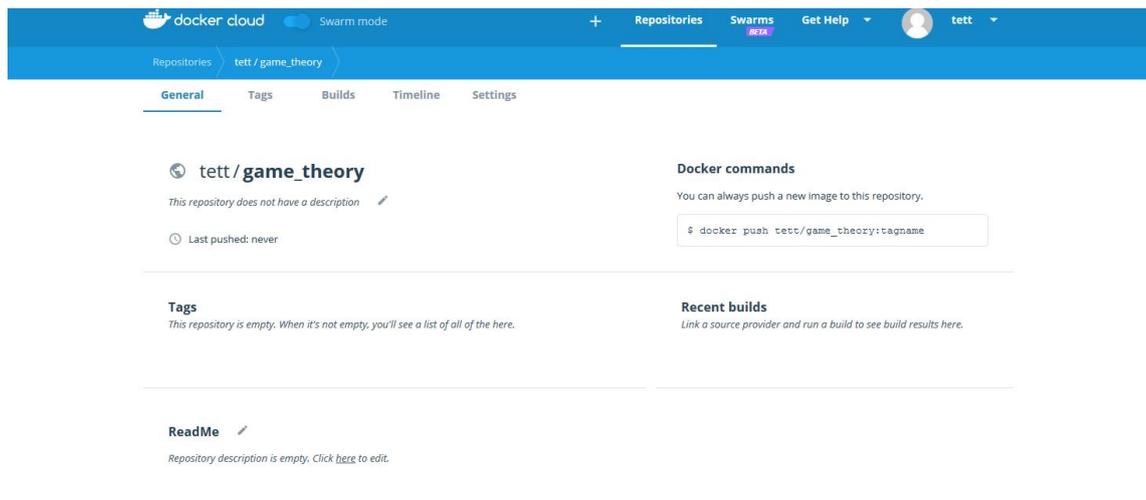


Figure 14 My repository in Docker Cloud

Next step is to login into Docker Hub. You will be able to see your image in both Docker Cloud and Docker Hub.

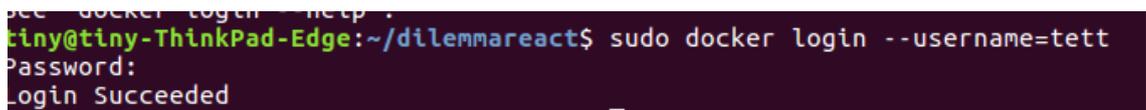


Figure 15 Login into Docker Hub

Push an image.

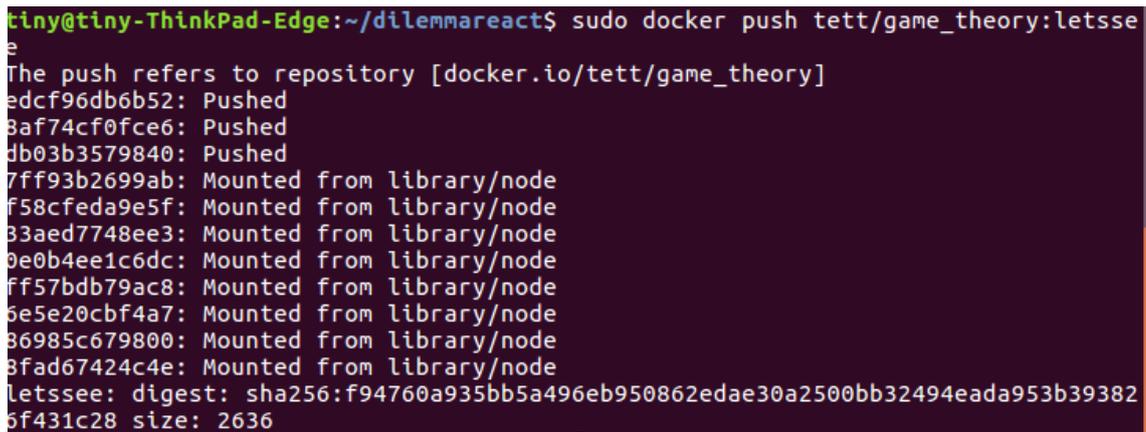


Figure 16 Image push

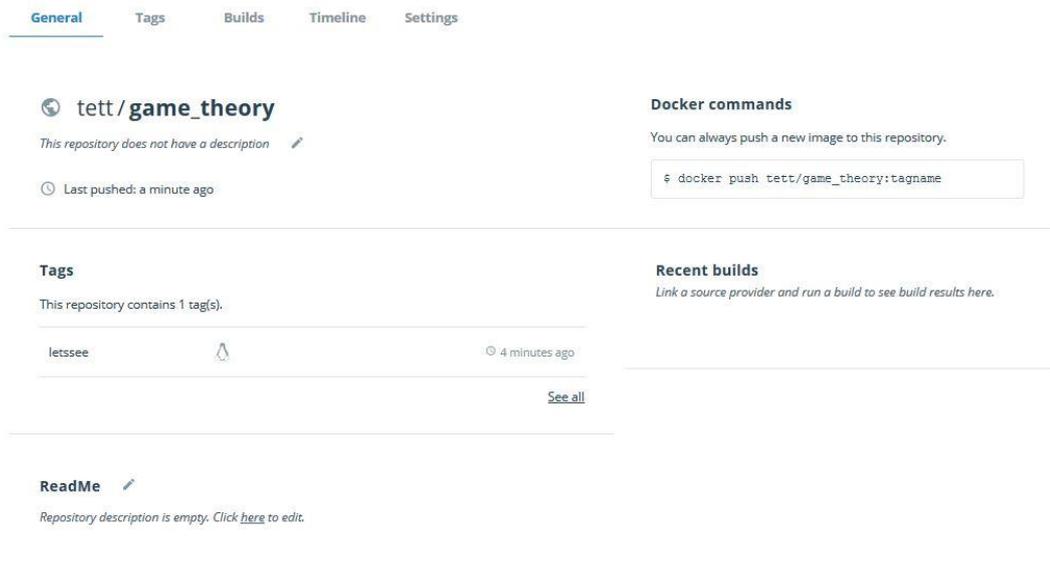


Figure 17 Docker Cloud repository interface

If you wish to find my image in Docker Hub just search for `game_theory`, this is what you can find.

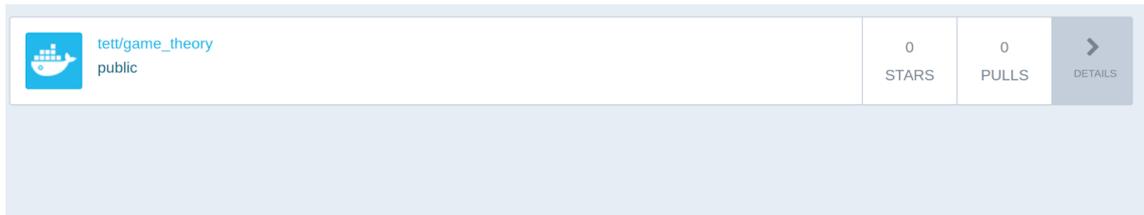


Figure 18 Docker Hub repository interface

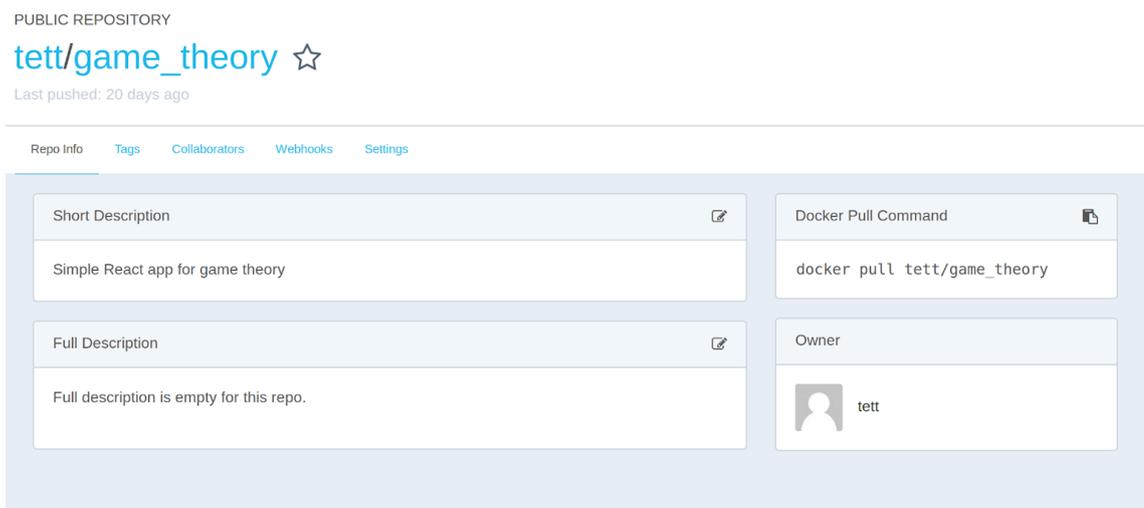


Figure 19 Repository interface

10 Docker Networks

```

tiny@tiny-ThinkPad-Edge:~$ sudo docker network ls
[sudo] password for tiny:
NETWORK ID          NAME                DRIVER              SCOPE
37839776a07b       bridge             bridge             local
140d61ad1e16       host               host               local
3353fa15e353       none              null               local

```

Figure 20 Docker networks

Docker has three built in networks: [13]

- bridge
- host
- none

You can specify which network you want to use with `--net` command.

None network means that the container is isolated. To make it run use:

```
$docker run -d --net none [your image]
```

```

tiny@tiny-ThinkPad-Edge:~$ sudo docker run -d --net none tett/game_theory:letsse
e
2dfb4382a13fce5ff5a1fc54f423e58f6520bbc6bcd4df80bf0ea2d3dea764af

```

Figure 21 Creating none network

None network provides the most protection.

```

tiny@tiny-ThinkPad-Edge:~$ sudo docker exec -it ecd867951034c8577a8a20656625a66
2aed91267a72ead6922e6dc76c5bb937a /bin/bash
root@ecd867951034:/my-app# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
ping: sending packet: Network is unreachable

```

Figure 22 Checking if none network is isolated

Containers in the same network can connect to each other. Containers from other networks can't connect to containers in the given one.

Bridge is the default network. If network is not specified this the the network type you are creating. Usually this kind of network is created in single containers which need a way to communicate.

Host is a network that removes network isolation between the container and the Docker host. Least protected network. This kind of containers are usually called open containers.

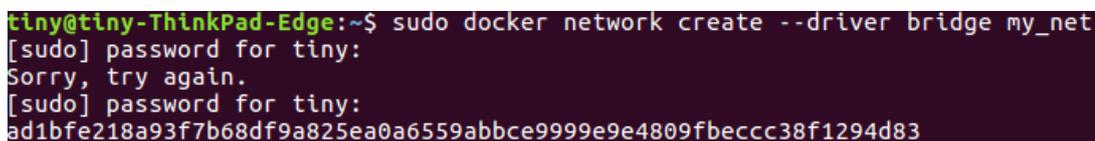
Overlay network connect multiple Docker daemons and enables swarm services to talk to each other. It allows communication for two single containers on different Docker daemons.

Macvlan network allows assigning a MAC address to your container, which shows your container as a physical device on your network. It's usually best choice when dealing with application that have to be directly connected to the physical network.

11 Creating a custom network

In order to create a custom network use command: [14]

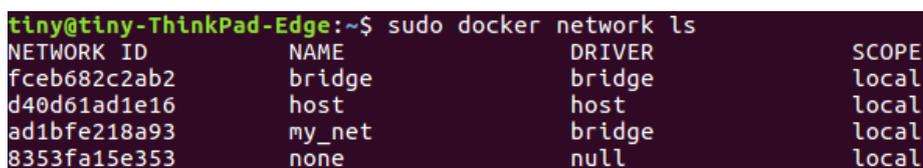
```
$docker network create --driver [you driver choice] [your network]
```



```
tiny@tiny-ThinkPad-Edge:~$ sudo docker network create --driver bridge my_net
[sudo] password for tiny:
Sorry, try again.
[sudo] password for tiny:
ad1bfe218a93f7b68df9a825ea0a6559abbce9999e9e4809fbcecc38f1294d83
```

Figure 23 Creating custom network

If we check networks again:



```
tiny@tiny-ThinkPad-Edge:~$ sudo docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
fceb682c2ab2       bridge             bridge              local
d40d61ad1e16       host               host                local
ad1bfe218a93       my_net             bridge              local
8353fa15e353       none               null                local
```

Figure 24 All the networks available

In a bridge network containers can have access to two network interfaces.

- Loopback interface
- Private interface

Containers in the same network can communicate with each other. We can define networks in docker-compose file as well.

```
1  version: '3'
2  services:
3      example:
4          build: .
5          ports:
6              - "3000:3000"
7          depends_on:
8              - redis
9          networks:
10             - sub
11     redis:
12         image: "redis:alpine"
13         networks:
14             - sub
15     networks:
16         sub:
17             driver:bridge
```

Figure 25 Including network into Docker-compose file

Networks are defined similar to other services, with sub being the name of my network. Network should be defined as well in other sections where it is being used.

You can create two networks which will provide network isolation between services.

```
1  version: '3'
2  services:
3    example:
4      build: .
5      ports:
6        - "3000:3000"
7      depends_on:
8        - redis
9      networks:
10       - fox
11   redis:
12     image: "redis:alpine"
13     networks:
14       - sub
15   networks:
16     sub:
17       driver:bridge
18     fox:
19       driver:bridge
```

Figure 27 Including more networks into Docker-compose

12 Using Docker in production

Opinions are divided whether it is safe to use Docker in production environment or not. Main concerns are that Docker is missing important security and data management. On the other hand, Docker is being developed at a very fast pace. In the case studies mentioned previously it can be verified that Docker in production can work and is in fact quite efficient.

Technically it is possible to run many different processes in one container, but it is better to run one specific process in each. It is easier to use containers with only one functionality. You can always spin up container to use in some other project, but you can't really spin up container which already has your database and you don't need it in another place. It is also easier to debug and find mistakes in one component out of the whole application than the whole application. Benefit of docker containers is their small size, so it is good to keep it that way especially when many containers have to be deployed and updated at the same time. The most important part though is to remember about security.

Once you deploy your containers to production be careful of the network vulnerabilities and make sure your data is protected.

13 Conclusion

The purpose of this thesis was to document learning of the Docker technology and research its apparent success. As a result, I can sum up that Docker is a very powerful tool which helped many companies to overcome their difficulties in resource management, isolation of environments, security issues and moving into the cloud. Since information is being received and sent as fast as ever before it is essential for services providers to ensure that they can give the best assistance for the customers.

Documenting my learning was not easy, since I had to remember to take screenshots of the code. In my opinion it is very important to see an example of a code as a beginner either as a picture or a short video. I tried to make my explanations as easy as possible for new Linux users as well. I covered all the topics necessary in order to be able to run Docker in test environment.

Overall, I think that Docker documentation and pool of developers provides good support for new Docker users.

References

- 1 Tao W. Docker Technologies for DevOps and Developers. E-material. <<https://www.udemy.com>>. Watched 10.09.2017
- 2 Rani Osnat. A Brief History of Containers: From the 1970s to 2017. E-material. <<https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016>>. Read 25.03.2018
- 3 ArchWiki. Change root. E-material. <https://wiki.archlinux.org/index.php/change_root>. Read 25.03.2018
- 4 Nick Martin. A brief history of Docker Containers' overnight success. E-material. <<https://searchservervirtualization.techtarget.com/feature/A-brief-history-of-Docker-Containers-overnight-success>>. Read 28.03.2018
- 5 Docker documentation. About Docker CE. E-material. <<https://docs.docker.com/install/>>. Read 10.09.2017
- 6 Tao W. Docker Technologies for DevOps and Developers. E-material. <<https://www.udemy.com>>. Watched 10.09.2017
- 7 Docker documentation. Definition of Docker Hub. E-material. <<https://docs.docker.com/search/?q=docker%20hub>>. Read 11.09.2017
- 8 Docker documentation. Docker customers. E-material. <<https://www.docker.com/customers>>. Read 20.09.2017
- 9 Benjamin Wootton. Who's using Docker? E-material. <<https://www.contino.io/insights/whos-using-docker>>. Read 25.09.2017
- 10 Tao W. Docker Technologies for DevOps and Developers. E-material. <<https://www.udemy.com>>. Watched 10.10.2017
- 11 Josh McMenemy. React and Docker for Development and Production. E-material. <<https://medium.com/@McMenemy/react-docker-for-development-and-production-6cb50a1218c5>>. Read 25.11.2017

- 12 Joel Koh. Docker hub vs Docker cloud. E-material. <<https://stackoverflow.com/questions/42735760/docker-hub-vs-docker-cloud>>. Read 20.03.2018
- 13 Docker documentation. Networking overview. E-material. <<https://docs.docker.com/network/>>. Read 21.03.2018
- 14 Tao W. Docker Technologies for DevOps and Developers. E-material. <<https://www.udemy.com>>. Watched 25.03.2018

