

Miika Moilanen

Deploying an application using Docker and Kubernetes

Deploying an application using Docker and Kubernetes

Miika Moilanen
Bachelor's Thesis
Spring 2018
Business Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
System administration, Business Information Technology

Author: Miika Moilanen

Title of thesis: Deploying an application using Docker and Kubernetes

Supervisor: Jukka Kaisto

Term and year when the thesis was submitted: Spring 2018 Number of pages: 44 + 7

This thesis researches container technologies using Docker and Kubernetes. The main objective is to create a Dockerfile, which forms the base image for the deployment. The image is then used to deploy to Kubernetes. This thesis was commissioned by Sparta Consulting Ltd. The idea came from the application development team with a need to make the deployment process more streamline for testing purposes. Although automation is not included in this thesis, the basis is made from which the creation of the automated deployment pipeline can be started.

The goal of this thesis is to find a quick and efficient way to deploy new versions of the application in a test- and potentially a production environment. The research for this thesis conducted from a practical viewpoint. The most used research method in this thesis was "Fail often and fail fast". Through this kind of thinking the wrong solutions are removed quickly, while the right answers remain.

Keywords: Docker, containers, virtualization, Kubernetes, open-source, Linux, networking.

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Järjestelmäasiantuntemus, Tietojenkäsittely

Tekijä: Miika Moilanen

Opinnäytetyön nimi: Deploying an application using Docker and Kubernetes

Työn ohjaaja: Jukka Kaisto

Työn valmistumislukukausi- ja vuosi: Kevät 2018

Number of pages: 44 + 7

Tämä opinnäytetyö tutkii konttitekniologioita, käyttäen Dockeria ja Kubernetesia. Perusideana on luoda *Dockerfile*, jonka tarkoituksena on tehdä kuvake applikaatiosta käyttöönottoa varten. Kuvake otetaan käyttöön myöhemmin Kubernetes ryppäässä. Idea tähän opinnäytetyö tuli tuotekehitystiimiltä, jolla on tarve nopeuttaa sovelluksen käyttöönotto prosessia testausta varten. Tässä opinnäytetyössä ei käydä automaatioprosessia läpi, mutta se luo perustan josta voi myöhemmin luoda automatisoidun sovelluksen käyttöönottoputken.

Opinnäytetyön tavoitteena on löytää nopea ja kustannustehokas keino käyttöönottaa uudet versiot applikaatiosta testiympäristöön ja potentiaalisesti myöskin produktioympäristöihin asiakkaalle. Tämä opinnäytetyö tehdään käytännönläheisestä näkökulmasta. Opinnäytetyössä eniten käytetty tutkimusmetodi on ”Epäonnistu nopeasti ja usein”. Tämän mallisella ajattelutavalla väärät ratkaisut ovat nopeasti löydetty ja jäljelle jää vain oikea vastaus.

Avainsanat: Docker, kontit, virtualisaatio, Kubernetes, avoin lähdekoodi, Linux

TABLE OF CONTENTS

| | | |
|-------|---------------------------------------|----|
| 1 | INTRODUCTION | 6 |
| 2 | DOCKER | 9 |
| 2.1 | Virtual machines and containers | 10 |
| 2.2 | Docker and Virtual machines..... | 11 |
| 2.3 | Linux containers | 12 |
| 2.4 | Storage drivers | 14 |
| 2.5 | Dockerfile | 14 |
| 2.5.1 | BUILD phase..... | 16 |
| 2.5.2 | RUN phase | 17 |
| 2.6 | Dockerfile best practices | 19 |
| 2.7 | Docker-compose | 20 |
| 3 | KUBERNETES | 22 |
| 4 | KUBERNETES CONCEPTS..... | 24 |
| 4.1 | Cluster..... | 24 |
| 4.2 | Pod..... | 25 |
| 4.3 | Service | 26 |
| 4.4 | Volumes | 26 |
| 4.5 | Volume deletion..... | 27 |
| 4.6 | Namespace | 27 |
| 4.7 | Ingresses..... | 28 |
| 5 | DEPLOYING THE APPLICATION..... | 29 |
| 5.1 | Minikube cluster creation..... | 29 |
| 5.2 | Nginx | 31 |
| 5.3 | Encrypting the traffic..... | 33 |
| 5.4 | Creating a certificate | 34 |
| 5.5 | Deployment | 35 |
| 6 | CONCLUSION..... | 38 |
| | REFERENCES | 40 |

1 INTRODUCTION

Before container technologies, deploying an application usually took quite a long time. The deployment had to be done manually, which cost the company time and resources. When container technologies came more popular with Docker and Kubernetes, the whole process became more streamlined and standardized. The container technologies can be used to automate the deployment process quite effortlessly and therefore the value of a well configured container platform is intangible. Docker is a tool to create an image of an application and the dependencies needed to run it. The image can then later be used on a containerization platform such as Kubernetes.

The two main components used in this thesis are Docker and Kubernetes. Docker is used to create a *Dockerimage* of the application by using a Dockerfile. A Dockerfile has all the instructions on how to build the final image for deployment and distribution. The images that are made are reusable perpetually. The image is then used by Kubernetes for the deployment. The benefits of Docker are, for example, the reusability of once created resources and the fast setup of the target environment, whether it is for testing or production purposes. This is achieved through *container technologies* made possible by Docker and Kubernetes. Container technology is a quite new technology which has been growing for the past five years. (Docker Company 2018, cited 14.1.2018.)

Once the Dockerimage is created with the Docker platform, it is ready to be used with the Kubernetes container platform. With the Docker platform a base image is created, which is then used by the Kubernetes deployment platform. At best this is done with a press of a button. The ease-of-deployment eliminates possible human errors in the process, which makes the deployment reliable, efficient and quick. The reason Kubernetes was selected is its versatility, scalability and the potential automatization of the deployment process. The technologies are still quite new and are being developed every day to improve the end-user experience, which already is enjoyable.

The field of Development and Operations (DevOps) benefit greatly from containerization in the form of automating the deployment. There are several types of software to create a *continuous integration and deployment* pipeline (CI/CD). This enables the DevOps team to deploy an application seamlessly to the targeted environment. Compared to normal virtual machines, containerized platforms require less configuration and can be deployed quickly with the CI/CD

pipeline. Container technologies also solve the problem of software environment mismatches because all the needed dependencies are installed inside the container and they do not communicate with the outer world. This way the container is isolated and has everything it needs to run the application. (Rubens 2017, cited 26.12.2017.)

With containers, all the possible mismatches between different software versions and operating systems are canceled out. It enables the developers to use whichever programming language and software tool they want to, if they can run it without problems inside the container. This combined with the deployment process, makes the whole ordeal agile, highly scalable and most importantly, *fast*. With Docker and Kubernetes, it is possible to create a *continuous integration- and deployment-* pipeline which for example guarantees a quickly deployed development version of an application to test locally. (Janmyr 2015, cited 4.1.2018.)

Workflow is seen in Figure 1. The whole installation is based on Ubuntu 16.04, on top of which a hypervisor installed in addition to Nginx. Inside the hypervisor Docker and Kubernetes Minikube and Kompose are installed. With the Dockerfile, according to specific instructions, the base image for the deployment is built. The image is then used in docker-compose.yml as the base image. The docker-compose.yml file is used by Kompose to deploy the application to Minikube. After the deployment is finished, an Nginx reverse proxy must be configured to redirect traffic to Minikube. A TLS- certificate was installed to enforce HTTPS- traffic. After opening the ports 80 and 443 as ingress and egress in the firewall configuration, traffic could access Minikube. The last step was to create an ingress controller and an ingress resource. The purpose of the ingress controller is to create an open port in the edge router between Minikube and the application. Now the application is accessible.

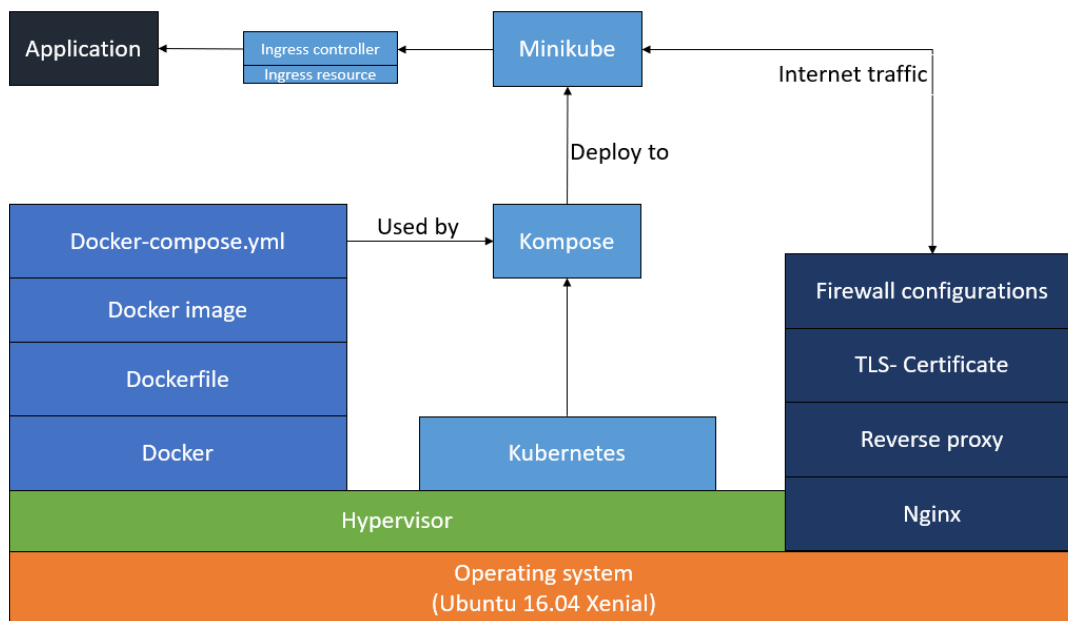


FIGURE 1. Workflow and description of resources used to achieve the result.

The company that assigned the thesis is Sparta Consulting Ltd. The company itself was founded in 2012. Sparta operates in the Information Management (IM) and Cyber Security consultancy business. In addition to support services in this junction, they have also developed a software product to support them in this. Currently there are below 50 Spartans. The employees mostly consist of consultants but also includes a small development and deployment team. The company has two main locations, the headquarters reside in the heart of Helsinki and the development team is located in Jyväskylä, Central Finland.

The consultants advise enterprises in areas like information management, business development and cybersecurity. A "Spartan" advises enterprises in which direction to take regarding these three areas. They do consulting in an ethical way, delivering worthwhile solutions to companies.

2 DOCKER

Docker is a tool that promises to easily encapsulate the process of creating a distributable artifact for any application. With it comes the easy deployment of an application at scale into any environment and streamlining the workflow and responsiveness of agile software organizations. (Matthias & Kane 2015, 1.) When talking about Docker containers, many people connect them to virtual machines however, this is neither completely right or completely wrong. Discerning these two concepts is challenging. While it is true that both are a certain type of virtualization, the difference between them is that containers are built to contain only the necessary libraries and dependencies inside them. Their bulkier counterpart, virtual machines, start off with a full operating system and all included software that come with them. (Docker eBook 2016, 3.)

One of the most powerful things about Docker is the flexibility it affords IT organizations. The decision of where to run your applications can be based 100% on what's right for your business. -- you can pick and choose and mix and match in whatever manner makes sense for your organization. -- With Docker containers you get a great combination of agility, portability, and control. (Coleman, 2016).

Compared to virtual machines, containers are reproducible standardized environments that follow a certain ideology, create once – use many. After a container environment with the Dockerfile is manually crafted, it is available to be utilized when needed. The containers take only seconds to deploy, while virtual machines take significantly more time.

For instance, the Finnish railway company VR Group, uses Docker to automate the deployment and testing process. Their problems were high operating costs, quality issues and a slow time-to-market process. After implementing the Docker EE (Enterprise Edition), their average cost savings were increased by 50%. The logging and monitoring was easier for all used applications. Standardizing the applications on one platform enables the usage everywhere. A delivery pipeline was set up, which works for the whole platform. This enables easy implementation of new items to the same environment. (Docker VR 2017, cited 9.2.2018)

2.1 Virtual machines and containers

When comparing container technologies to normal virtual machines, there are several upsides to it. Containers are fast to set up and configure, while virtual machines are more bulky and slow to set up and configure. Since the containers nature is very agile, they provide a good basis for development & operations (DevOps) processes. To update a new version of the application, a CI/CD- pipeline can be utilized. This way the application is quickly installed on the target environment. This enables the fast testing of the newly developed versions of the application and pushing new versions to production environments. Even though containers and virtual machines are quite different, they can be combined to get the good sides of them both. The robustness of the virtual machine and the agility of containers, provide a good basis for the deployment process. (Docker container 2018, cited 13.1.2018.)

“Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers.” (Docker container 2018, cited 13.1.2018). Virtual machines are built on top of a *hypervisor*, which allows several virtual machines to run on one machine (FIGURE 2). Each virtual machine instance contains a full copy of an operating system and all the dependencies needed to run, which take up several gigabytes of storage space. Virtual machines also take several minutes to boot up. (Docker container 2018, cited 13.1.2018.)

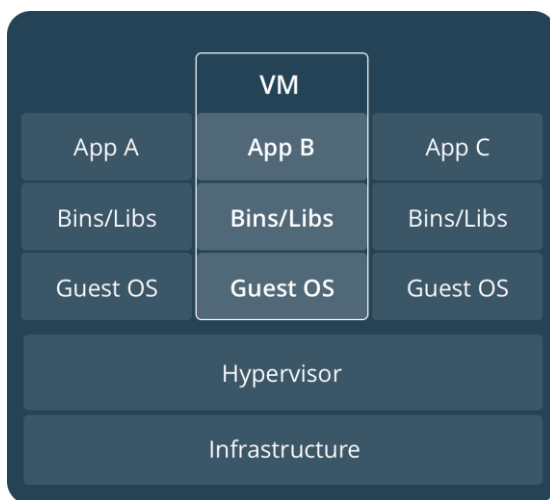


FIGURE 2. Depiction of VM architecture (Docker container 2018, cited 13.1.2018)

According to the Docker container introduction page, containers are an abstraction at the application layer that packages code and dependencies together. Containers are built on top of the

Host operating system (FIGURE 3). Several of them can be run simultaneously and they share the same OS (Operating System) kernel with other containers. Each container runs as an isolated process in user space, which means that there is not necessarily communication between them. Containers are typically only megabytes in size and boot up virtually instantly. (Docker container 2018, cited 13.1.2018).

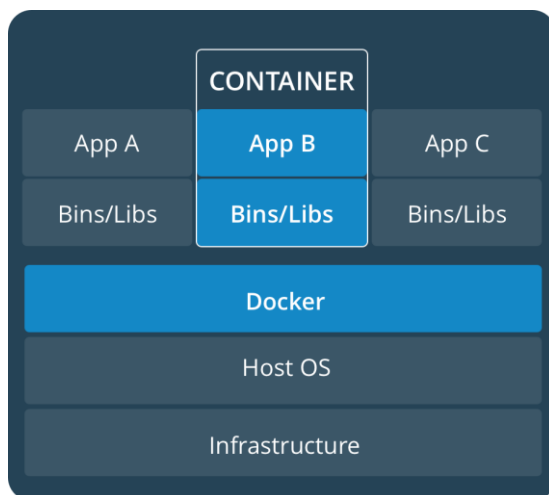


FIGURE 3. Depiction of container architecture (Docker container 2018, cited 13.1.2018)

2.2 Docker and Virtual machines

Instead of using Docker as a standalone process, they can be combined with a virtual machine. All hypervisors are a good platform for the Docker host: VirtualBox, Hyper-V, AWS EC2 Instance. No matter the hypervisor, Docker will perform well. Sometimes a virtual machine might be the place to run the docker container (FIGURE 4), but you don't necessarily need to. The container can be run as a stand-alone service on top of bare metal (Docker eBook 2016, 5; Coleman 2016, cited 18.01.2018.)

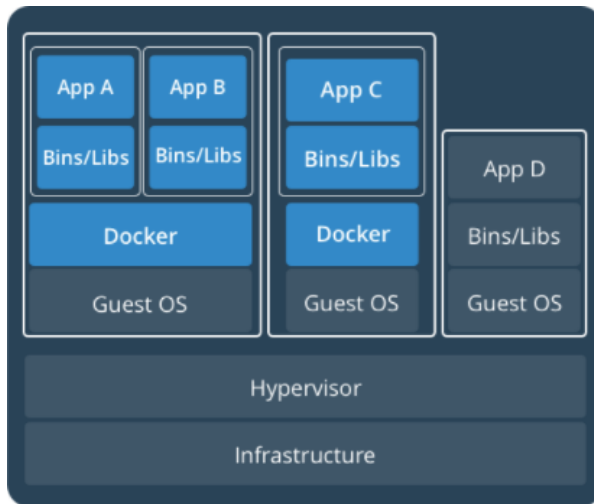


FIGURE 4. Docker built inside a virtual machine. (Docker container 2018, cited 13.1.2018).

During the early years of virtual machines, they gained popularity for their ability to enable higher levels of server utilization, which is still true today. By mixing and combining Docker hosts with regular virtual machines, system administrators can maximize efficiency from their physical hardware. (Coleman 2016, cited 18.01.2018). Building a container cluster on top of a virtual machine, whether it was made with Docker Swarm or Kubernetes, enables the usage of all the resources provided by the physical machine to maximize performance.

2.3 Linux containers

When Docker first came out, Linux- based containers had been there for quite some time and the technologies it is based on are not brand new. At first, Dockers' purpose was to build a specialized Linux Container (LXC). Docker then detached itself from it and created its own platform. The predecessor of Docker, LXC, had been around for almost a decade and at first Docker's purpose was to build a specialized LXC container. (Matthias & Kane 2015, 4; Upguard 2017, cited 13.1.2018). At the most basic level when comparing these two technologies there are some similarities (TABLE 1). They both are light use-space virtualization mechanisms and they both use cgroups (control groups) and namespaces for resource management. (Wang 2017, cited 18.01.2018).

TABLE 1. Similarities between LXC and Docker (Rajdep 2014, slide 25/30).

| Parameter | LXC | Docker |
|----------------------|---|--|
| Process isolation | Uses PID namespace | Uses PID namespace |
| Resource isolation | Uses cgroups | Uses cgroups |
| Network isolation | Uses net namespace | Uses net namespace |
| Filesystem isolation | Using chroot | Using chroot |
| Container lifecycle | Tools lxc-create, lxc-stop, lxc-start to create, start and stop a container | Uses docker daemon and a client to manage the containers |

When it comes to understanding containers, the concepts of *chroot*, *cgroup* and *namespace* must first be understood. They are the Linux kernel features that create the boundaries between containers and the processes running on the host. (Wang 2017, cited 18.01.2018.) According to Wang, the Linux namespaces take a portion of the system resources and give them to a single process, making it look like they are dedicated to that specific process.

Linux control groups, or *cgroups*, allow you to allocate resources to a specific group of processes. If there is, for example, an application that uses too much of the computers resources, they can be reallocated to a *cgroup*. This way the usage of CPU cycles and RAM memory can be determined.

The difference between namespaces and *cgroups* is that namespaces only deal with a single process and *cgroups* allocate resources for a group of processes. (Wang 2017, cited 18.01.2018.) By allocating resources per each process or a group of processes, it is possible to scale up and down the needed amount of resources when, for example, during traffic peaks. This makes the utilization of processing power of the physical computer possible and more importantly, efficient.

Chroot (change root) is used to change the working directory of the application. Its purpose is to isolate certain applications from the operating system. This is called a *chroot jail*. This is especially

handy when a program is tested that could potentially harm the computer or is insecure in some way. An important thing to remember is disabling root permissions from the application which is placed inside the jail, so that it cannot run privileged commands. Other potential use cases are, for example, running 32-bit applications on 64-bit operating systems, executing old versions of certain applications on modern operating systems. (Ubuntu 2015, cited 12.2.2018.)

2.4 Storage drivers

“For image management, Docker relies heavily on its storage backend, which communicates with the underlying Linux filesystem to build and manage the multiple layers that combine into a single usable image.” (Matthias & Kane 2015, 44). In case the operating systems’ kernel supports multiple storage drivers, Docker has a list of usable storage drivers, if no driver is configured separately. By default, the Docker Community edition uses the overlay2- filesystem, which provides a quick copy-on-write system for image management (Docker storage 2018, cited 17.01.2018).

The default filesystem was used in this project because as there was no need to switch to a different one. Depending on the operating system Docker is installed on, some filesystems might not be enabled, and need specific drivers installed. When having doubts choosing the right storage driver, the best and safest way is to use a modern Linux distribution with a kernel that supports the overlay2 storage driver. (Docker storage 2018, cited 17.01.2018.)

There are generally two levels regarding the storage drivers, *file and block* level. For example, the filesystem drivers *aufs, overlay, and overlay2* operate at the file level. They use memory more efficiently, but in turn the writable layer of the container can grow unnecessarily large in write-heavy workloads. The default overlay2- driver was chosen since there are not any write-heavy workloads in the project. The *devicemapper, btrfs* and *zfs* are block-level storage drivers and they perform well on write-heavy workloads (Docker storage 2018, cited 17.01.2018.)

2.5 Dockerfile

To build an image from an application a Dockerfile is needed. The purpose of the Dockerfile is to automate the image building process, in which all the necessary dependencies and libraries are installed. This project required the configuring of a multi-stage Dockerfile. This enables defining

multiple build stages in the same file. It makes the build process more efficient through reducing the size of the final image, which is determined in the last phase of the build (FIGURE 5) (Dockerfile 2017, cited 25.12.2017.) In the first phase all the needed dependencies are installed and compressed into an *artifact*. An artifact in Linux environments is usually a compressed .tar.gz- file. In the second phase the artifact is taken and extracted. The result is a ready Docker image, which can be further used in a deployment.

```
[momi@localhost sentinel]$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
app                  test               d35884a2b995      17 minutes ago     245MB
<none>              <none>            60a7f5279606      18 minutes ago     1.51GB
```

FIGURE 5. Images built with multiphase dockerfile, first and second phase.

Although premade images are available for use from the Docker Hub (<https://hub.docker.com/>), it is sometimes better to make a specific Dockerfile. This way, you know how the final image is built, what licenses and/or properties it contains. The downside of this is that the responsibility to keep the Dockerfile updated falls on the organization. It is also possible to combine prepared images with self-made Dockerfiles to maximize efficiency.

The Dockerfile supports 13 different commands, as seen in TABLE 2, which tell the Dockerfile how to build the image and how to run it inside a container. There are two phases in the building process: *Build* and *Run*. In the BUILD -phase you determine the commands which are executed during the build process. In the RUN- phase the commands specified are run when the container is run from the image. The two commands *WORKDIR* and *USER* can be used by both phases. (Janmyr 2015, cited 4.1.2018.)

TABLE 2. Instructions used by Dockerfile. (Janmyr 2015, cited 4.1.2018.)

| BUILD | Both | RUN |
|------------|---------|--------|
| FROM | WORKDIR | CMD |
| MAINTAINER | USER | ENV |
| COPY | | EXPOSE |

| | | |
|-----|--|------------|
| ADD | | VOLUME |
| RUN | | ENTRYPOINT |

2.5.1 BUILD phase

The FROM command tells the Dockerfile where to get the image for the for the build. Depending on the use case, the Dockerfile can start FROM *scratch*, which starts the container without any operating system. Normally, an operating system is pulled from the Dockerhub to act as a basis for the final image. Many popular Linux distributions have their own official Dockerfiles there, such as Ubuntu, Centos, Debian, Alpine and CoreOS. (Dockerfile 2017, cited 25.12.2017.)

The MAINTAINER command simply dictates the name and e-mail of the author. The purpose of this command is to notify the end-user who to contact in case of problem situations. It is good to keep in mind that the author in question might not keep the Dockerfile updated and does not reply to questions. In this case, it is best to create a new Dockerfile. (Dockerfile 2017, cited 25.12.2017.)

COPY command is used to copy items from the sources system to the target destination inside the container. It can also be used to specify multiple sources. In the case of this project, the folder where the application resided, was copied. The command: *COPY ./ /root* (FIGURE 6) states that all the contents of the current folder, are copied to the root path inside that image. (Janetakis 2017, cited 18.01.2018.)

The ADD command is the same as COPY, but with ADD, the extraction of a single tar- file is possible from the source to the destination. In addition, an URL- address can be used instead of a local file. Use cases might be when extracting a remote TAR- file into a specific directory in the Docker image. (Janetakis 2017, cited 18.01.2018.)

The RUN command executes commands inside a new layer on the current image. The results are then committed and used for the next step inside the Dockerfile. This is used when installing important dependencies or updates. For example, if the base image of the Dockerfile, usually an operating system, needs to be updated so a command needs to be executed: *RUN apt-get upgrade*. (Dockerfile 2017, cited 25.12.2017.)

The ONBUILD command adds a trigger instruction into the image, which is executed later. This is useful when building an image which is used later as a base for other images. For example something like this could be added to the Dockerfile: ONBUILD ADD ./app/src. What this does, is that it registers advance instructions to run in the next build stage. (Dockerfile 2017, cited 25.12.2017.)

2.5.2 RUN phase

The purpose of the CMD instruction is to provide a command, which is executed when the container is built from the final image. There can only be one CMD instruction, and if there are several the latest one is used. A use case for this specific instruction might be, for example, running a command shell inside the container: CMD /bin/bash. (Matthias & Kane 2015, 44.)

The ENV instruction is meant to define the environment of the application running inside the container. These are heavily application specific, for example when determining users and passwords for a PostgreSQL instance. These are best specified inside a docker-compose.yml file, if one is used in a project. (Matthias & Kane 2015, 43.)

The EXPOSE instruction tells docker which ports the container listens to during runtime. The port being listened to can be specified as TCP or UDP. This instruction does not publish the port, it just informs which port will be published. (Dockerfile 2017, cited 25.12.2017.) In this project, the port assignment is done within the *docker-compose.yml* file. (APPENDIX 7.)

“The VOLUME instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.”. (Dockerfile 2017, cited 25.12.2017). It can be used to mount for example a shell script, which is then run in the docker-compose up phase. Volumes can be also used in docker environments to create a persistent storage for important information such as user databases.

The ENTRYPOINT instruction allows the user to configure a container that runs as an executable process. Like the CMD instruction, when listing several entrypoints only the last one will have an effect in the Dockerfile. Entrypoint can be used in specific scenarios, where the container needs to behave as if it was the executable it withholds. It is used when the end user is not allowed to

override the specified executable. (Dockerfile 2017, cited 25.12.2017; DeHamer 2015, cited 10.2.2018.)

For this project, not all Dockerfile instructions were used. The CMD command for instance was moved to the *docker-compose.yml* file. This was done so that command can be easily modified without rebuilding the image. This saved tons of time in the process.

Here is the Dockerfile created for this project (FIGURE 6). For convenience and compatibility uses, the image *nikolauska/phoenix:1.5.3-ubuntu* was pulled straight from the Docker hub (<https://hub.docker.com>) to be used in this project. This made the deployment process easier due to having a ready image on which to base the build. The *mix* commands are specific to the application and not related to this thesis, so they will not be explained.

```
FROM nikolauska/phoenix:1.5.3-ubuntu

#Build release
COPY ./ /root
WORKDIR /root

#Set ENV variables
ENV MIX_ENV=prod \
    COOKIE=secret \
    NAME=app

#Dependencies
RUN rm -rf deps _build \
    && mix deps.get --only=prod \
    && mix compile \
    && mix npm.clean \
    && mix npm.install \
    && mix release \
    && rm -rf /var/lib/apt/lists/*

FROM ubuntu:16.04

#Updates, install packages
RUN apt-get update && apt-get upgrade -y \
    && apt-get install -y locales tar openssl \
    && locale-gen en_US.UTF-8

#Copy TAR file from first phase of build and run it
WORKDIR /root
COPY --from=0 /root/app/app.tar.gz .
RUN tar -xzvf app.tar.gz -C /root \
    && rm app.tar.gz
```

FIGURE 6. Instructions for the Dockerfile made for this project

2.6 Dockerfile best practices

Docker images are layered. When building an image, Docker creates a new intermediate container for each instruction described in the Dockerfile. When the commands are chained together to form a coherent line of build instructions, it reduces the build time and resources the Dockerfile might use (FIGURE 7). (Dockerfile guide 2017, cited 25.12.2017.)

```
#Un-optimized
RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y locales
RUN apt-get install -y tar
RUN apt-get install -y openssh

#Optimized
RUN apt-get update \
&& apt-get upgrade -y \
&& apt-get install -y locales tar openssh
```

FIGURE 7. Chaining commands in a Dockerfile

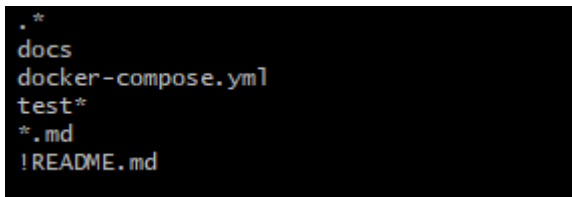
Each of the RUN commands described in the “Un-optimized” part, create their own intermediate container. Each of them takes time to setup making the build process slower than the optimized version, which only creates one layer to handle the instructions. (Dockerfile guide 2017, cited 25.12.2017.)

Using multistage builds help decrease the final image size. In the example before, the first stage of the build is where all the dependencies and libraries are installed, the application is built and compressed into an *artifact*. In the second phase of the build the artifact is taken and unpacked. The application itself contains specific information on how to run it, and it will be determined in the docker-compose.yml file (APPENDIX 7). The image contains the needed dependencies to run the application, and it will be referred to in the docker-compose.yml file.

When configuring the Dockerfile, a plan should be made what the final image needs and doesn't need. For instance, there is no need for a PDF reader inside a database instance. Minimizing the number of unnecessary packages is one of the biggest goals. It helps make the image fast to use, and efficient when thinking about processing power. (McKendrick & Gallagher 2017, 36; Dockerfile guide 2017, cited 25.12.2017.)

The purpose of the `.dockerignore` file is to exclude the unwanted files which are not needed in the docker build process. The build context is what the Dockerfile uses to build the final image. By using a `.dockerignore` file, all the irrelevant items can be left out of the build context. (Dockerfile guide. 2017). They will seem quite familiar to people who have been working with `.gitignore` files in GitHub. (Cane 2017, cited 18.01.2018; McKendrick & Gallagher 2017, 36.)

To leave out items of the build context, for example the `docker-compose.yml` file, just simply type in the name of the file. To add items to the build context, an exclamation mark (!) can be added before the filename (FIGURE 8). The file is read from top to bottom, which means that the instructions on top will be executed first. As an example: when telling the file to leave out all markdown files (`*.md`), a specific file can be added with the exclamation mark flag (`!README.md`) (Cane 2017, cited 18.01.2018).



```
*  
docs  
docker-compose.yml  
test*  
*.md  
!README.md
```

FIGURE 8. Example of a `.dockerignore` file. (Cane 2017, cited 18.01.2018).

2.7 Docker-compose

“Compose is a tool for defining and running multi-container Docker applications.” (Docker Compose. 2017). When thinking about Docker Compose and Dockerfiles, in a sense they are quite similar: Both have two sets of unique instructions for building images and running containers. (Podvaznikov 2017, cited 11.01.2018.) After the successful building of the Docker image, the `docker-compose.yml` file can be configured.

The purpose of the file is to build and link the containers together. This is used, for example, when making a multi-container application such as linking an application to one or multiple databases. In this thesis the actual `docker-compose up` command is not used. The Kubernetes’ own tool, `kompose`, is used. It is very similar to `docker-compose` but instead of building the image straight from the `docker-compose.yml` file, it first translates it into Kubernetes readable resources. After translating, it can then use the instructions to deploy the applications inside containers. (Kubernetes

kompose 2018, cited 14.1.2018.) An example docker-compose.yml file (APPENDIX 7) in which an application (app) is linked with three database instances (DB1, DB2, DB3)

3 KUBERNETES

“Kubernetes is an open-source platform designed to automate deploying, scaling, and operating application containers.” (Kubernetes 2018, cited 14.1.2018). Kubernetes is the leading container orchestration engine in the field of containerization. Developed by Google, it uses the Docker images as a basis to deploy applications into the containers. With Kubernetes, the containers are easily scaled up, destroyed and remade. Compared to normal virtual machines, they are deployed faster, more efficiently and reliably. Docker creates the image, which is used in Kubernetes. In the world of growing virtualization and the Internet of Things, applications and services need to be deployed quickly and efficiently. This is where Kubernetes comes in.

Instead of operating at the hardware level and rather in the container level, Kubernetes provides some features related to a Platform as a Service (PaaS). PaaS usually refers to a cloud service. In the cloud service, the user can develop, deploy and run applications using several environments provided by the service provider. Essentially, the provider takes all the responsibility in installing and configuring the environment. This way the customer is free to apply the application code to the cloud. This is what Kubernetes basically does, except that the cluster is managed by the developer or cluster administrator and it can be done locally. These features include for example, deployment, scaling, load balancing, logging and monitoring. Kubernetes comprises of a set of independent control processes that drives the current state of the deployment to the wanted result. (Kubernetes 2018, cited 14.1.2018; Meegan 2016, cited 12.2.2018.)

The reason Kubernetes was chosen instead of the native Docker cluster, Docker Swarm, is its scalability, portability and self-healing attributes. Kubernetes has been around longer than Docker Swarm and therefore has much more documentation. It also has more widespread 3rd party application support available. (Kubernetes 2018, cited 14.1.2018; Docker 2017, cited 13.1.2018.)

The popular game *Pokémon GO*, uses Kubernetes containers. They were especially critical during the launch, where millions of users connected almost simultaneously. The actual traffic that had to be handled was 50 times larger compared to the expected traffic. This is where the scalability of Kubernetes kicks in. When the traffic increases rapidly, Kubernetes sees this and automatically deploys more containers to adapt. Deployed on the Google Cloud, it was a great success and tens

of thousands of cores were provided dynamically to enable an enjoyable user experience to the end user. (Stone 2017, cited 9.2.2018.)

4 KUBERNETES CONCEPTS

Before working with Kubernetes, it's important to know some of the basic concepts concerning Kubernetes architecture. It is easier to work when the basic concepts are known when starting on a new subject. The concepts used in the deployment process are described in the next chapters which are the following: Cluster, Node, Master, Pod, Service, Volume, Namespace and Ingress.

4.1 Cluster

“A cluster is a collection of hosts storage and networking resources that Kubernetes uses to run the various workloads that comprise your system.” (Sayfan 2017, cited 25.12.2017). The *Minikube single node cluster*, which is used in this project, consists of three resources: The master, which coordinates the cluster. The second part are the *nodes* which manage run the *kubelet*, *kube proxy* and the *Docker engine*. (FIGURE 10). (Kubernetes Cluster 2018, cited 14.1.2018.)

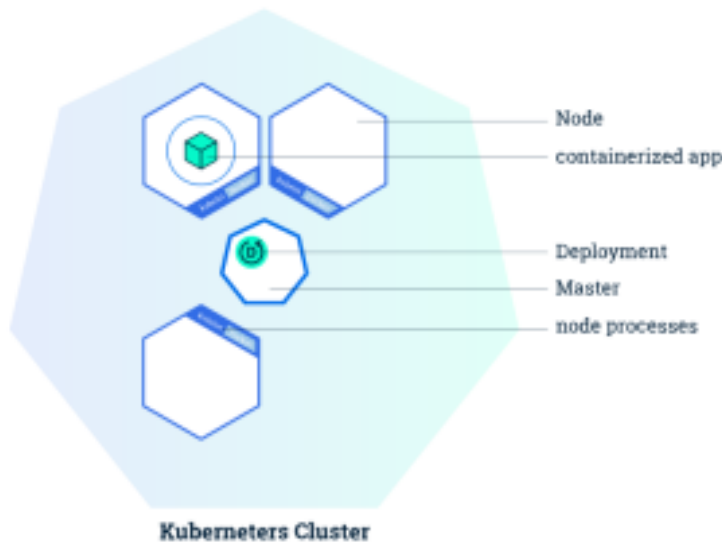


FIGURE 9. Kubernetes cluster depicted. (Kubernetes Cluster 2018, cited 14.1.2018.)

The *master* is the head of Kubernetes cluster. It consists of multiple parts, such as an API server, a scheduler and a controller manager. “The master is responsible for the global cluster-level scheduling of pods and handling events.” (Sayfan 2017, cited 25.12.2017). All the instructions

made via *kubectl* go through the API server, which are then redirected to the designated worker nodes.

The worker units inside the clusters are called *nodes*. A node is a single host inside the cluster, it can be a physical or a virtual machine. Their purpose is to manage *Pods*. Each node runs several Kubernetes managed components, such as *kube proxy* and *kube ingress*. All the nodes are managed by the Kubernetes master and their job is to do all the work given by the Kubernetes master. (Sayfan 2017, cited 25.12.2017).

4.2 Pod

A pod consists of one or more containers, with shared network and storage and instructions on how to run containers. Pods are managed by the nodes. They are the smallest deployable units which can be created in Kubernetes. "Containers within a pod share an IP address, a port space and can find each other via localhost". Pods can be used to create vertical application stacks, such as a LAMP-stack, although their main purpose is to support co-located and managed helper programs, like proxies, file and data loaders, log and checkpoint backups. (Kubernetes pod 2018, cited 14.1.2018; Kubernetes Cluster 2018, cited 14.1.2018.)

When pods are destroyed, they are not resurrected. Specifically, *ReplicationControllers* are designed to create and delete pods dynamically when, for example, commencing rolling updates or scaling the deployment up or down. (Kubernetes service 2018, cited 14.1.2018.) This enables for the processes running inside the pod to have a good uptime, when in a disaster situation they are remade by the Replication Controller.

They can also communicate using standard inter-process communications (IPC). Containers in different pods have unique IP addresses and cannot communicate via IPC without specific configuration. They usually communicate via pod IP addresses. (Kubernetes Cluster 2018, cited 14.1.2018).

4.3 Service

The basic idea of services is that they define a policy with which there is a way to gain access to the pods. They take care of the variables needed for communication: an IP address, ports and with a group of pods, also load balancing. When a service and a deployment are linked, the service should be started first and then the deployment. They can be deploying using *one* YAML file. The instructions need to be separated with a line containing several hyphens. (Abbassi 2016, cited 24.1.2018.)

“Services are used to expose some functionality to users or other services.” (Sayfan 2017, cited 25.12.2017). Kubernetes pods don’t have a long lifecycle and when they are taken down, they are not resurrected. Pods are created dynamically when taken down. Because of this, to communicate with newly created pods there is a need for a concept which abstracts away the pod, and this is achieved with services. (Kubernetes service 2018, cited 14.1.2018. Hong 2017, cited 11.01.2018.)

4.4 Volumes

Kubernetes uses PersistentVolumes (PV) to persist data between reboots and system catastrophies. So that the PV can be used, a *PersistentVolumeClaim* (PVC) must be created. Their job is to link the PV to the container for which the PVC is configured. They can be implemented straight on the host, through Network File Shares (NFS), and various other methods. (Kubernetes Storage 2018, cited 23.1.2018.)

For the purposes of this thesis, since we don’t have valuable data which would result in a disaster when lost, we’ll be using dynamically allocated PersistentVolumes with the reclaim policy of “Delete”. To keep the volumes intact after deleting the PersistentVolumeClaims (PVC), the reclaim policy must be changed to “Retain”. This way when the claim is deleted, the volumes are moved to the “released” phase, where the data it withholds can be recovered manually. (Kubernetes Storage 2018, cited 23.1.2018.)

4.5 Volume deletion

The PV's are dynamically created with each deployment with Minikube when the default-storageclass addon is enabled. They are set up according to the specifications made for the *PVC*, which in this case reside inside the `docker-compose.yml` file. Dynamically created PersistentVolumes have a default reclaim policy of "Delete". (Kubernetes volume 2018, cited 22.01.2018.)

The reason for the deletion policy for dynamically created PVs is the goal of automating the lifecycle of storage resources (Kubernetes Storage 2018, cited 23.1.2018). If the PV's didn't delete themselves on PVC deletion, they could quickly flood the storage of the computer and cause problems to the inattentive cluster-administrator. This means that when the PVC's are deleted so are the volumes.

4.6 Namespace

In Kubernetes, you can assign virtual clusters called *namespaces*. They are used to keep different versions of the application separate, such as development and production versions (Kubernetes namespace 2018, cited 16.01.2018). This is to ensure potential mix ups between versions. A namespace is created based on the instructions inside a YAML file with, for example, the following information (FIGURE 11).

The namespaces in Kubernetes are based on Linux namespaces. They have the same kind of working principle. The purposes of these are to allocate resources to a specific namespace along with isolating the applications deployed between different namespaces.

```
apiVersion: v1
kind: Namespace
metadata:
  name: prod
  labels:
    name: prod
```

FIGURE 10. Prod namespace creation instructions

4.7 Ingresses

For Minikube to properly redirect traffic to the exposed services, an *Ingress* is needed. The services and pods inside the cluster network are only accessible internally. All the traffic that try to access the services, is dropped or redirected by an *edge router*. By default, the cluster is isolated from the Internet so that the cluster network is not directly accessible from outside (Oranagwa 2017, cited 16.01.2018; Kubernetes ingress 2018, cited 10.2.2018).

An Ingress allows inbound connection to reach the cluster services through the edge router by creating a “hole” in it. To access the service, an *Ingress resource* must be created. The resources are created via YAML files (FIGURE 12). For the production namespace, the following ingress was created.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: prod-ingress
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: prod.example.domain.com
    http:
      paths:
      - path:
          backend:
            serviceName: prod-app
            servicePort: 4000
```

FIGURE 11. Instructions to create an Ingress resource.

To make the resource work, the cluster must have an Ingress controller. The controller is responsible for taking the resource and using its instructions to redirect traffic to the wanted destination. Minikube versions v0.14.0 and above come with Nginx ingress setup as an addon. It is enabled by typing *minikube addons enable ingress* in the Linux terminal. (Oranagwa 2017, cited 16.01.2018.)

5 DEPLOYING THE APPLICATION

This chapter goes into detail on how the deployment was achieved. The different components used in the process are explained and how they are linked together. The application is deployed with a tool called Kompose on top of a Minikube cluster. To redirect the traffic, a Nginx reverse proxy is configured. The traffic is redirected to the Minikube master node, which in turn redirects the traffic to the application through an *ingress controller*. For the controller to function, an *ingress resource* is created. Its job is to create a hole in the *edge router* of the Minikube cluster.

The application is deployed onto an Ubuntu 16.04 Xenial server. The server itself resides in OpenStack, to which an SSH connection is made from the local workstation. Ubuntu was chosen as the basis for the deployment because the author has some familiarity with it. For the deployment process, the following tools were installed: Docker (appendix 1), Minikube (appendix 2), Kubectl (appendix 3), Kompose (appendix 4), Nginx (appendix 5), basic authorization for the website (appendix 6) and Oracle VirtualBox.

5.1 Minikube cluster creation

Before the Minikube cluster can be created, a hypervisor is needed. In this case, Oracle VirtualBox was installed. The Ubuntu file repositories contain the hypervisor by default, so the installation is easy. It is installed by running `sudo apt-get install virtualbox`. The command then installs the latest version of the hypervisor.

When creating the Minikube single node cluster, the target system needs to have the necessary resources to run it. By default; Minikube uses 1048 GB of RAM and 1 CPU. It can be modified with the following command: `minikube start -cpus=2 -memory=2048`. (Aliakhtar 2016, cited 11.01.2018). For this deployment, only the default resources are needed. If a different hypervisor than Oracle VirtualBox is used, it needs to be determined with the “`--vm-driver=`” (FIGURE 13), however, for this project the default VirtualBox driver was used.

```
Starting local Kubernetes v1.8.0 cluster...
Starting VM...
Downloading Minikube ISO
140.01 MB / 140.01 MB [=====] 100.00% 0s
Getting VM IP address...
Moving files into cluster...
Downloading localkube binary
148.25 MB / 148.25 MB [=====] 100.00% 0s
0 B / 65 B [-----] 0.00%
65 B / 65 B [=====] 100.00% 0sSetting up certs...
Connecting to cluster...
Setting up kubeconfig...
Starting cluster components...
Kubectl is now configured to use the cluster.
Loading cached images from config file.
```

FIGURE 12. Creation of the minikube cluster with the command: `minikube start --vm-driver=virtualbox`

Now that the single node cluster is running, the IP address of the *master node* must be found out. The IP address can be checked IP address by running `minikube ip` (FIGURE 14) or if more detailed information is needed `kubectl cluster-info` can be run (FIGURE 15). The `cluster-info` command shows all the components running in a cluster. For Minikube, there is only the Kubernetes master.

```
miika.moilanen@bitidemo:~$ minikube ip
192.168.99.100
```

FIGURE 13. Fetching Minikube IP address.

```
miika.moilanen@bitidemo:~$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443
```

FIGURE 14. Fetching Minikube IP address with `kubectl`

For the Minikube cluster to communicate with Docker, the following environment variable must be set. This is needed so that Kubernetes can communicate with the Docker daemon to fetch the image used in the deployment process. The variable is set with: `eval $(minikube docker-env)`. (Kubernetes quickstart 2018, cited 11.01.2018.)

After setting the variable, the image can be loaded to the Docker daemon. To achieve this, the secure copy command (`scp`) was used to transfer the docker image. The image was moved from a local virtual machine to the target machine. When the transfer was completed, the image was loaded from the tar file with the `docker load -i` command (FIGURE 16).

```

6bc5936494e3: Loading layer 4.232MB/4.232MB
b72bc2ebc4c9: Loading layer 2.56kB/2.56kB
e82f94d5e5d0: Loading layer 1.536kB/1.536kB
52bd4dffe5aa: Loading layer 37.41MB/37.41MB
f819a6fbeb0: Loading layer 26.11kB/26.11kB
1fc9374f38d6: Loading layer 2.56kB/2.56kB
703ab2fc3bd2: Loading layer 3.072kB/3.072kB
29521a62d9bf: Loading layer 7.168kB/7.168kB
cd32d454f5df: Loading layer 1.536kB/1.536kB
Loaded image: postgres:9.6-alpine
788ce2310e2f: Loading layer 126.8MB/126.8MB
aa4e47c45116: Loading layer 15.87kB/15.87kB
b3968bc26fbd: Loading layer 14.85kB/14.85kB
c9748fbf541d: Loading layer 5.632kB/5.632kB
2f5b0990636a: Loading layer 3.072kB/3.072kB
0b95783b559f: Loading layer 67.3MB/67.3MB
5fec295091b1: Loading layer 16.11MB/16.11MB
75eb6a9bf1f: Loading layer 26.87MB/26.87MB
Loaded image: huginn:latest

```

FIGURE 15. Loading the compressed Docker image on the target machine.

With the Minikube environment set to communicate with the Docker daemon, the following output should be available with the `docker images` command (FIGURE 17). The two images, `postgres` and `app`, are the ones loaded from the tar file. The rest of the images are native to Kubernetes and are automatically loaded upon installation.

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|--|---------------|--------------|--------------|--------|
| postgres | 9.6-alpine | 7470b931fc2e | 12 days ago | 37.8MB |
| app | 1.5.3-ubuntu | c208c4bf3414 | 2 weeks ago | 231MB |
| gcr.io/google_containers/kubernetes-dashboard-amd64 | v1.8.0 | 55dbc28356f2 | 8 weeks ago | 119MB |
| quay.io/kubernetes-ingress-controller/nginx-ingress-controller | 0.9.0-beta.17 | 5292886cc48f | 2 months ago | 184MB |
| gcr.io/k8s-minikube/storage-provisioner | v1.8.0 | 4689081edb10 | 2 months ago | 80.8MB |

FIGURE 16. Docker images output after setting the environment variable and loading new image.

5.2 Nginx

“NGINX is open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more.” (Nginx glossary 2018, cited 16.01.2018.) It was originally created by Igor Sysoev in 1999 to tackle a problem called *C10K* which derives from “10000 concurrent connections”. The problem concerned the existing web servers that experienced difficulties when handling large numbers of concurrent connections. The project was then open sourced in 2004 and after it grew exponentially, Sysoev founded *Nginx, Inc.* to maintain the development process and offer a commercial product, *Nginx Plus*, for enterprises. Nowadays, Nginx powers more than 50% of sites, such as Netflix, in the World Wide Web (Nginx glossary 2018, cited 16.01.2018).

Nginx was designed to be a multifunction tool. It can be used as a load balancer, reverse proxy, content cache and a web server, which decreases the number of tools used in the organization and

software mismatches. In this project, Nginx is used as a reverse proxy to redirect traffic from the Internet to the exposed Minikube nodes, through the Minikube cluster master. Nginx was chosen for its well documented user manual and its relatively easy-to-configure nature. (Ellingwood 2016, cited 2.1.2018; Nginx glossary 2018, cited 16.01.2018.)

Once Nginx is installed and Minikube is up and running, the Nginx reverse proxy needs to be configured. The configuration file should dictate how the Nginx proxy redirects traffic from the Internet to the Minikube master node cluster. From the Master node, the traffic is redirected to the application inside the cluster. To achieve this, configuration files need to be created in the following location: `/etc/nginx/conf.d`. By default, there are no configuration files, so new files must be made from scratch. The Minikube IP address needs to be found out first, as seen in FIGURE 14.

The configuration of the Nginx reverse proxy is shown in FIGURES 18, 19 and 20, in the configurations made for the website are described. For each figure, the main points are explained to clarify their purpose. The template which was used for this thesis, was made by the development team and it gave good base for the configuration. It was found from the company's internal GitLab repository. The first configuration block, as seen in FIGURE 18, listens to the port 80 for HTTP traffic. If traffic is found, it then redirects it to port 443 for HTTPS.

```
server {
    listen      80;
    listen      [::]:80;
    server_name example.domain.com;
    root        /var/www/example.domain.com;

    location '/.well-known/acme-challenge' {
        root /var/www/example.domain.com;
        index index.html index.htm;
        try_files $uri uri/ =404;
    }

    location / {
        return 302 https://example.domain.com$request_uri;
    }
}
```

FIGURE 17. First server block, listens to HTTP traffic on port 80.

This is the second server block for a TLS enabled server (FIGURE 19). It listens to HTTPS traffic on the default port 443 and has SSL certificates configured. The SSL certificates were requested with CertBot, which is a software application made by LetsEncrypt. CertBot will be explained in the

next chapter. Enforcing the SSL protocol TLSv1.2 is a good practice, because its older versions, 1.1 and 1.0, are insecure and pose a security risk for the server.

```
# Settings for a TLS enabled server.
server {
    listen      443 ssl ;
    listen     [::]:443 ssl http2 ;
    #include snippets/ssl-example.com.conf;
    #include snippets/ssl-params.conf;

    server_name     example.domain.com;
    root            /var/www/example.domain.com;
    ssl_certificate /etc/letsencrypt/live/example.domain.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/example.domain.com/privkey.pem;
    ssl_protocols  TLSv1.2;
}
```

FIGURE 18. Second server block for HTTPS traffic.

Next is the location block (FIGURE 20). Here the redirect address is configured to redirect traffic to the Minikube master. The basic authorization (APPENDIX 7) and specific access restrictions (IP-blacklisting) and configurations go here.

```
location / {
# Setting the Minikube address as the proxy pass.
    proxy_pass http://192.168.99.100;
# Redirect traffic here
    proxy_redirect http://192.168.99.100/ $scheme://$host/;
    proxy_set_header X-Real-IP $remote_addr; # http://wiki.nginx.org/HttpProxyModule
    proxy_set_header Host $host;
    proxy_http_version 1.1;
# WebSocket proxying - from http://nginx.org/en/docs/http/websocket.html
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection $connection_upgrade;
# Setting up basic authorization
    auth_basic "This webdomain is private. Misuse will be punished";
    auth_basic_user_file /etc/nginx/.htpasswd;
# Setting access restrictions.
    allow 12.345.678.90;
    deny all;
}
```

FIGURE 19. Nginx configuration files, location part. Private information has been changed.

5.3 Encrypting the traffic

For encryption a software application made by Let's Encrypt was used. Let's Encrypt is a free to use, automated certificate authority that uses the Automatic Certificate Management Environment protocol to provide free TLS/SSL certificates to any compatible client (Boucheron 2017, cited 11.01.2018). The purpose of the certificates is to encrypt the traffic between the webserver and its users.

There are several clients with which the certificate can be allocated. In this thesis, a software application *CertBot* was used. Citing the CertBot introduction page: “Certbot is part of EFF’s effort to encrypt the entire Internet. Secure communication over the Web relies on HTTPS, which requires the use of a digital certificate that lets browsers verify the identity of web servers (e.g., is that really google.com?).” (Certbot Intro 2016, cited 11.01.2018.)

Certbot is developed by the Electronic Frontier Foundation, and in addition to just verifying domain ownerships and fetching certificates, it can automatically configure TLS/SSL for the web server (Boucheron 2017, cited 11.01.2018). To this day roughly 50% of websites in the Internet use HTTPS. This is a step towards a more secure Internet. HTTPS encrypts the traffic from the webserver to the client, making it harder for an attacker to hijack, for example, credit card information and passwords.

5.4 Creating a certificate

Configuring a certificate with Certbot is done with `sudo certbot`, or `sudo certbot certonly` in case the Nginx configuration file needs to be manually configured. A dry run was simulated to show the certificate creation process (FIGURE 21).

```
miika.moilanen@bitidemo:~$ sudo certbot certonly --dry-run
[sudo] password for miika.moilanen:
Saving debug log to /var/log/letsencrypt/letsencrypt.log

How would you like to authenticate with the ACME CA?
-----
1: Spin up a temporary webserver (standalone)
2: Place files in webroot directory (webroot)
-----
Select the appropriate number [1-2] then [enter] (press 'c' to cancel): 2
Plugins selected: Authenticator webroot, Installer None
Please enter in your domain name(s) (comma and/or space separated) (Enter 'c'
to cancel): .com
Cert not due for renewal, but simulating renewal for dry run
Renewing an existing certificate
Performing the following challenges:
http-01 challenge for .com

Select the webroot for .com:
-----
1: Enter a new webroot
-----
Press 1 [enter] to confirm the selection (press 'c' to cancel): 1
Input the webroot for .com: (Enter 'c' to cancel): /var/www/ .com
waiting for verification...
Cleaning up challenges

IMPORTANT NOTES:
- The dry run was successful.
```

FIGURE 20. Simulated CertBot certificate creation. Domain names blacked out due to privacy reasons.

When the certificate creation is complete, the client tells where the certificates are located, which by default is:

```
/etc/letsencrypt/live/domain_name/fullchain.pem
/etc/letsencrypt/live/domain_name/privkey.pem.
```

Now that the certificates are configured and ready to use. They are then added to the Nginx configuration file which was shown in the previous figure. (FIGURE 21). Whenever making changes to the configuration files, the Nginx service must be restarted to apply the settings.

5.5 Deployment

Now that the configurations are ready, certificates set, the application can now be deployed to Minikube. Like docker-compose, Kubernetes has its own version called *Kompose*. During the build process, it first translates the *docker-compose.yml* file into several *YAML* files which are readable by Kubernetes. From the files it can then read the instructions for the deployment. When using *kompose up*, the *YAML* files do not persist and are deleted after the building is done. (Kubernetes kompose 2018, cited 14.1.2018.)

If different kind of resources are needed, the docker-compose file can be first converted into Kubernetes readable syntax. This is done via *kompose convert*. From one file, depending how it is configured, there will be several *YAML* files with each their own instructions for deployment. (Kubernetes kompose 2018, cited 14.1.2018.) For instance, for one docker-compose file which deploys one application you might get two files for *deployment* and *service*. The application is then deployed using the *kompose up* command (FIGURE 22). A specific file can be determined by using the *-f* flag, for instance, *kompose up -f prod-application.yaml*.

```
INFO We are going to create Kubernetes Deployments, Services and PersistentVolumeClaims for your Dockerized application. If
you need different kind of resources, use the 'kompose convert' and 'kubectl create -f' commands instead.
INFO Deploying application in "prod" namespace
INFO Successfully created Service: db
INFO Successfully created Service: db2
INFO Successfully created Service: db3
INFO Successfully created Service: prod-app
INFO Successfully created Deployment: db
INFO Successfully created PersistentVolumeClaim: db-claim0 of size 100Mi. If your cluster has dynamic storage provisioning,
you don't have to do anything. Otherwise you have to create PersistentVolume to make PVC work
INFO Successfully created Deployment: db2
INFO Successfully created PersistentVolumeClaim: db2-claim0 of size 100Mi. If your cluster has dynamic storage provisioning
, you don't have to do anything. Otherwise you have to create PersistentVolume to make PVC work
INFO Successfully created Deployment: db3
INFO Successfully created PersistentVolumeClaim: db3-claim0 of size 100Mi. If your cluster has dynamic storage provisioning
, you don't have to do anything. Otherwise you have to create PersistentVolume to make PVC work
INFO Successfully created Deployment: prod-app
Your application has been deployed to Kubernetes. You can run 'kubectl get deployment,svc,pods,pvc' for details.
```

FIGURE 21. Output of *kompose up*.

The deployment is ready, in this case we deployed an application in to the “prod” namespace. The deployment consists of three databases, which are linked to the application. To view the details of the deployment, the following command can be run: `kubectl get deployment,svc,pods,pvc,pv`. (FIGURE 23).

```

NAME          DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
deploy/db     1        1        1            1           2m
deploy/db2    1        1        1            1           2m
deploy/db3    1        1        1            1           2m
deploy/prod-app 1        1        1            1           2m

NAME          TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
svc/db        ClusterIP     None             <none>            55555/TCP        2m
svc/db2       ClusterIP     None             <none>            55555/TCP        2m
svc/db3       ClusterIP     None             <none>            55555/TCP        2m
svc/prod-app  NodePort     10.111.222.93   <none>            4000:31400/TCP  2m

NAME          READY    STATUS    RESTARTS  AGE
po/db-7f76677b-vbdnx  1/1     Running  0          2m
po/db2-65cbf77698-jbtz2  1/1     Running  0          2m
po/db3-55f5d598d-mb1xc  1/1     Running  0          2m
po/prod-app-68746f9d58-d5g4q  1/1     Running  0          2m

NAME          STATUS    VOLUME                                     CAPACITY  ACCESS MODES  STORAGECLASS  AGE
pvc/db-claim0  Bound    pvc-098d20b5-0006-11e8-82ab-0800276b600d  100Mi     RWO            standard      2m
pvc/db2-claim0  Bound    pvc-0993f349-0006-11e8-82ab-0800276b600d  100Mi     RWO            standard      2m
pvc/db3-claim0  Bound    pvc-0997ca4e-0006-11e8-82ab-0800276b600d  100Mi     RWO            standard      2m

NAME          CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS    CLAIM                STORAGECLASS  REASON  AGE
pv/pvc-098d20b5-0006-11e8-82ab-0800276b600d  100Mi     RWO            Delete          Bound    prod/db-claim0      standard    2m
pv/pvc-0993f349-0006-11e8-82ab-0800276b600d  100Mi     RWO            Delete          Bound    prod/db2-claim0     standard    2m
pv/pvc-0997ca4e-0006-11e8-82ab-0800276b600d  100Mi     RWO            Delete          Bound    prod/db3-claim0     standard    2m

```

FIGURE 22. Output of deployed application, Deployments, services, pods, PersistentVolumeClaims (PVC) and PersistentVolumes (PV).

To access the application, *prod-app*, in this case Kubernetes uses the exposed port 31400. The Nginx reverse proxy that was configured, sends the incoming traffic to the Kubernetes master. From the master node, the traffic is sent to the application, through the Ingress controller. The database instances communicate with each other inside the cluster, so there is no need for expose them to the Internet. The application is exposed as a service via *NodePort*, which is determined in the *docker-compose.yml* file with a Kubernetes label (APPENDIX 7). The service persists between deployment reboots and has a static port assigned to.

When the Nginx- reverse proxy is configured to redirect traffic to the Minikube cluster master, it will automatically know where to redirect the traffic if an ingress controller is enabled and has a properly configured resource. For different versions of applications (dev, staging, prod) namespaces and an ingress resource for each namespace should be created to point to right place (FIGURES 11 & 12). With the Nginx proxy configured, the application exposed as *NodePort* and the Minikube ingress is set to point at the *prod-app* service in production namespace, the application is now accessible in the assigned web domain. (FIGURE 24).

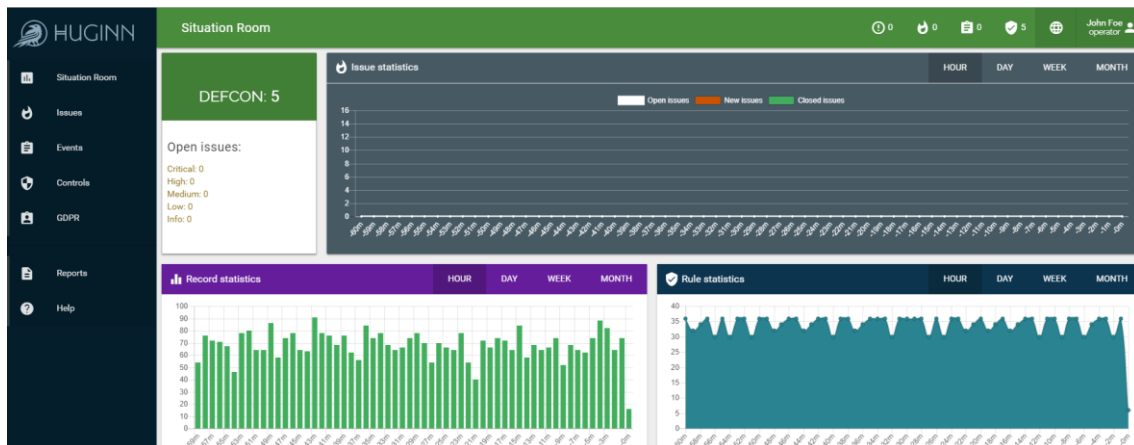


FIGURE 23. The dashboard of the application showing different statistics.

The front page opens to the situation room of the Huginn application and it is the heart of the application, from where the current situation is displayed. There are three different areas that are being monitored. The “Issue statistics” field declares if there are any errors in the data reported by the rules created. Issues generated can be monitored hourly, daily, weekly or monthly. The “Record statistics” monitor all the generated rows in databases, deleted, changed and new rows. Rule statistics display how many rule checks per run have been checked. The situation room shows the recent issues created and tells which ones are the most critical.

The purpose of the application is to detect information anomalies in the targeted system. If such anomalies are detected, they are then reported in the form of "Issues". This enables the observer to react quickly against possible threats and mitigate already happened misuses quickly. For instance, the billing process can be monitored. Huginn pays attention to the IBAN numbers which are used to complete the payment. If a mismatch is found, Huginn creates an issue. This prevents false billing addresses and increases the chances of preventing the accident if reacted to quickly.

6 CONCLUSION

The goal of the project was to create a Docker image of the Huginn application and deploy it on Kubernetes. The goal was achieved, and the test version of the application was available for the allotted time. The next step would be to use the existing files created during the process for automation and deployment to a real production environment.

The whole project was a blast to do. Before starting I had never heard of Docker and Kubernetes and it took quite a lot of research and trying out new things to finally get to the wanted result. The whole project took around two months to complete. Some things could have been done better in this thesis, but I am happy with the result.

The hardest part of this thesis was tackling a whole new subject. With next to none knowledge of Docker and Kubernetes, it required a lot of reading through the manuals and trial of error. When nearing towards the end of the thesis, keeping up with references became harder and possible typing errors also grew more frequent and required more attention.

What was done well in my opinion, was the configuration of the Dockerfile and docker-compose.yml file. When moving on to the Kompose tool, there were no problems when deploying the application. Also, the Nginx proxy configuration was well done, for a first timer. The created configuration files can be used afterwards giving direction to similar projects.

What could have been done better, was for example using a bigger cluster, such as the minimal Kubernetes cluster, kubernetes-core. Since Minikube is meant for local development, it was not the best choice for the deployment. But since they generally use the same commands, only the installation process is different. Also, instead of using the docker-compose.yml file to do the deployment, Kubernetes specific YAML files could have been made, but for this thesis it was not required.

When starting the next Docker and Kubernetes related project, it will be much easier to do and instead of Googling the answers for a few weeks I will know where to begin and what needs to be done to reach the goal. I feel that this project broadened my knowledge in the areas of

containerization, virtualization, domains and networking overall and has set me on the path to be a DevOps professional one day.

REFERENCES

Abbassi, P. (2016) Understanding Basic Kubernetes Concepts III – Services give you abstraction. Cited 24.01.2018, <https://blog.giantswarm.io/basic-kubernetes-concepts-iii-services-give-abstraction>

Aliakhtar. (2016). Insufficient memory / CPU? Cited 11.01.2018, <https://github.com/kubernetes/minikube/issues/567>.

Boucheron, Brian. (2017). An Introduction to Let's Encrypt. Cited 11.01.2018, <https://www.digitalocean.com/community/tutorials/an-introduction-to-let-s-encrypt>.

Cane, B (2017). Leveraging the Dockerignore file to create smaller images. Cited 18.01.2018, <https://blog.codeship.com/leveraging-the-dockerignore-file-to-create-smaller-images/>

Certbot Intro. (2016) Introduction. Cited 11.01.2018, <https://certbot.eff.org/docs/intro.html>

Coleman, M. (2016). Containers and VMs together. Cited 18.01.2018, <https://blog.docker.com/2016/04/containers-and-vms-together/>

DeHamer, Brian. (2015). Dockerfile: ENTRYPOINT vs CMD. Cited 10.02.2018, <https://www.ctl.io/developers/blog/post/dockerfile-entrypoint-vs-cmd/>

Dockerfile (2017). Dockerfile reference. Cited 25.12 2017, <https://docs.docker.com/engine/reference/builder/>

Dockerfile guide (2017). Best practices for writing Dockerfiles. Cited 25.12.2017, https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/

Docker Company. (2018). Docker Milestones. Cited 14.1.2018, <https://www.docker.com/company>

Docker Compose. (2017). Docker Compose. Cited 11.01.2018, <https://docs.docker.com/compose/>

Docker container. (2018). What container? Cited 13.1.2018, <https://www.docker.com/what-container>

Docker eBook. (2016). Docker for the Virtualization Admin eBook. Docker. 3-5. Cited 28.12.2017, https://goto.docker.com/rs/929-FJL-178/images/Docker-for-Virtualization-Admin-eBook.pdf?mkt_tok=eyJpIjoiTIRaa09ERXhNelE1TVRabClInQiOiJNdHVcl1N5a3hqdlp1ekUzR3hhQ0xtUW5UUElzYVd5aXR4XC9BSEZpZVVJT05KZndGWVY4ZURpTzJ0bmdsK2U2VkFBaUxrNTBFMmRVN1g1WmNkS0w3RmhvMHAwa3RhcTNWY3UxWXNxRXJLQTF3PSJ9

Docker storage. (2018) Select a storage driver. Cited 17.01.2018, <https://docs.docker.com/engine/userguide/storagedriver/selectadriver/>

Docker VR. (2018). Finnish railways unifies app modernization projects with Docker Enterprise edition. Cited 9.2.2018, <https://www.docker.com/customers/finnish-railways-unifies-app-modernization-projects-docker-enterprise-edition>

Ellingwood, Justin. (2016). How to Install Nginx on Ubuntu 16.04. Cited 2.1.2018, <https://www.digitalocean.com/community/tutorials/how-to-install-nginx-on-ubuntu-16-04>

Hong, K. (2017). Running Kubernetes locally via Minikube. Cited 11.01.2018, <http://www.bogotobogo.com/DevOps/DevOps-Kubernetes-1-Running-Kubernetes-Locally-via-Minikube.php>

Janetakakis, N. (2017). Docker Tip #2: The difference between COPY and ADD in a Dockerfile. Cited 18.01.2018, <https://nickjanetakakis.com/blog/docker-tip-2-the-difference-between-copy-and-add-in-a-dockerfile>

Janmyr, A. (2015). A not very short introduction to docker. Cited 4.1.2018, <https://blog.jayway.com/2015/03/21/a-not-very-short-introduction-to-docker/>

Kubernetes. (2018) What is Kubernetes? Cited 14.1.2018, <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>

Kubernetes cluster. (2018) Cluster Intro. Cited 14.1.2018,
<https://kubernetes.io/docs/tutorials/kubernetes-basics/cluster-intro/>

Kubernetes ingress. (2018). Ingress. Cited 10.2.2018,
<https://kubernetes.io/docs/concepts/services-networking/ingress/>

Kubernetes kompose. (2018). Translate a Docker Compose File to Kubernetes Resources. Cited 14.1.2018, <https://kubernetes.io/docs/tools/kompose/user-guide/#kompose-convert>

Kubernetes namespace. (2018). Namespaces. Cited 16.01.2018,
<https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/>

Kubernetes pod. (2018). Pods. Cited 14.01.2018,
<https://kubernetes.io/docs/concepts/workloads/pods/pod/>

Kubernetes quickstart. (2018). Getting started guide: Running Kubernetes Locally via Minikube. Cited 11.01.2018, <https://kubernetes.io/docs/getting-started-guides/minikube/>

Kubernetes service. (2018). Services. Cited 14.01.2018,
<https://kubernetes.io/docs/concepts/services-networking/service/>

Kubernetes storage. (2018). Dynamic Provisioning and Storage classes in Kubernetes. Cited 23.01.2018, <http://blog.kubernetes.io/2017/03/dynamic-provisioning-and-storage-classes-kubernetes.html>

Kubernetes volume. (2018). Persistent Volumes. Cited 22.01.2018,
<https://kubernetes.io/docs/concepts/storage/persistent-volumes/>

Matthias, Karl. & Kane, Sean P. (2015). Docker Up & Running. United States of America: O'Reilly Media, Inc, 1-44.

McKendrick, Russ & Gallagher, Scott. (2017) Mastering Docker. Second Edition. Birmingham, UK: Packt Publishing Ltd, 36.

Meegan, J. (2016). A practical guide to platform as a service: What is PaaS? Cited 12.2.2018, <https://www.ibm.com/blogs/cloud-computing/2016/08/practical-guide-paas/>

Nginx glossary. (2018) What is Nginx? Cited 16.01.2018, <https://www.nginx.com/resources/glossary/nginx/>

Oranagwa, O. (2017). Setting up Ingress on Minikube. Cited 16.01.2018, <https://medium.com/@Oskarr3/setting-up-ingress-on-minikube-6ae825e98f82>

Podviaznikov, Anton. (2017). The Versatility of Docker Compose. Cited 11.01.2018, <https://runnable.com/blog/the-versatility-of-docker-compose>

Rahul, K. (2016). How to install Oracle VirtualBox 5.2 on Ubuntu. Cited 25.12.2017, <https://tecadmin.net/install-oracle-virtualbox-on-ubuntu/>

Rajdep. (2014). Virtualization vs Containerization to support PaaS. Cited 13.01.2018, <https://www.slideshare.net/rajdeep/conference-presentationv3>

Rubens, Paul. (2017). What are containers and why do you need them? Cited 26.12.2017, <https://www.cio.com/article/2924995/software/what-are-containers-and-why-do-you-need-them.html#toc-1>

Sayfan, Gigi. (2017). Mastering Kubernetes. Packt Publishing, 2017. Cited 25.12.2017, http://proquestcombo.safaribooksonline.com.ezp.oamk.fi:2048/book/software-engineering-and-development/9781786461001/1dot-understanding-kubernetes-architecture/ch01s02_html

Stone, L. (2016) Bringing Pokémon GO to life on Google Cloud. Cited 9.2.2018, <https://cloudplatform.googleblog.com/2016/09/bringing-Pokemon-GO-to-life-on-Google-Cloud.html>

Ubuntu. (2015). BasicChroot. Cited 12.2.20128, <https://help.ubuntu.com/community/BasicChroot>

Upguard. (2017). Docker vs LXC. Cited 13.01.2018, <https://www.upguard.com/articles/docker-vs-lxc>

Wang, C (2017). What is Docker? Linux containers explained. Cited 18.01.2018,
<https://www.infoworld.com/article/3204171/linux/what-is-docker-linux-containers-explained.html>

INSTALLING DOCKER

APPENDIX 1

Install Docker dependencies (Ubuntu 16.04)

```
sudo apt-get install \  
    apt-transport-https \  
    ca-certificates \  
    curl \  
    software-properties-common
```

Add the GNU Privacy Guard- key (GPG) for Docker

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

Compare fingerprints:

```
sudo apt-key fingerprint 0EBFCD88
```

Add Docker repository:

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu  
$(lsb_release -cs) stable"
```

Install Docker:

```
sudo apt-get update  
sudo apt-get install docker-ce
```

Download Minikube:

```
curl -Lo minikube https://storage.googleapis.com/minikube/releases/v0.24.1/minikube-linux-
```

Change permissions:

```
chmod +x minikube
```

Moving the file to a different path. It can be changed in case the user wants to.

```
sudo mv minikube /usr/local/bin/
```

INSTALLING KUBECTL

APPENDIX 3

Kubectl installation:

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s  
https://storage.googleapis.com/kubernetes-  
release/release/stable.txt)/bin/linux/amd64/kubectl
```

Install specific version (1.9.0):

```
curl -LO  
https://storage.googleapis.com/kubernetesrelease/release/v1.9.0/bin/linux/amd64/kubectl
```

Permissions for kubectl:

```
chmod +x ./kubectl
```

Relocating kubectl binary:

```
sudo mv ./kubectl /usr/local/bin/kubectl
```

Autocomplete commands:

```
echo "source <(kubectl completion bash)" >> ~/.bashrc
```

Download the binary via Curl:

```
curl -L https://github.com/kubernetes/kompose/releases/download/v1.7.0/kompose-linux-  
amd64 -o kompose
```

Change permissions so it can be executed:

```
chmod +x kompose
```

Move the file to a different location, can be changed to a different one:

```
sudo mv ./kompose /usr/local/bin/kompose
```


INSTALLING NGINX

APPENDIX 5

Installing Nginx for Ubuntu 16.04:

```
sudo apt-get update
```

```
sudo apt-get install nginx
```

UFW Firewall configuration:

```
sudo ufw allow 'Nginx HTTPS'
```

```
sudo ufw allow 'Nginx HTTP'
```

```
sudo ufw allow 'OpenSSH'
```

Enabling firewall:

```
sudo ufw enable
```

```
sudo service ufw start
```

INSTALLING BASIC AUTH

APPENDIX 6

```
sudo apt-get install apache2-utils  
sudo htpasswd -c /etc/nginx/.htpasswd jerry
```

Add filepath to reverse proxy configuration:

```
auth_basic_user_file /etc/nginx/.htpasswd;
```

```

version: '3.0'
services:
  app:
    restart: always
    image: app:1.5.3-ubuntu
    networks:
      app_net:
    command: ./bin/application foreground
    ports:
      - "4000:4000"
    environment:
      - NAME=application
      - PORT=4000
      - HOST_NAME=prod.example.domain.com
      - COOKIE=secret
      - CHECK_ORIGIN=true
      - LANG=en_US.UTF-8
      - LANGUAGE=en_US:en
      - LC_ALL=en_US.UTF-8
      - MAX_SESSION_AGE=1800
      - RULE_POOL=10
      - ENABLE_DLP=false
      - APPLICATION_LOGIN_ENABLED=true
      - REPLACE_OS_VARS=true
      - DB1_URL=postgres://postgres:postgres@db1/db1
      - DB2_POOL=10
      - DB2_URL=postgres://postgres:postgres@db2/db2
      - DB2_POOL=10
      - DB3_URL=postgres://postgres:postgres@db3/db3
      - DB3_POOL=10
    labels:
      kompose.service.type: nodeport
  db1:
    restart: always
    image: postgres:9.6-alpine
    networks:
      app_net:
    environment:
      - DEBUG=false
      - POSTGRES_DB=db1
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
  db2:
    restart: always
    image: postgres:9.6-alpine
    networks:
      app_net:
    environment:
      - DEBUG=false
      - POSTGRES_DB=db2
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
  db3:
    restart: always
    image: postgres:9.6-alpine
    networks:
      app_net:
    environment:
      - DEBUG=false
      - POSTGRES_DB=db3
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
networks:
  app_net:
    driver: bridge

```