BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

MMO PLUGIN FOR DEPLOYMENT OF MICROSERVICES INTO THE CLUSTER

MODUL MMO PRO NASAZENÍ MIKROSLUŽEB DO CLUSTERU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTOR **JAKUB KULICH**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. ADAM HEROUT, PhD.**
VEDOUCÍ PRÁCE

BRNO 2018

Zadání bakalářské práce/20508/2017/xkulic03

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Kulich Jakub**
Obor: Informační technologie
Téma: **Modul MMO pro nasazení mikroslužeb do clusteru**
MMO Plugin for Deployment of Microservices into the Cluster
Kategorie: Web

Pokyny:

1. Seznamte se s problematikou mikroslužeb a jejich orchestrace v clusterech.
2. Prozkoumejte a popište dostupné orchestrační nástroje.
3. Navrhněte zásuvný modul pro usnadnění práce s mikroslužbami; zaměřte se na tyto funkce: nasazení služeb do clusteru, spojení se systémem MMO, správa závislostí mezi službami.
4. Implementujte prototyp zásuvného modulu.
5. Ověřte a zhodnoťte funkčnost zásuvného modulu na vhodných příkladech/prototypech. Iterativně vylepšujte vytvořené řešení.
6. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte plakátek a krátké video pro prezentování projektu.

Literatura:

- dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3, značné rozpracování bodů 4 a 5.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Vedoucí: **Herout Adam, prof. Ing., Ph.D.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
L.S. 612 66 Brno, Božetěchova 2

[illegible]

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstract

Orchestration of applications with microservice architecture is difficult. Available tools do not allow deployment of an application to the user without deep knowledge of target platform. Another problem is that many mistakes are done when these tools are used. MMO – Monorepo Microservice Orchestrator is a tool that makes the development of applications with microservice architecture easier. MMO can be extended with parts that can help the user to avoid problems mentioned above. One part is used for generation of the configurations for deployment of the application to Kubernetes and second part is used for deployment of the application the Kubernetes cluster. A result of using MMO extension is an ability of beginner users to create deployment configurations, faster creation of deployment configurations by advanced users and decreased number of mistakes done when the application is deployed manually by Kubernetes users.

Abstrakt

Aplikácie s architektúrou typu „mikroslužby“ je náročné orchestrovať. Dostupné nástroje neumožňujú užívateľom nasadenie aplikácie bez veľkej znalosti cieľovej platformy. Ďalším problémom je, že užívatelia pri nepozornosti robia chyby pri používaní týchto nástrojov. Vývoj aplikácií s architektúrou typu „mikroslužby“ zjednodušuje open-source nástroj MMO – Monorepo Microservice Orchestrator. Rozšírením tohto nástroja o určité časti nám umožňuje vyhnúť sa spomínaným problémom. Jedna časť slúži na generovanie konfigurácií pre nasadenie aplikácie do Kubernetes clustera. Druhá časť umožňuje samotné nasadenie aplikácie do Kubernetes clustera. Výsledkom používania rozšírenia nástroja MMO je uľahčenie vytvárania konfiguračných súborov u začiatočníckych používateľov nástroja Kubernetes, urýchlenie vytvorenia konfiguračných súborov u pokročilých užívateľov nástroja Kubernetes a zníženie počtu chýb ktoré užívatelia robia pri ručnom nasadení aplikácie.

Keywords

deployment, microservice, architecture, monorepo,kubernetes, cluster

Klíčová slova

nasadenie, mikroslužby, monorepo, kubernetes

Reference

KULICH, Jakub. *MMO Plugin for Deployment of Microservices into the cluster*. Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Adam Herout, PhD.

Rozšířený abstrakt

Aplikácie s architektúrou typu „mikroslužby“ je náročné orchestrovať. Vzniklo množstvo nástrojov na orchestráciu jednotlivých služieb v clusteri, je však problém vybrať spomedzi nich ten správny pre konkrétne použitie. Každý nástroj má určité výhody a určité nevýhody. Niektoré sú vhodné pre veľké projekty, iné zase pre projekty menšej veľkosti. Táto práca sa venuje porovnaniu týchto nástrojov a výberu nástroja, ktorý bude najviac vyhovovať väčšine používateľov.

Konkrétny nástroj vybratý pre túto prácu je open-source nástroj Kubernetes. Dostupné nástroje pre správu aplikácií bežiacich v Kubernetes neumožňujú užívateľom nasadenie aplikácie bez veľkej znalosti cieľovej platformy. Ďalším problémom je, že užívatelia pri nepozornosti robia chyby pri používaní týchto nástrojov. Tieto chyby môžu viesť k časovým a finančným stratám v závislosti od konkrétnej chyby. V rámci tejto práce bol urobený prieskum medzi užívateľmi nástroja Kubernetes pre lepšie pochopenie toho ako užívatelia nástroj používajú.

Vývoj aplikácií s architektúrou typu „mikroslužby“ zjednodušuje open-source nástroj MMO – Monorepo Microservice Orchestration. Nástroj MMO momentálne dokáže generovať projekty a ich mikroslužby podľa šablón, tiež podporuje zásuvné moduly, pomocou ktorých dokáže užívateľ rozšíriť základnú funkčnosť nástroja. Nástroju chýbajú funkcie, ktoré by zjednodušili nasadenie aplikácie do Kubernetes. Rozšírením tohto nástroja o určitú funkcionalitu, umožňuje užívateľovi vyhnúť sa vyššie spomínaným problémom. Tieto funkcie budú slúžiť k tomu aby pomohli vývojárom s vytvorením konfigurácií pre nasadenie aplikácií do Kubernetes a tiež aby pomohli so samotným nasadením aplikácie do Kubernetes clustera. Rozšírenie nástroja je vo forme užívateľského rozhrania, ktoré je pre toto použitie prehľadnejšie ako konzolová aplikácia.

Výsledkom tejto práce je rozšírenie nástroja MMO, ktoré rieši vyššie spomínané problémy. Rozšírenie umožňuje generovanie konfiguračných súborov pre nasadenie aplikácií do Kubernetes clustera. To umožňuje začiatočníckym používateľom nástroja Kubernetes vytvoriť konfiguračné nástroje aj bez hlbšej znalosti nástroja Kubernetes. U pokročilých užívateľov nástroja Kubernetes prichádza k urýchleniu vytvorenia konfiguračných súborov. Ďalšia funkcia ktorú prináša rozšírenie nástroja MMO vytvorené vrámci tejto práce je funkcia určená pre nasadenie aplikácie. Aplikáciu je možné nasadiť dvoma spôsobmi. Prvý spôsob je nasadenie pomocou GitHub Deployments API, kde nasadenie prebieha v rámci automatizovaného procesu. Druhý spôsob je priame nasadenie do Kubernetes clustera. Táto funkcia znižuje počet chýb, ktoré užívatelia robia pri ručnom nasadení aplikácie. Znížením počtu chýb môže ušetriť vývojár svoj čas a tým aj financie.

Práca sa tiež zaoberá pokusom, kde bol vytvorený prototyp platformy určenej na nasadenie aplikácií. Samostatná platforma by si našla svojich užívateľov, ale až v prípade, kedy by integrovala väčšie množstvo funkcií. Rozsah práce na takejto veľkej platforme však nie je realizovateľný v rámci bakalárskej práce.

MMO Plugin for Deployment of Microservices into the cluster

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of prof. Ing. Adam Herout, Phd. The supplementary information was provided by Bc. Peter Malina. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

. . . .
Jakub Kulich
May 14, 2018

Acknowledgements

First of all, I have to thank my supervisor Adam Herout. This thesis would have never been accomplished without his assistance and ability to motivate me. I would also like to thank Peter Malina for his suggestions.

Contents

Chapter 1

Introduction

The complexity of developed applications had brought microservice architecture pattern to use. A disadvantage of the microservice architecture is the difficulty to orchestrate applications built with this pattern in the cluster. The goal of this thesis is to extend open-source tool *MMO – Monorepo Microservice Orchestration* so it helps developers to deploy application to the cluster. *MMO* is an open-source tool that helps to develop projects with microservice architecture. The second object of thesis is to compare available tools for orchestrating applications in cluster and select one that *MMO* has to be compatible with.

The second chapter breaks down the problem of microservice architecture, how applications with this architecture are distributed to target platforms and how they are orchestrated. The third chapter talks about Kubernetes and its concepts to understand how applications should be deployed. The third chapter also presents the results of the survey about using Kubernetes to understand Kubernetes users better. This is a prerequisite for finding the best design of the *MMO* extension. The fourth chapter shows existing solutions, their advantages and disadvantages. The fifth and sixth chapters talk about design of the extension and implementation of this design. The seventh chapter is demonstrating the benefits of using *MMO* extension for application deployment as a result of the multiple experiments. The last chapter summarizes the important findings that result from this thesis.

Chapter 2

Deploying Applications with Microservice Architecture to the Cluster

Software development is usually done in accordance with iterative models. The classic development cycle of the iterative model has these parts: planning, development, testing, evaluation. Figure 2.1 shows development cycle from the point of the view of the DevOps engineer. Some parts of the development cycle such as planning, testing, etc are omitted from the diagram in the Figure because they are not important for this thesis. Each part of the development cycle in the figure has one section dedicated in this chapter.

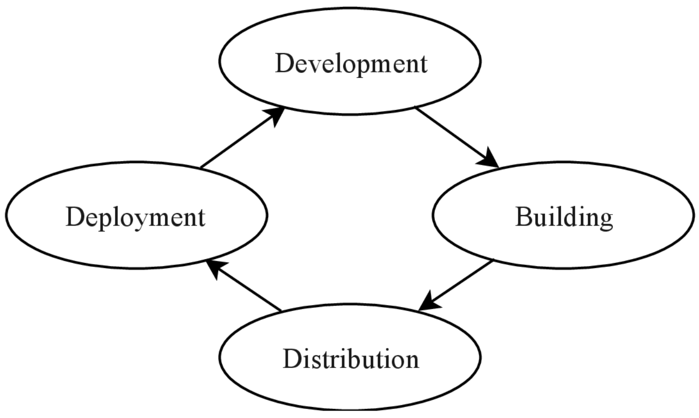

Figure 2.1: Development cycle of iterative development model from the DevOps engineer point of view. When new version of application is developed, it has to be built, tested, distributed to the environment where it will run and in the last step, it has to be launched.

2.1 Development of Microservice Applications

Microservice architecture is an architecture used for building complex applications. The biggest problem in complex applications is their size and the number of users they have to serve. One of the new architecture patterns is called microservice architecture. This architecture solves problems with managing development teams and scaling application but brings other downsides.

2.1.1 Microservice Architecture

Microservices are a type of software architecture used in the server applications [8]. Opposite architecture pattern is monolithic architecture. The principle of the microservice architecture is to divide application to the logical parts where each part acts as a standalone service responsible for its tasks. Decomposition of the monolith to microservices can be seen in Figure 2.2. For example, an e-commerce application with monolithic architecture is one server that does all the work. Using the microservice architecture, application can be broken to multiple services that do small tasks – one can manage users with authentication and authorization, another one can manage products, another service will manage finance services (payments, invoices), etc.

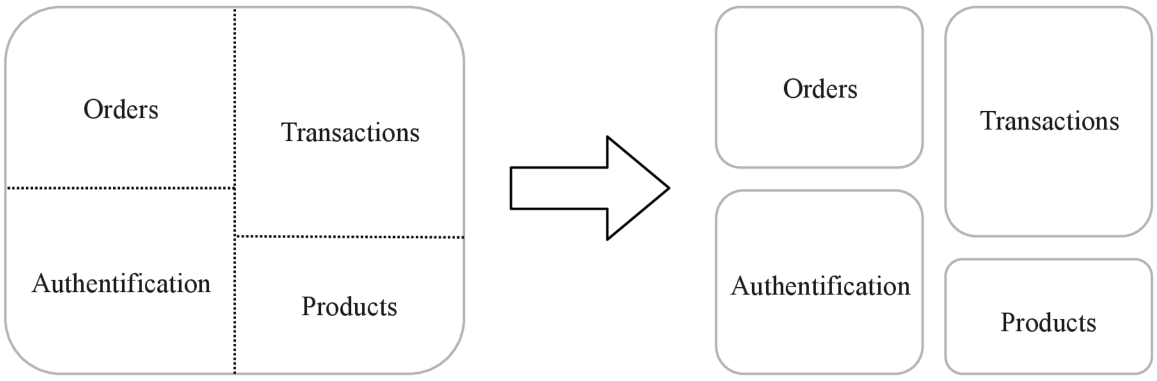

Figure 2.2: Decomposition of the monolithic application to application with microservice architecture

A microservice architecture offers better control over the source code – each service is in its own repository or in one part of a shared repository [8]. This can be also reached in monolithic application but it is easy to break this rule. This plays well with team management, because microservice is working as a standalone unit. One team can develop one service and multiple services can be divided into multiple teams. This makes development faster and more agile. It is also bringing one downside and it is that services must have well defined communication interfaces to communicate between themselves. The most usual ways of the communication between the services is using HTTP protocol or using RPCs (Remote Procedure Calls) [7]. A favorite RPC framework between developers is *gRPC – Google Remote Procedure Call*[1], which supports more than 10 of the most favorite programming languages. It may play a role in the next advantage of the microservice architecture – a microservice architecture is a polyglot. That means that each of the services can be programmed in a different language and can use different technology stack that is most suitable for the use case of the service. Another advantage of microservices is their scalability. Just services that are resource intensive, can be scaled. In the monolithic application, whole application would have to be scaled which is not that easy as small single-purpose

[1] https://grpc.io

service. A example of scaling the application with the microservice architecture is shown in Figure 2.3.

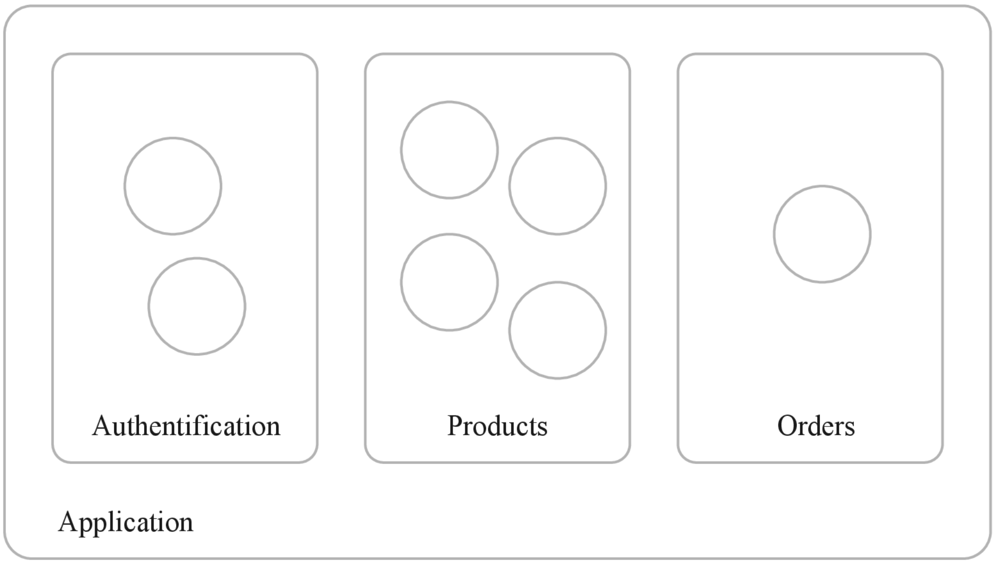

Figure 2.3: Scalability of the services – each service can have different number of replicas

Microservice architecture offers a better tolerance of errors in comparison to monolithic architecture [8]. Critical error in one service does not affect the runtime of the whole application.

As mentioned before, microservice architecture brings multiple downsides:

- much more difficult orchestration of the services,
- a more difficult setup of the continuous integration,
- larger network overhead,
- more complex monitoring of the services.

The biggest drawback of the microservice architecture is the complexity of setting up continuous integration [8]. Continuous Integration (CI) pipeline consists of multiple steps like building, testing [4]. All advantages mentioned above are turning into disadvantages when it comes to setting up the continuous integration. It is a lot harder to set up CI pipeline for the project with microservice architecture which can be written in the multiple programming languages.

Another big disadvantage is the deployment of the application to a server. This also follows the previous problem, because deployment of the application is usually done from the CI system [8]. Monolithic application deployment is easy because everything that has to be run is one application binary. Multiple applications have to be deployed and linked in the network in the application with microservice architecture. The microservice architecture also brings overhead to the network layer of the application. All communication between the services is done on the network layer. They must know where is another service with which they want to communicate so there always have to be service that is responsible for service discovery.

A minor problem of the microservice architecture is service monitoring. In monolithic architecture is only one application that will be monitored. There is need for the service that is dedicated to collecting logs in microservice architecture.

There may be questions how small a microservice should be or how many microservices should the application have. This can differ from application to application but the common approach is that a single microservice should be refactored or replaced by the a version in a short time [8].

2.1.2 Managing Source Code of the Microservice Applications

The microservice architecture has multiple approaches to source code control. One of the approaches is that each service has its own dedicated repository. This will make the service completely isolated from other services. Developer has high control over the microservice in continuous integration in this approach. Someone prefers having each service in its own repository to have access control. They can have fine-grained permission over the services. A disadvantage of this approach is that there are difficulties when it comes to sharing the communication interfaces between services. For example, programming language Go does not allow using dependencies from private repositories, because dependencies are cloned over HTTP. A workaround in the local developer machine is to configure git in a way that every clone of the repository over HTTP is done over SSH in the background. It is working on the developer's machine because the developer has access to all dependencies. This does not work in the continuous integration systems, because deploy key has to be added to the SSH client for each dependency that need to be cloned. Security mechanisms of the version control system providers do not allow creating one key for multiple repositories.

The second approach is storing all services in the one repository. Repository with multiple services or applications is called monorepo. Continuous integration tools have minimal support for this approach so sometimes so these tools have to be hacked to run standalone builds for each service. The scale of a monorepo can be different. Monorepo can contain 2 services or it can contain multiple projects. For example, Google stores a majority of their projects in one big monorepo [9].

2.1.3 MMO – Monorepo Microservice Orchestrator

MMO is an open-source tool that helps with the development of monorepo applications with microservice architecture[2]. MMO has a set of features that help generate the structure of a project and its services. Generation is done based on templates so when user wants to generate a service written in a different programming language then everything that has to be done is to provide another template.

Another feature of the MMO are plug-ins that are used for generating the code. Existing plug-ins are for example plug-ins that generate:

- *gRPC* stub and client from the *Protobuf* definition (*Protobuf* is a new mechanism for serializing structured data in binary format developed by Google [1]).
- *gRPC* gateway from the *Protobuf* – this allows us to generate a REST API from the Protobuf with extended annotations.

[2]https://github.com/flowup/mmo

- *Swagger* definition from the *Protobuf*. This plug-in works with combination with the previously mentioned plug-ins (*Swagger* definition is documentation of the REST API).

- *Angular* client from the *Swagger* definition in programming language *Typescript*.

This combination of the plug-ins allows to generate a *gRPC* server with REST API endpoints with type compatible frontend client.

MMO's version is actually Beta so a lot of features will come later. One of the missing features is an option to make deployment of the application to cluster easier.

2.2 Application Building and Distribution

Before building the application, we need to know the platform where the application will be running. Based on that the application is built on this specific platform and then it can be delivered to that platform.

Applications are orchestrated in multiple ways:

- Operation system runs directly on hardware and the application runs in the installed operating system.

- Hardware is virtualized and the application runs in the operating system installed on the virtualized hardware.

- The application is running in a container.

First of the options is not used because it takes a lot of effort when an application has to be migrated from a server to another one. The second option is mostly used because of the flexibility that is offered. The last option is latest and its advantages and disadvantages against the second option will be summarized in this section.

2.2.1 Release Models of the Software

Software has to be released after its development. Two most known release models of the software are standard stable release model and rolling release model [5]. According to the standard stable release model, software is released at intervals in which bugs are fixed and new features are added. These intervals can have a different length from software to software, e.g. some distributions of Linux operation system are released two times in a year. The second way of releasing software, rolling release model. Rolling release means that software is released frequently in small updates.

Rolling releases are more preferred because it is more natural to get new features when they are finished rather than waiting some time for a set of features when release planned.

When we choose to have a rolling release we have to care much more about release process because updates are much more frequent too (e.g. it can be with every push to the software repository).

2.2.2 Distribution of Binaries

Distribution of binaries is a standard approach that exists as long as software development. This approach is working but in modern software development, it starts to be outdated.

Binaries are usually pushed to the virtual machines that have all needed dependencies installed. Lot of time we run into the conflicts where multiple applications need same dependency but different versions. This leads to a lot of weird hacks and fixes. Another problem which can occur is running multiple instances of the one network application – operation systems do not allow us to bind the same port to multiple applications. Some applications allow us to change port but a lot of times it is not comfortable. Another disadvantage is that application can have non-deterministic behavior on different systems due to versions of dependencies where every new combination of versions can lead to new type of behavior.

Binaries can be distributed through operating system's package managers. Preferred way when free or open source software is distributed. Another way to distribute binaries is through ssh, for example, utility *scp*. We have to realize that the binary itself is not sometimes sufficient. We have to distribute configurations of the applications, secrets (passwords, API keys, . . .). One of the available solutions for this is *Ansible*.

Ansible is a tool for automating deployments. It allows us to push binaries or other files such as configurations [10]. *Ansible* also supports a lot of modules for managing dependencies through package managers on the different operating systems, managing systemd and a lot more. *Ansible* has the feature that makes possible to create templates (of configurations for example) that are dynamically filled before deployment.

Even though we have tools like *Ansible*, deploying applications is not easy and does not feel natural. This leads to researching new technology. Results of this researching are container technologies.

2.2.3 Containers

Containerization is technology that allows running applications in the isolated environment [3]. A containerization can be compared to virtualization. Virtualization is technology that virtualizes CPU, RAM and other hardware and we can run some operation system on this virtualized hardware.

Containers with virtualized operation systems have in common that they are isolated. Containers do not use virtualized hardware, they share the kernel with the host system. We can run multiple containers on one host system.

The container is fully isolated from the environment as was mentioned above. Containers do not see other processes running on the host machine or in other containers. They are running on their own network in default. They do not share any environment variables with the host system. Containers also cannot access host's systems volumes.

Container always has metadata and read-only volume that contains all data needed for runtime.

Containers offer multiple advantages over the binaries. The largest one is that we can pack all dependencies that application needs into the container. That will ensure that application will run on any platform and it will also run when two applications will use different versions of the same dependency. Containers have deterministic behavior and run in the same way on all platforms. That makes easy migration from one platform to another. Side by side comparison of application runtime in a virtual machine and in the container is shown in Figure 2.4.

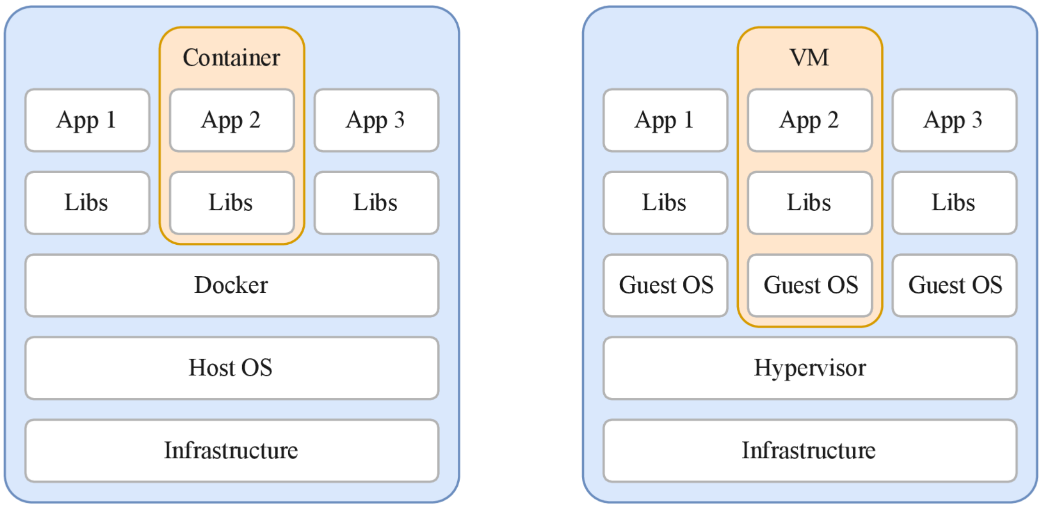

Figure 2.4: Comparison of the applications running in the containers (left) and in the virtual machines (right). Application running in the container is running directly on the host operating system. Application running in the virtualized operating system have to be running on the virtualized hardware.

The Open Container Initiative (OCI)

A lot of container solutions appeared and it was necessary to define some standard how containers should look and how they are supposed to run. In 2015 *OCI* was launched by *Docker*, *CoreOS* and other leaders in the container industry.

OCI is a lightweight, open governance structure (project), formed under the auspices of the Linux Foundation, for the express purpose of creating open industry standards around container formats and runtime [2].

OCI currently contains two specifications: the Runtime Specification and the Image Specification. The Runtime Specification outlines how to run a "filesystem bundle" that is unpacked on disk. At a high-level, an OCI implementation would download an *OCI* Image then unpack that image into an OCI Runtime filesystem bundle. At this point, the *OCI* Runtime Bundle would be run by an OCI Runtime.

Docker

Docker is the best-known container platform. It meets *OCI* standard (was the first contributor to the standard as mentioned above)[3].

Isolation is achieved using Linux kernel's features – cgroups, kernel namespaces and using OverlayFS to manage volumes.

The image is created on union filesystem (OverlayFS). Every image is based on base *Docker* image and there are multiple layers that describe how to recreate a filesystem as we see in Figure 2.5. This makes storing and sharing of the *Docker* images less resource intensive. When *Docker* image is transferred or stored, only new layers that are not present are transferred or stored. Each image has a tag. The tag is used for versioning of the images. Default tag is "latest".

[3]https://docs.docker.com

There is a difference between the Docker image and the Docker container. *Docker* image is set of layers, that are build based on the Dockerfile. This set of layers are read-only and they contain everything needed for runtime. When *Docker* image is run, then it is named Docker container. A container has a read-write layer above the read-only layers of the *Docker* image. It is a necessity for the application in the container so it can write changes. Difference between Docker image and container can be also seen in Figure 2.5. A read-write layer is discarded always on the container removal. To keep data persistent we need to mount needed directories or files to the host system.

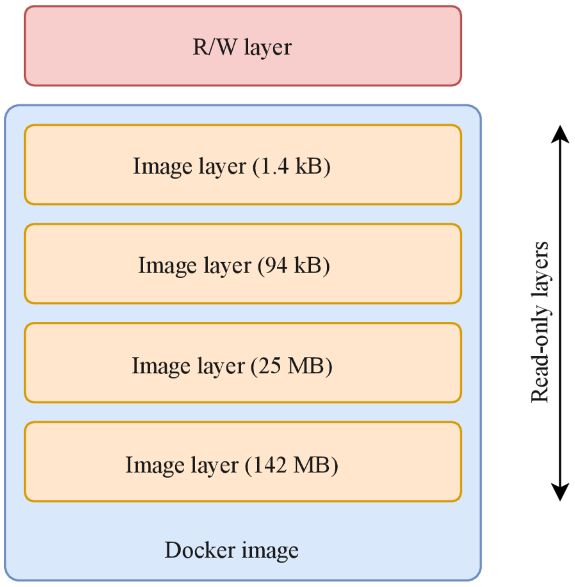

Figure 2.5: Structure of the *Docker* container – container is basically *Docker* image with thin read-write layer

Docker is not just a container provider but it is the whole platform. The most useful part of the platform is *Docker registry*. It is a tool to manage images. The image can be pulled from a remote registry to a local registry or local image can be pushed to the remote registry. There are registries like *Docker Hub*, *Google Container Registry (GCR)*, *Amazon EC2 Container Registry (ECR)* and more that are public and can be used to share Docker images between users or machines. These registries have usually secured connection and access control so the only user with valid credentials has access to private images. Distribution of images through mentioned registries is very easy thanks to native versioning of the images.

Docker is integrated into the multiple infrastructure tools and the most of the container orchestration tools as well:

- *Kubernetes*
- *Docker swarm*
- *Marathon on Mesos*

LXC and LXD

LXC (LinuX Containers) are very similar to *Docker* containers from the side of functionality. Isolation of the processes is done in the same way[4].

Definition of the container is slightly different in *LXC*. It aims to virtualization like the experience so there are images of Linux distributions available (Ubuntu, Fedora, RHEL, etc). A user should run these images like a virtual machine and install needed applications inside.

Does not support versioning and does not have ecosystem build around so managing LXC images is not as comfortable as in *Docker*. *LXC* is deprecated and it will only get security updates until April 2019. Support is moved to the new project called *LXD*.

rkt

The core execution unit of rkt is the pod, a collection of one or more applications executing in a shared context (*rkt*'s pods are synonymous with the concept of *Kubernetes* orchestration system)[5]. rkt allows users to apply different configurations (like isolation parameters) at both pod-level and at the more granular per-application level. *rkt*'s architecture means that each pod executes directly in the classic Unix process model (i.e. there is no central daemon), in a self-contained, isolated environment. *rkt* implements a modern, open, standard container format, the App Container (appc) spec, but can also execute other container images, like those created with *Docker*.

rkt meets *OCI* standard. The image format is different from the *Docker* images. It does not need the special registry for providing images. Appc images can be hosted on the HTTP(s) server. Its main advantage is that it can convert *Docker* image to its own format and then run it. Containers are run in the simpler way and *rkt* does not need daemon such as Docker. The disadvantage is that it does not have full support in orchestrators. Runtime is not possible without virtualization layer on operation systems Microsoft Windows and Apple OS X.

2.3 Deployment of the container applications

Deployment of the container applications to the cluster is much more abstract than deploying applications using Ansible as mentioned in the 2.2.2. Clusters can have a variety of sizes – clusters can be small where less than 10 nodes are present or there can be clusters with the size of hundreds of nodes. With infrastructure this big it is impossible to manage running applications manually. We use can use one of the container orchestration tool.

Container orchestrators are responsible for the runtime of the container application in a cluster. These applications do a lot of tasks to manage application runtime. These tasks are:

- Running container with provided configuration (published ports, used volumes, number of replicas, ...)
- Provide service discovery so services can communicate among themselves
- Restarting containers in case they are not healthy

[4] `https://linuxcontainers.org`
[5] `https://coreos.com/rkt/docs/latest/`

- Provide load-balancing and metric and log systems

2.3.1 Container schedulers

Docker Swarm

The solution developed by *Docker*[6]. It was originally a part of the Docker core. Local development with docker swarm can be done easily because *Docker Swarm* can be run as a single node cluster for these purposes. Applications are orchestrated using CLI that is very similar to standalone Docker. Another way is to use *Docker Compose* files. *Docker Compose* is the tool for running multi-container applications. *Docker Compose* file uses yaml format and contains all needed information about the application. This information is a list of containers to run, mounted volumes, published ports and the most of the options that are available in *Docker*. To write *Docker Compose* configurations for *Docker Swarm*, everything that is needed is knowledge of how *Docker* works. *Docker Compose* configurations are just *Docker* commands transformed to the configuration.

The disadvantage of *Docker swarm* is that it does not provide that much configuration options as other orchestrators. Another disadvantage is small support from cloud providers.

Kubernetes

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications [6]. Kubernetes has set of features that allow running applications in a massive scale. From all of the orchestrators, it has the widest community. It's developed by Google, Red Hat, CoreOS. There are also special Linux distributions that are build just for running *Kubernetes* – the two most known are CoreOS and RancherOS. Kubernetes also develops Minikube which is just one lightweight Kubernetes master node running in a virtual machine and can be used for running applications locally. This is a nice feature that can be used for local development. Kubernetes is also supported by largest cloud providers – Google Cloud Platform, Amazon Web Services, Microsoft Azure.

Kubernetes has a lot of configuration options. This is a disadvantage from the point of view that user needs very wide knowledge to deploy a working application. Deploying the single container application is not as easy as with other orchestration tools. We can see an example of *Kubernetes* configuration in the code 2.1. Configuration represents an application that has only one service – *Redis* database. In the code 2.2, we can see the configuration for the same application but for the *Docker Swarm*. Kubernetes supports Role-based access control (RBAC) so users can have restricted access only to parts of *Kubernetes* that are responsible for.

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    io.kompose.service: redis-master
  name: redis-master
spec:
  replicas: 1
  template:
```

[6] https://docs.docker.com/engine/swarm/

```
    spec:
      containers:
      - image: k8s.gcr.io/redis:e2e
        name: redis-master
        ports:
        - containerPort: 6379
---
apiVersion: v1
kind: Service
metadata:
  name: redis-master
spec:
  ports:
  - name: db
    port: 6379
    targetPort: 6379
```

Listing 2.1: *Kubernetes* configuration that can be used for deployment of *Redis* database to *Kubernetes* cluster.

```
version: "3"

services:
  redis-master:
    image: k8s.gcr.io/redis:e2e
    ports:
    - "6379:6379"
```

Listing 2.2: Configuragation that can be used for running *Redis* database using *Docker Compose*. Docker Compose configurations are less complex than configurations for Kubernetes that can be seen in Listing 2.1

Company Red Hat develops its own version of Kubernetes called *Openshift*. *Openshift* offers additional functionality to *Kubernetes*. It brings new security concepts to *Kubernetes*. One of these concepts is that containers are not run in standard way under root user but they are run by the non-root user. Because of this common images are unusable but RedHat offers popular images that are compatible with the OpenShift. Other features that are added to OpenShift is multi-tenancy, tools for project management or continuous integration and deployment tools.

Nomad

Nomad is scheduler developed by HashiCorp[7]. *Nomad* is more general purpose than *Kubernetes* or *Docker swarm*. *Nomad* supports virtualized, containerized and standalone applications so we can run *Docker* container or application's binary. Nomad is designed with extensible drivers. Nomad has much simpler architecture than Kubernetes. It does not have features like service discovery but a lot of functionality can be extended by HashiCorp's other projects – *Consul* for service discovery, *Vault* for key and password management, etc.

[7] https://www.nomadproject.io/docs/

Mesos

Mesos is a solution from company Apache[8]. *Mesos* provides resources allocation in the data center based on utilization of the resources. It can switch nodes on when needed and it can switch off nodes that are no longer needed. *Mesos* can be fine-tuned for different applications.

Mesos is great for large-scale applications that are heterogeneous. The application does not have to be just bunch of the containers. It does not have support from cloud providers so hardware needs to be bought or rent.

The disadvantage of Mesos is that it cannot run locally, similar to *Kubernetes*, it does have tool *Mini Mesos* that makes possible experimenting and testing. *Mesos* is much more heavyweight than Docker swarm or Kubernetes.

[8]`http://mesos.apache.org/documentation/latest/`

Chapter 3

Kubernetes

Kubernetes was selected as a tool for orchestrating applications. *Kubernetes* was selected because of the options it provides when running applications and the second important reason is its community. Good community is priceless when it comes to some problem solving. Other advantage over other tools is the support of the cloud providers – it allows effortless migration of an application to the different provider in the case of need.

3.1 Kubernetes Architecture

3.1.1 Kubernetes Node Types

Kubernetes cluster needs at least two nodes in the cluster and at least one of the nodes have to be master node [6]. Other nodes are simply called "nodes". In a case, where high-availability cluster is needed, there may be multiple master nodes. Master node is always running tools that are making global decisions about the whole cluster, detecting and responding to the cluster events.

3.1.2 Kubernetes Master Components

These components are running on Kubernetes master nodes [6]:

- `kube-apiserver` – component that exposes Kubernetes API (Kubernetes API is used to control Kubernetes).
- `etcd` – consistent and highly-available key value store used as Kubernetes backing store for all cluster data.
- `kube-scheduler` – component that watches newly created pods and assigns them node where they should run.
- `kube-controller-manager` – component that runs controllers. Controllers are tools that are watching the state of the cluster and are making changes to reach the desired state of the cluster.
- `cloud-controller-manager` – component that runs controllers that interact with cloud providers.

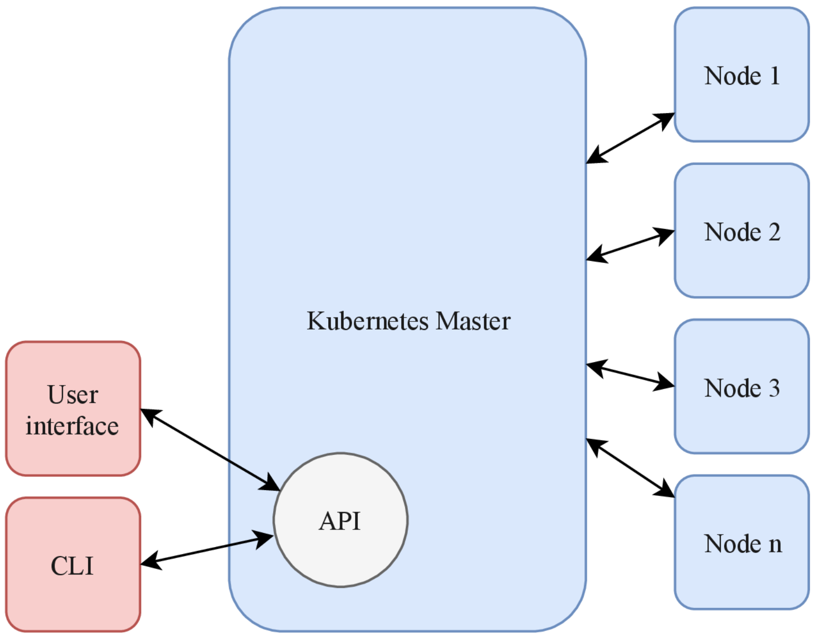

Figure 3.1: Architecture of Kubernetes. Kubernetes Master node is running API for controlling cluster and communicating with other nodes.

3.1.3 Kubernetes Node Components

These components are running on every node (even on the master nodes) [6]:

- `kubelet` – component that ensures that pods assigned to node are running and healthy.
- `kube-proxy` – component that is responsible for networking (service discovery, DNS resolving and more).
- `Container Runtime` – software responsible for running containers (Docker, rkt, runc or any container implementation meeting OCI runtime-spec as mentioned in the 2.2.3).

3.2 Basic principles of Kubernetes

As mentioned before, we are describing final state of the application that we want to reach. Kubernetes is trying to reach this state with its tools. For describing desired state we are using Kubernetes workloads [6]. After deploying workloads to the cluster, application will not run instantly based on the deployed workload, but Kubernetes will use workload to deploy application. There are different types of these workloads. The most important workloads are described in the section 3.3.

Kubernetes has also other type of resources than workloads – for networking (mentioned in the section 3.4), managing configurations and volumes (mentioned in the section 3.5). We can call all these Kubernetes objects. Each object of one kind must have a unique name within a namespace.

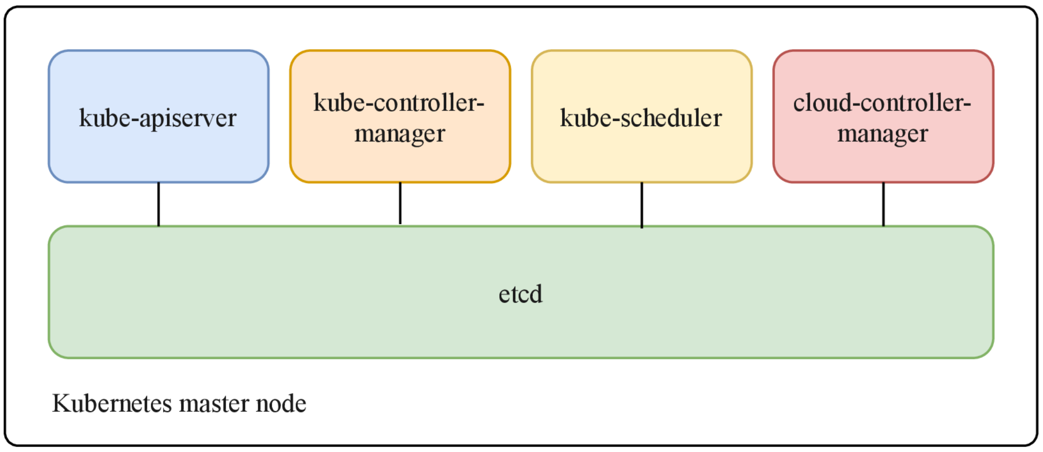

Figure 3.2: Components present in Kubernetes Master node. All components are using component etcd for storing data about the cluster.

A namespace is a way to divide cluster logically for different reasons[1]. One of the reasons can be running of multiple environments of application in each namespace. Each namespace can have CPU and memory resource restriction.

Labels are key/value pairs that can be tied to objects[2]. Labels do not have to be unique like names. Labels can be used for organizational purposes. For example, we can add label "release = beta" to object, for another object we can add the label "release = canary". Labels are sometimes necessary in specific Kubernetes objects – Kubernetes Service is using label selector to determine which Pods are able to receive network traffic. Another usage of the labels is object listing and filtering based on the defined labels. Objects should contain only necessary labels. When we want to add key/value meta-data that is just informational we should use annotations.

3.3 Description of Kubernetes Workloads

3.3.1 Pods

A pod is the most basic unit that can be deployed to Kubernetes [6]. As we can see in Figure 3.4, the pod encapsulates container (or group of tightly coupled containers), storage resources and a unique IP. For example, we can have a pod that has two containers, where one container is gRPC server and a second container that acts as an HTTP proxy to the gRPC server in the first container. The pod can be configured in a way that we know from the configuration of the Docker container. We can mount volumes to pod's containers, publish ports, set up health checks and more.

Another thing we can configure is container's hardware resources – requests and limits of the CPU and memory. The request is maximum of the resources the container can take. Kubernetes allows the container to allocate more resources than defined in the requests, when requested resources are free on Kubernetes node. The resource requests is also an information that Kubernetes uses for scheduling pods on the nodes. Pod is not schedulable on the node that has less free resources than a pod requests. The limit is the value of which

[1]https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/
[2]https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

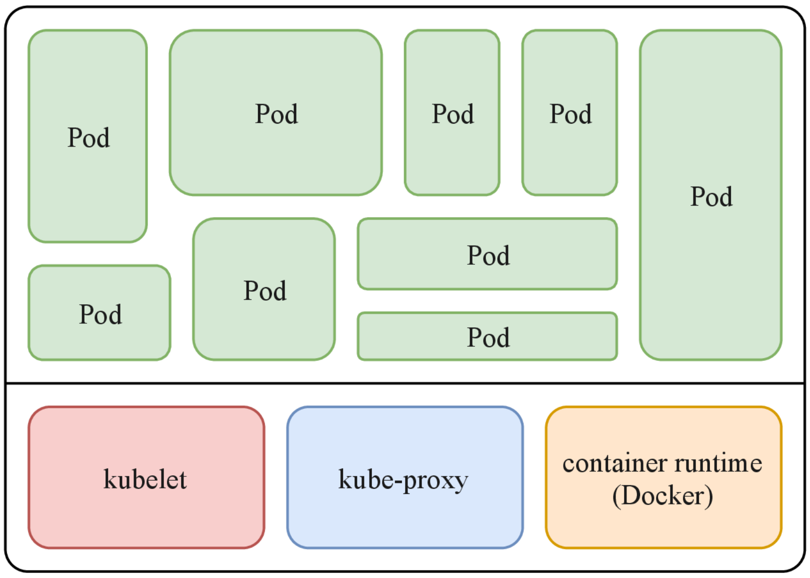

Figure 3.3: Components present in Kubernetes node. Kubelet is the spart that is communicating with Kubernetes master node, and it is running Pods using container runtime. Kube-proxy is responsible for service discovery

are resources restricted. Second parameter – limit – is used as a hard resource limit for the container. If container takes more resources than its defined limit then container will be marked for termination in the short time. If the container does not free the resources in this short time, it will be restarted.

The pod is intended to serve as a single instance of the running application. For horizontal scaling of the application, multiple pods should be created.

3.3.2 Deployments

A deployment is abstract Kubernetes object that is describing how deployed application should look like [6]. Deployment has always pod template – pods are created based on this template. Deploying deployment resource to Kubernetes will indirectly create number of pods (based on the number of replicas) by creating ReplicaSet object. Deployments have set of advanced features for updating or roll-backing the application. New ReplicaSet is created always when the Deployment is changed.

One of the features are rolling updates. The rolling update is a process of updating the application when we have more than one replica of the application. When rolling update is triggered, Pods are replaced one by one with newer version. Deployment has parameter “replicas” that says how many of the replicas should be available during the rolling update process. With this feature, we can update application without downtime. Reverse operation to rolling update is called roll back. This can be used when application has some serious bug and we want to downgrade application to the previous version.

3.3.3 Replication Controllers

Replication controller is the resource that controls the creation of the pods in the cluster [6]. It uses the template of the Pod for Pod creation. Replication controller must ensure that

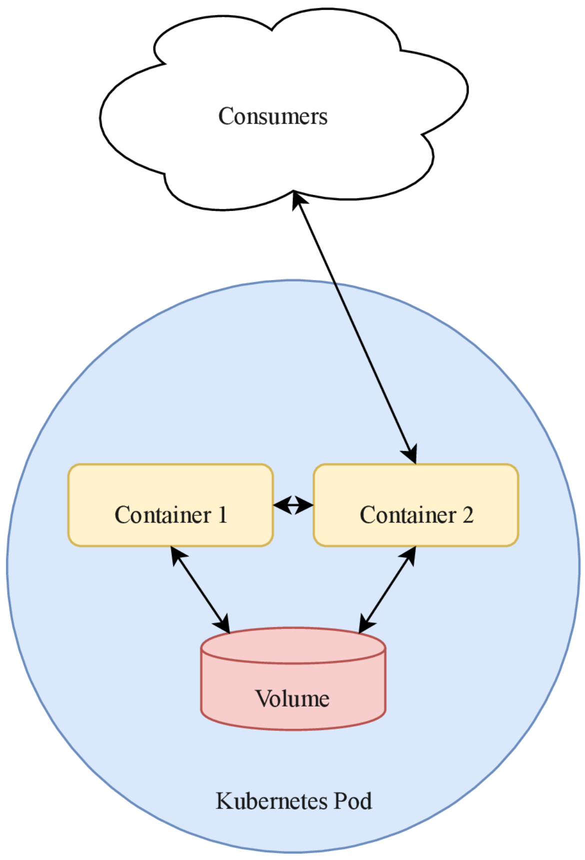

Figure 3.4: Example of Kubernetes Pod that has two containers communicating between themselves and one container is serving to consumers (clients). Both of containers are using one shared volume.

number of replicas defined in configuration is same as a number of replicas running. Replication controllers support rolling updates as the Deployments, but they are not automatic and they have to be triggered manually using kubectl tool.

In actual implementation it is not recommended creating replication controller. We should create a resource called "Deployment" that manages the creation of ReplicaSets.

3.3.4 ReplicaSet

ReplicaSet is next generation of Replication controller that has the support of label selectors [6].

3.4 Description of the objects for network management

3.4.1 Services

Pod has IP address assigned on the creation. Pods have to communicate between themselves and it would not be possible only using IP addresses. Kubernetes Services are used for service discovery and load-balancing between Pods [6]. Pod that should be accessible over the network needs to have Service created. After the creation of the Service, DNS entry is reserved inside the namespace for all Pods that are matching Service's label selector. We can then access the Pod using this DNS name `<service-name>` within the namespace. In case we want to access this service outside of the namespace we have to use fully qualified domain name `<service-name>.<namespace-name>.svc.cluster.local`.

Services have different types. Default Service type is ClusterIP. This makes Service accessible only from the inside of the cluster. Next option is NodePort. This will make Service accessible from the outside of the cluster on the every node in the cluster (Each node has range of the ports reserved for this purpose). There are other options such as LoadBalancer that are only available in Kubernetes instances from cloud providers. Option LoadBalancer will make Pod accessible based on the configuration of the load-balancer – each cloud provider offers different configuration.

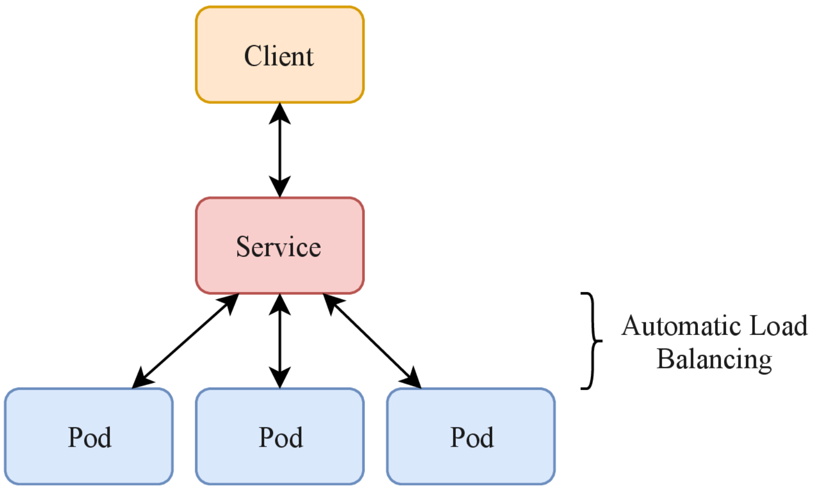

Figure 3.5: How Kubernetes Services works – multiple instances of one service are accessed by one Kubernetes Service. Component kube-proxy is responsible for correct functionality of this.

3.4.2 Ingress controllers and Ingresses

Ingress controller is a gateway to Services from the outside of the cluster [6]. Ingress controller works on the application level of the network model. Ingress controller is usually reverse HTTP proxy (Nginx for example) with watcher of Kubernetes resources named "Ingress". When watcher captures changes in Ingress resources, then new configuration for the reverse proxy is generated and reverse proxy is reloaded. Multiple ingress controllers are available:

- Official Nginx ingress controller (developed by Kubernetes).
- Nginx ingress controller (developed by Nginx).
- GKE (Google Kubernetes Engine) ingress controller.
- Træfik ingress controller – based on the reverse-proxy Træfik developed in Go.

Each ingress controller has slightly different set of the features. These features can be:

- Usage of the TLS (Transport Layer Security) certificates for the used domains.

- Enabling CORS (Cross Origin Resource Sharing) for the locations[3].
- Websocket support
- Authentication
- And more...

Ingress contains information how to route HTTP(s) requests to individual services. Ingress controllers can usually route requests based on sub-domains and paths.

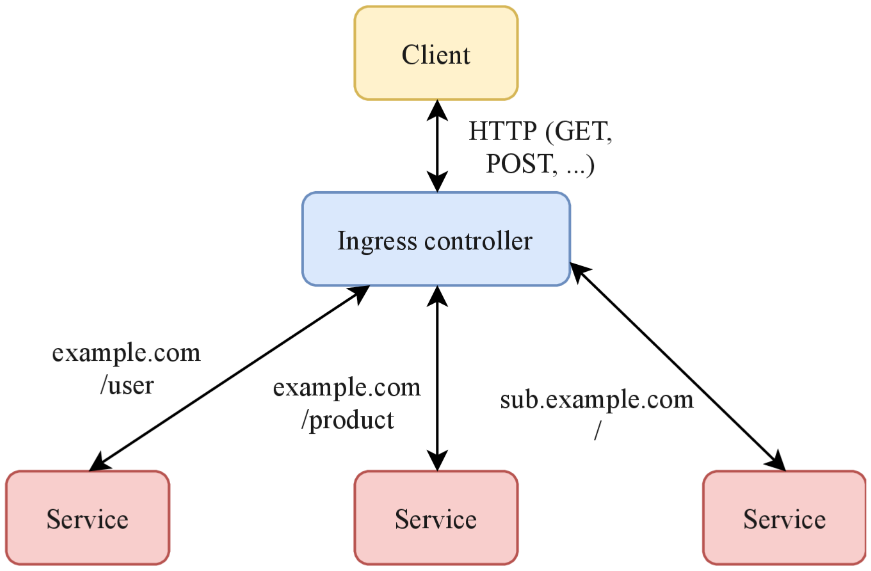

Figure 3.6: Kubernetes Ingress controller works on the application level of the network model. This allows routing of the HTTP requests to different Services based on hostname or path they are requesting.

3.5 Description of the other Kubernetes objects

3.5.1 Configmaps

Configmaps are resources for storing configuration files [6]. Configmaps can be used in the application multiple ways. One option is configuration in a special format where each line contains key-value pair in format `KEY: VALUE`. Each line of the Configmap is exported to the container of a Pod as a environment variable. Second option is mount whole configuration file of Configmap as a volume in the Pod's container. These two are most used and other options are documented in Kubernetes documentation[4].

3.5.2 Secrets

Secret is very similar Kubernetes object to Configmap. Only different is that secret should be used for storing sensitive data such as keys and passwords. Usage of the secrets is well documented in Secrets section of Kubernetes documentation[5].

[3]https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
[4]https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/
[5]https://kubernetes.io/docs/concepts/configuration/secret/

3.6 Storing Kubernetes configurations in the repository

3.6.1 Static Kubernetes resources or Kubernetes resource templates

In a lot of cases, we do not want static deployment resources. Static resources are good in the case that deployed resource is not changed on regular basis, for example, databases. When we want use Rolling Updates advantage of Kubernetes Deployments, we have to change the image in the Deployment resource before deploy. This can be done in two ways.

First option is to store template of Kubernetes resource – this can be bash template (variables in the tempalate have format `${GIT_COMMIT_HASH}`). Advantage of this is that resources can be much more dynamic. All variables in Kubernetes resource can be changed in one step. For example, these variables can be: Docker image or its tag, labels in the application that can contain deployed commit or branch, etc. Disadvantage of this option is that application cannot be deployed directly using `kubectl` but it has to be preprocessed using tool that fills out variables in the template.

Second option is to store normal Kubernetes resources. Resources are more static and change of each value must be done in one step. Application should be running after deploy to Kubernetes without changing values in the resource. When we want to deploy different version of the application, we need to change Docker image in the resource. This can be done in the continuous integration system using utility `sed`.

Second option will be more useful, and it will also save problems that are described in the 3.8.

```
kind: Service
apiVersion: v1
metadata:
  name: example-service
spec:
  selector:
    app: example-service
  ports:
  - protocol: TCP
    port: 80
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: example-service
  labels:
    commit: ${GIT_COMMIT}
    branch: ${GIT_BRANCH}
spec:
  replicas: 1
  selector:
    matchLabels:
      app: example-service
  template:
    metadata:
      labels:
        app: example-service
        commit: ${GIT_COMMIT}
        branch: ${GIT_BRANCH}
    spec:
      containers:
      - name: example-service
        image: ${DOCKER_REGISTRY}-example-service:${GIT_COMMIT}
        ports:
        - name: http
          containerPort: 80
          protocol: TCP
        env:
        - name: GCP_PROJECT_ID
         value: ${PROJECT_NAME}
```

Listing 3.1: Example of Kubernetes resource template

3.6.2 Structure of Kubernetes resources

Kubernetes resources can be structured in many ways. Universal way to structure Kubernetes resources does not exist. Each application has different needs and different size, according to which we select best structure for files. These ways can be:

- Directory that contains all configurations – this can be fitting for small applications that share configurations between environments (staging, development, production).
- Directory that contains sub-directory for each environment – complex applications that are running in different ways between environments.
- Combined structure – configurations that are shared are in one directory and others that are different for environments are in the specific environment directory.

Survey about structuring Kubernetes configurations is in the next section 3.7. These three mentioned above are the most used and some other can be used too in some specific applications.

3.7 Survey about using Kubernetes

Survey about Kubernetes was conducted for two reasons. One reason is verify that other users are doing mistakes with `kubectl` tool when they are deploying applications to Kubernetes. Other reason for conducting the survey is to understand how users are using Kubernetes. Survey form had only four questions so it does not discourage respondents from filling out the form. Questions were following:

1. Do you use Kubernetes resource templates? (Yes or No)
2. How do you structure your Kubernetes resources? Provided multi-choice options were:
    - All environments (development, staging, production) share one configuration
    - Each environment has its own configuration

    - Combined – some configurations are shared across all environments and others are created for each specific environment

3. How many times did you deploy application to different namespace? Provided options were:
    - 0
    - 1 – 5
    - 5+
4. How many times did you deploy application to different cluster? Provided options were:
    - 0
    - 1 – 5
    - 5+

Respondents had option to leave the comment that can extend their selected answer. Survey form was posted to Kubernetes community on the social network Reddit and 26 respondents have filled the survey form.

In Figure 3.7, we can see on the chart that the most of the users are using the templates of Kubernetes resources. One respondent did not know what are Kubernetes resource templates and one user noted that they are using Helm charts (mentioned in the 4.2) and only thing that is template are Configmaps that are used as environments mappings. Second chart is shown in Figure 3.8 and it shows how users structure their Kubernetes resources in the repositories. One of the respondents noted that they have shared resources between environments and they have only different environment mappings.

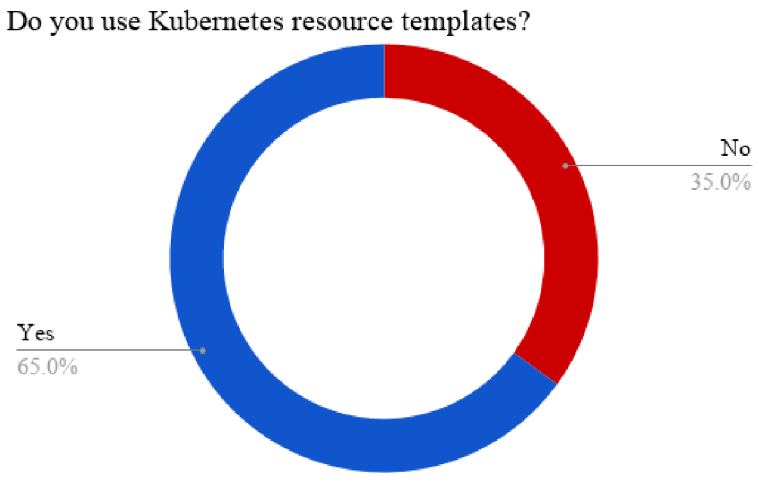

Figure 3.7: Chart shows how much of Kubernetes users are using Kubernetes resource templates

Second part of the results is talking about mistakes that are done with `kubectl`. We can see that only 27 % of the respondents did not deploy application to the different namespace on the chart in Figure 3.9. Similar numbers are resulting from question that

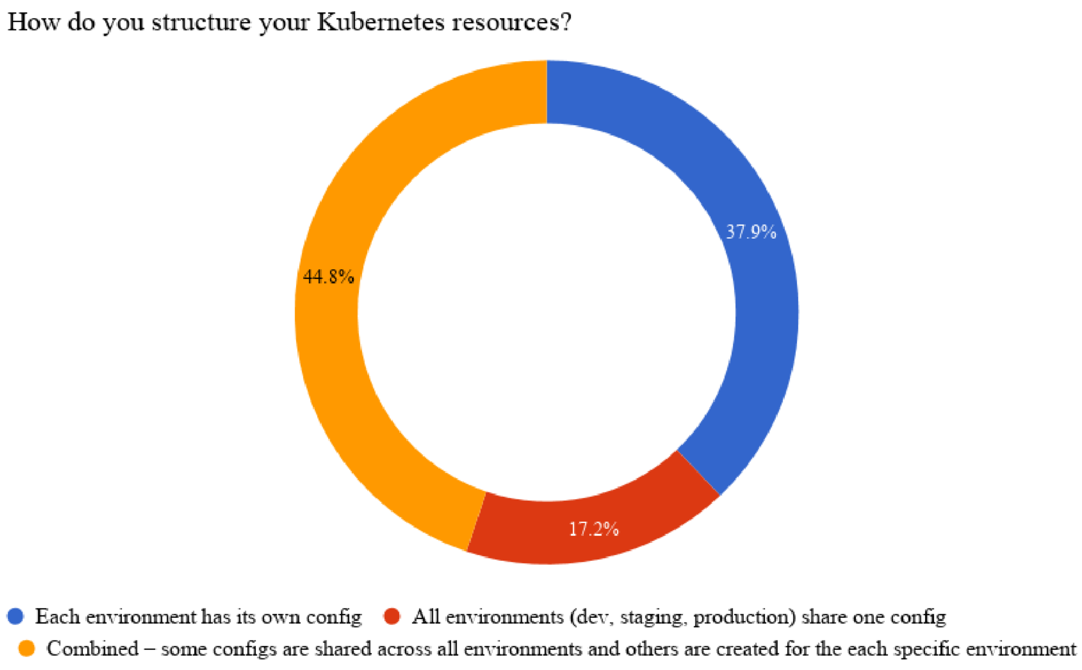

Figure 3.8: Chart shows how are Kubernetes structuring their Kubernetes resources

talks about deploying application to the different cluster. Only 38 % of the users did not deploy application to the different cluster yet as we see on the chart in Figure 3.10. One respondent stated that first or second mistake cannot happen in their company, because each state-modifying kubectl commands have to be reviewed and approved by the other coworker.

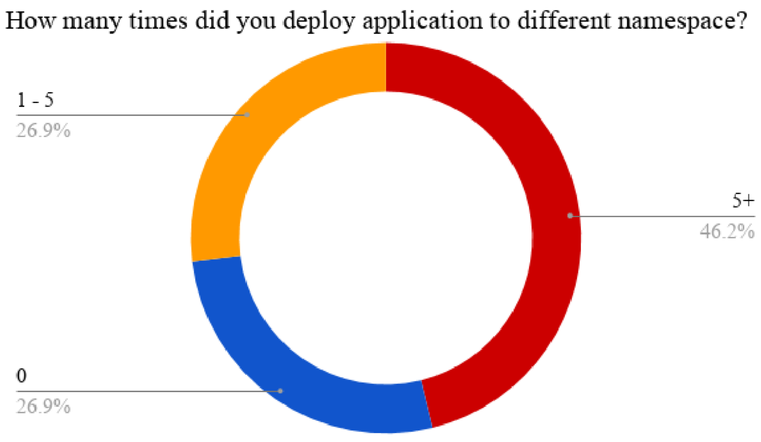

Figure 3.9: Chart how many times they deployed application to the wrong Kubernetes namespace

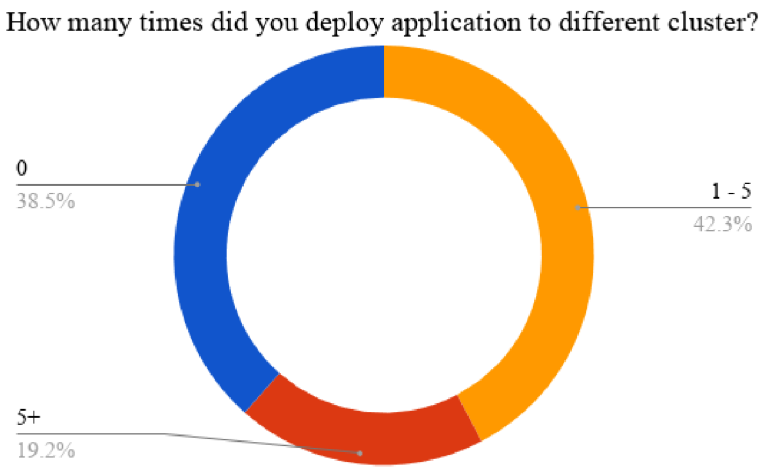

Figure 3.10: Chart shows how many times they deployed application to the wrong Kubernetes cluster

3.8 Local Development

Local development of the application should be done in the environment that is similar to production environment as much as possible. With this approach, we can avoid problems that results from the differences between environments. This is possible with minikube. Minikube is official tool for running lightweight development cluster locally[6]. Cluster has only one node that is running in the virtual machine.

Minikube is controlled as a normal cluster – with kubectl tool. We would need to maintain two versions of the deployment resources – one that is the template used in the continuous integration and the second one is resource ready for usage in Kubernetes. This can bring huge inconsistencies between staging and local environment. Another option is running some utility that fills the template before the every deploy to the minikube. Templates can be problem in a local development. It creates one more step that is needed to deployment of an application. Another problem that can be encountered is filling labels that are not available in the time of development such as git branch or git commit hash.

[6] https://github.com/kubernetes/minikube

Chapter 4

Existing solutions for deployment of applications to Kubernetes

Kubernetes can be controlled using official command line interface tool kubectl or using its API. *kubectl* is official console tool developed by Kubernetes so it supports all operations for Kubernetes management. *kubectl* is communicating over Kubernetes API and supports multiple versions of Kubernetes API (is back-compatible). With kubectl, for example, resources can be managed, logs from pods can be viewed, deployments can be scaled, rollback can be issued. *kubectl* is a tool for manual managing of Kubernetes. It does not support any type of the automation. Automation is usually done by creating scripts that use kubectl to manage Kubernetes and applications running inside. Another option is running Kubernetes Dashboard using command `kubectl proxy`. Kubernetes Dashboard is user interface for controlling and monitoring Kubernetes.

4.1 Continuous Integration (CI) solutions

CI providers support Kubernetes using its plug-ins. These plug-ins are usually wrappers of kubectl tool.

Wercker has step named kubectl. This step is just wrapper of the kubectl tool and this step can do all kinds of operations that kubectl supports. In case we do not want to use static deployment resources we can transform these resources to bash templates and use the step that fills the template with values from system environment, and then we can use kubectl step to deploy this filled template.

On a platform Travis CI, we can create the script that installs kubectl tool that we can later use. Solution to use resource templates is similar to the solution in Wercker.

Other CI providers offer similar solutions for deploying applications to Kubernetes as two mentioned above.

4.2 Helm – package manager for Kubernetes

Helm is package manager for Kubernetes[1]. It allows deployment of application with its dependencies to the cluster. The package is named chart. The chart contains chart descriptor, Kubernetes resources templates, file with application configuration and helpers.

[1] https://github.com/kubernetes/helm

Kubernetes resources templates are filled with application configuration before deployment and helpers can provide an interactive way to finish deployment. Helpers can be used for fine-tuning deployment for selected environment. Deployment can be slightly changed depending on the environment that is selected – minikube, self-hosted cluster, cluster from cloud provider, etc.

Chart can also have dependencies – list of other charts that should be deployed with the main chart. One of these dependencies can be database, for example.

Helm charts would be good solutions for our usage, but they are complex for common usage and developers and DevOps engineers would have to learn how to use it.

4.3 Other existing solutions

In the beginning of the March 2018, Google created open-source tool skaffold[2] for deployment of applications to Kubernetes. skaffold helps with deployment to Kubernetes in the multiple ways:

- Local development with minikube.
- Remote development with dedicated cluster.
- Deployment of the application to the staging or production environment.

skaffold also supports code watching and automatic building of the Docker images and deploying application. Power of this tool is development mode where source code of the application is watched and application is built and deployed automatically when change is made.

[2]https://github.com/GoogleContainerTools/skaffold

Chapter 5

Design of the MMO Extension for Application Deployment

The *MMO* extension for deploying monorepo applications with microservice architecture was designed as a user interface. The user interface can show a lot more information to the user and it is less confusing in this type of application than command line interface. User interface of the *MMO* will be used for managing Kubernetes configurations and deploying application or its services but can be later extended with existing features of the MMO that are actually implemented as the features of the command line interface.

User interface will have backend and frontend part. Each part has section dedicated to it below. User interface will have multiple parts, each function the tool provides will have dedicated part of the interface.

One of the sections should be overview of the services that are in the project. Overview has list of the services where each of the services will have a link to standalone service detail page.

Service detail page will contain list of *MMO* plug-ins that service use. Second part of the service detail page is list of Kubernetes resources that service has. Management of the configuration with editor will be useful in the projects of bigger size (more than 10 microservices) because developer does not have to search for multiple resources in one folder, but he will see just the that resources that belong to service. This is possible due to fact that user is developing usually one service at a time. Service detail page will also contain tool for generating Kubernetes resources.

Second section of the application is used for deploying application. The application can be deployed directly to Kubernetes cluster. Another way to deploy an application can be GitHub deployment which can be utilized in the continuous integration and continuous deployment tools for deploying the application.

The data transportation is defined in the Protobuf definition from which both server and client are generated.

5.1 Design of the Backend

As shown in Figure 5.1, the backend part of the application will be a gRPC server written in Go and it will work as a layer above the MMO core. gRPC is using protocol HTTP/2 for communication and it is not fully supported by all browsers. Due to this, a proxy will between the frontend client and the gRPC server. This proxy translates HTTP requests with JSON body to the gRPC requests.

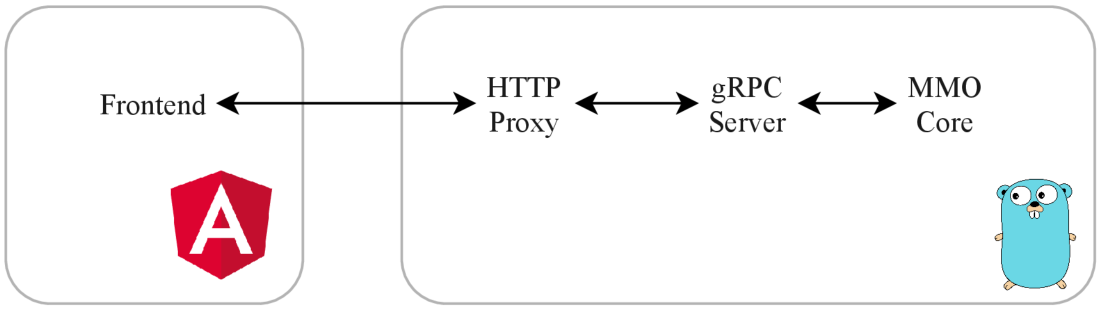

Figure 5.1: Design of the application. Angular Frontend communicates with gRPC through HTTP Proxy and gRPC server is built on the top of existing core of MMO

The application will be developed using MMO. The API will be defined using Protobuf. We will use set of MMO plugins that:

1. Generate gRPC server and stubs from the Protobuf definition
2. Generate REST API from the Protobuf definition (this REST API is proxy that translates a RESTful JSON API into gRPC[1])
3. Generate Swagger definition from the Protobuf definition
4. Generate Angular API client from the Swagger definition

This set of plug-ins ensure that communication between frontend and the backend of the application is type safe. There are only two things that have to be implemented – business logic of the backend of the application and visual part of the frontend of the application.

5.1.1 Design of Kubernetes Resource Generation

This section will talk about generation of Kubernetes resources in detail. For Kubernetes resources generation, we need information about:

- Ports used by services
- Volumes that services need to being persistent (if they need to be persistent)
- Environment variables used by service

After collecting this information we can generate Kubernetes resources. Resources are generated to directory named `infrastructure` in the root of the project. This directory will contain multiple directories – one directory per target environment.

In the results of Kubernetes usage survey in Section 3.7 can be seen that most of Kubernetes users are trying to share resources across environments as most as possible. Because of this, directory `shared` will be always present and it will be used for storing resources that are shared across environments. It will also be the default directory for Kubernetes resource generation. Other directories can be for example: `production`, `staging`. Deploying application is easy with this structure of resources. When we want to deploy the

[1]https://github.com/grpc-ecosystem/grpc-gateway

application to staging environment be have to deploy resources from `shared` directory and resources from `staging` directory. Another advantage is that the services are as same as possible across environments and little nuances between environments can be in environment specific folder. This can be for example environment mapping stored as a Configmap (approach used by some Kubernetes users that results from the survey in Section 3.7).

Another problem that has to be solved is format of the generated Kubernetes resource. As mentioned in Section 3.6.1, we can generate Kubernetes resource or its template. Results from survey in Section 3.7 say that the most of the users prefer templates. Generated resources by MMO are not very complex and each user prefers different type of variables in the resource template. This can be also dependent on the type of application that is user orchestrating in Kubernetes. Only one variable – Docker image name – is worth of swapping by variable currently generated resource. Due to this fact, generated resource will not be template. If some user wants to use templates, he just needs to swap image name by variable and extend Kubernetes resources with his parameters.

Generation of the resources is implemented using Go HTML templates. This library is part of the standard library of Go. Go HTML templates are used for generating static HTML web sites, but they will be also useful for generating Kubernetes resources. For generating file, we need template and object that is passed to template. Properties of the object will be used for filling the given template. Templates also support more advanced features such as loops and conditions. For example, loops can be used for adding multiple ports to the Kubernetes Deployment and Service.

Deployments

Deployments are the most important resources along with the Services. For generating deployment file we need this information:

- Ports used by service
- Mounted volumes to the container
- Environment variables used by the application

More advanced features of Kubernetes will not be generated. There are lot of these advanced features and form for generating resources would be cluttered. When some feature will be demanded by number of the users, it can be added to the form later.

Services

When user provides information about the used ports we can also generate Kubernetes Services. We will be assuming that ports that user provided are using TCP protocol.

Persistent Volume Claims

Another resource that we want to generate is persistent volume claim. For generating this type of resource only information we need is mounted volume and its size. Persistent Volume Claims (PVC) are generated always when the user specifies that he wants to persist data on disk of a node. We generate PVC resource when user specifies volume mount during generation process.

Configmaps and Secrets

It would be a nice feature to generate Configmap and Secret resources but these are created directly from file using kubectl tool.

Ingresses

The ingress resources are quite simple to understand and it would be counterproductive to make tool for generating this resource. Another disadvantage of the generation is that resources can have little differences that depend on the type of the Ingress Controller used in Kubernetes.

5.1.2 Application deployment design

This section will talk about deployment of the application to the Kubernetes in detail.

GitHub Deployment

GitHub deployments is feature hidden to regular users of the Github but it is mentioned in API documentation. However, documentation of this functionality is very poor. Some experimentation was needed to determine what are the deployments good for and how this endpoint works. After some time spent with experimentation with deployment endpoint on the GitHub and continuous integration tools, it has been found out that GitHub Deployments can be very useful feature for continuous delivery. In the continuous delivery process, manual approval for the deployment by the human is needed [5].

Github Deployments work in the following way:

1. POST request to Github deployment API is sent with the following information – reference (branch, commit hash, tag, release, etc[2]), environment and message (other optional request parameters are in the Github's API documentation[3]).

2. Github will check reference and if reference is valid then Github sends webhook requests to all webhook services registered in the repository which are listening to event type "Deployment".

3. Receiver of the webhook event can utilize information about deployment, for example, continuous integration tools can run deployment pipeline of the application.

Advantage of deploying using continuous integration tool is that all steps on which deployment depends are done (build, test, etc). If continuous integration is set up then deploy from CI is preferred because individual steps of the pipeline done by people can be done with mistakes. These mistakes can lead to putting application down.

GitHub deployments are using GitHub API and GitHub personal access key is needed for API authentication. This key should be exported to environment variable `GITHUB_TOKEN`. MMO will read API key from the environment variable and utilizes it when GitHub deployment is invoked from the user interface.

[2] https://developer.github.com/v3/git/refs/

[3] https://developer.github.com/v3/repos/deployments/

Manual Deployment

Manual deploy should be used when application is deployed to the cluster and Github deployment is not available. Manual deploy takes all files from "shared" directory and all files from environment specific directory that user provided as a target environment. Confirmation summary is shown with the list of resources that will be deployed and information about the cluster. When user confirms the deployment, confirmation is sent to the backend, where kubectl tool is invoked to deploy resources to the cluster. Log from the deployment is sent back to the frontend so user knows about the success or fail.

5.2 Design of the Frontend

Frontend part will be implemented in web application framework *Angular*. Frontend will be designed according to *Material Design* Guidelines[4]. People are familiar with *Material Design* and another advantage is that *Kubernetes* Dashboard uses *Material Design* too, so people will get used to it faster. Frontend will communicate with backend as shown in Figure 5.1 in the previous section.

Angular framework allows us to create application composed of reusable components. Whole application is one big component. This big component is composed of smaller components and these smaller components can be again composed of even smaller components. Application's main component will be divided to the 4 main components:

- Overview Component
- Service Detail Component (accessible from the Overview Component)
- Plug-ins Component
- Deployment Component

Main application's component will always have 2 navigation components: *Material Toolbar* and *Material Navigation Drawer* that has buttons for accessing 4 components mentioned above. A button of component that is active will be highlighted. The component that is active in default will be Overview component after accessing application. Toolbar will contain only button for toggling Navigation Drawer. It can be later used for adding another buttons. Position of the Toolbar and Navigation Drawer can be seen in Figure 5.2.

5.2.1 Design of the Overview Component

Overview component is used for showing overview information about *MMO* project to the user. Second part of the overview can show list of the *MMO* services to the user. Each service present in the list has also its description and hyper link to the detail of the service. After clicking this link, Service Detail Component will be shown to the user. Service Detail Component is described in the following section 5.2.2. Component will have two *Material Cards* for both project information and list of services. This can be seen in Figure 5.2.

5.2.2 Design of the Service Detail Component

Service Detail Component should show information about particular service. This information involves a list of *MMO* plugins that service uses and list of Kubernetes resources

[4]https://material.io

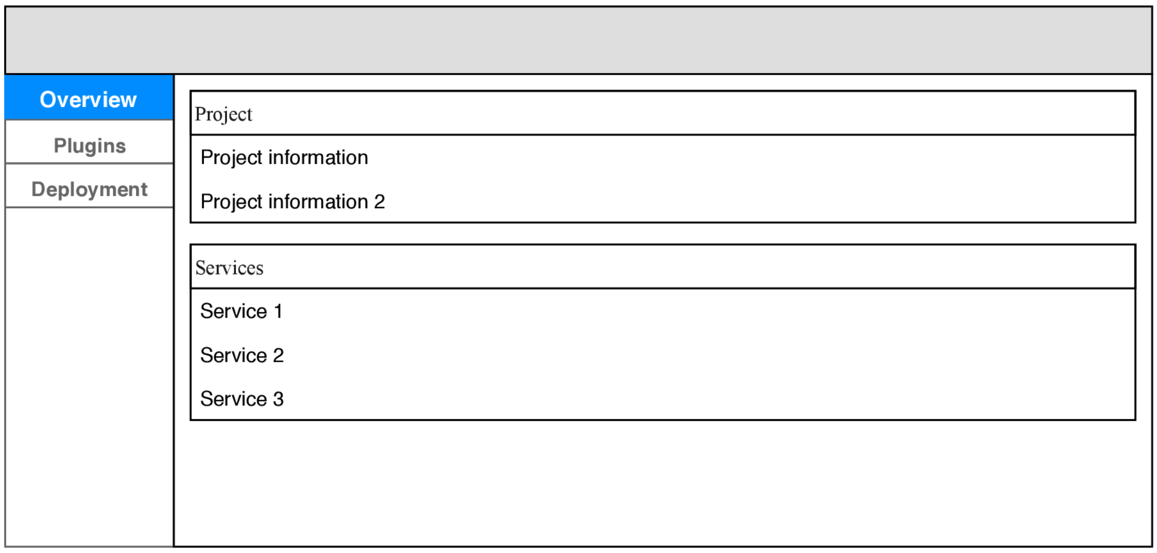

Figure 5.2: Wireframe of the Overview Component. Grey rectangle on the top side of wireframe is *Material Toolbar*. Menu on the left side of wireframe is *Material Navigation Drawer*. Overview component has two *Material Cards*. One card contains information about *MMO* project. Second card contains list of *MMO* services with hyperactive link to detail of the service.

associated with that service. Again, list of MMO plug-ins is in the separate *Material Card* and list of Kubernetes resources is in another separate *Material Card* as seen in Figure 5.3.

Each Kubernetes resource has one row in the table. First column of the row is check box for selecting resources, second column is the type of resource, third and fourth are buttons for editing and deleting Kubernetes resource. When “Edit” button is clicked, dialog window opens with the content of the resource will be opened as seen in Figure 5.4.

On the bottom of the card, there are two action buttons – for creating new resource for the service and second for deploying all resources that are selected. When button for creating new resource is clicked, dialog window with form is shown. This window is shown in Figure 5.5. It contains inputs needed for generating valid Kubernetes configuration for deployment of application:

- Name of the resource (service).
- List of the ports that service uses.
- List of the environment variables that service needs.
- List of the volumes that have to be mounted to container.

Each list of items has button that appends new empty row on the bottom of the list. “Add volume” button is little different – dropdown list of options is shown when clicked. Each option appends different type of row. First option adds “Persistent Volume Claim” volume, second option adds “GCE Disk” volume. Field values will be filled with the information that can get be fetched from the API. These values can be determined from the MMO service plug-ins – e.g. some plug-in is used for generation of the REST API that runs on the port number 50080, so we can add this port to the form.

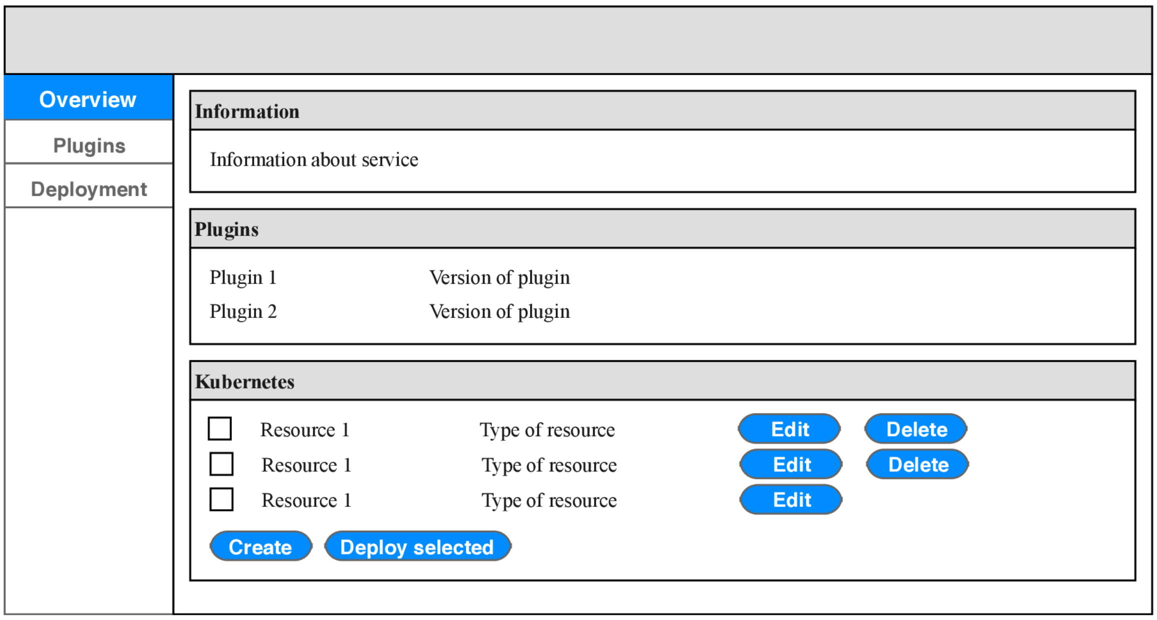

Figure 5.3: Wireframe of Service Detail Component. Component is divided to three *Material Cards*. First contains information about service. Second contains list of *MMO* plug-ins that service uses and last one has list of Kubernetes resources with ability to select, edit or delete them. Selected resources can be deployed or new resource can be generated with two buttons at the bottom of the last card.

Two buttons are on the bottom of the dialog's window – for creating Kubernetes resource and for canceling generation of the resource (this will close dialog).

5.2.3 Design of the Plug-ins Component

Plug-ins Component is the most simple component among main components. Component has *Material Card*, which contains list of the global *MMO* plug-ins. This is shown on the wireframe in Figure 5.6. This component will be later extended with plug-in management (this is not goal of this thesis).

5.2.4 Design of the Deployment Component

Deployment component will be used for deploying application. Component (shown in Figure 5.7) is divided to two sections, each in the standalone *Material Card*. First section is for deploying using GitHub Deployments which are described in Section 5.1.2. As mentioned there, we need GitHub reference, deployment subject (form of message that can be later used, for example, reason for deployment) and environment.

Second part is used for manual deployment to Kubernetes cluser. As mentioned in Section 5.1.2, we need target namespace, cluster and source environment for manual deployment. Source environment determines which set of Kubernetes resources will be used in the project. When user clicks on "Deploy" button, confirmation dialog with summary information will be shown. Summary information contains list of deployed resources, name of the Kubernetes cluster and namespace. User have to confirm deployment. After confir-

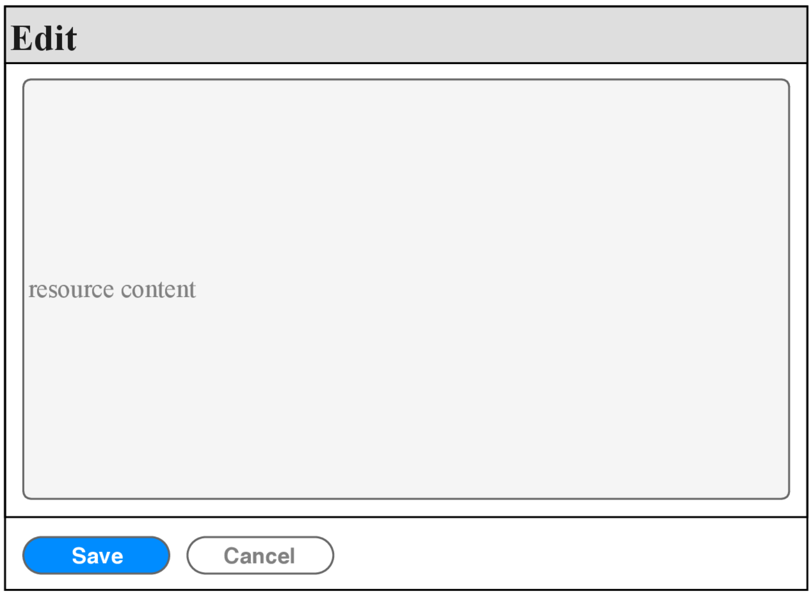

Figure 5.4: Wireframe of the dialog window for editing Kubernetes resources.

mation, output log will be shown in the dialog window to inform user about the deployment status.

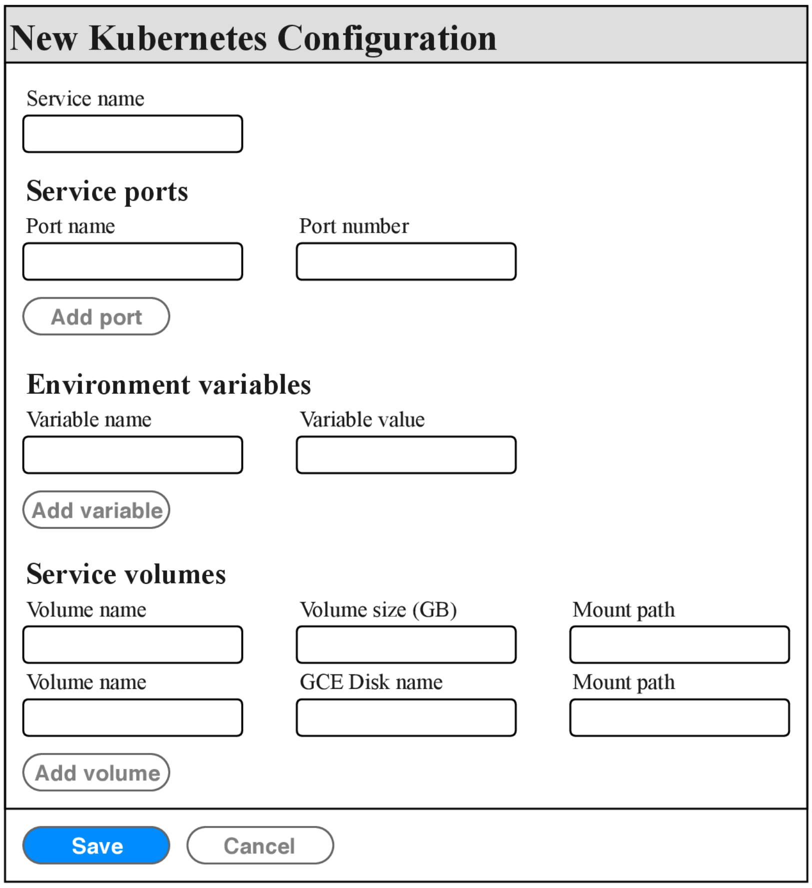

Figure 5.5: Wireframe of the dialog window for generating Kubernetes resources.

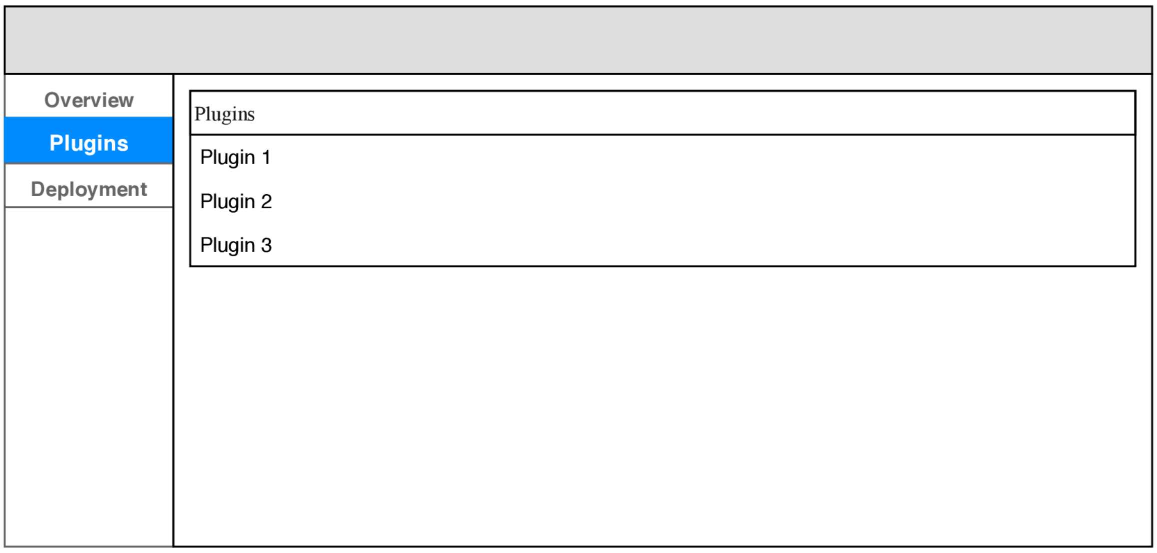

Figure 5.6: Wireframe of the Plug-ins Component. For now, just list of global *MMO* plug-ins is shown in the card. Plug-in management can be added later.

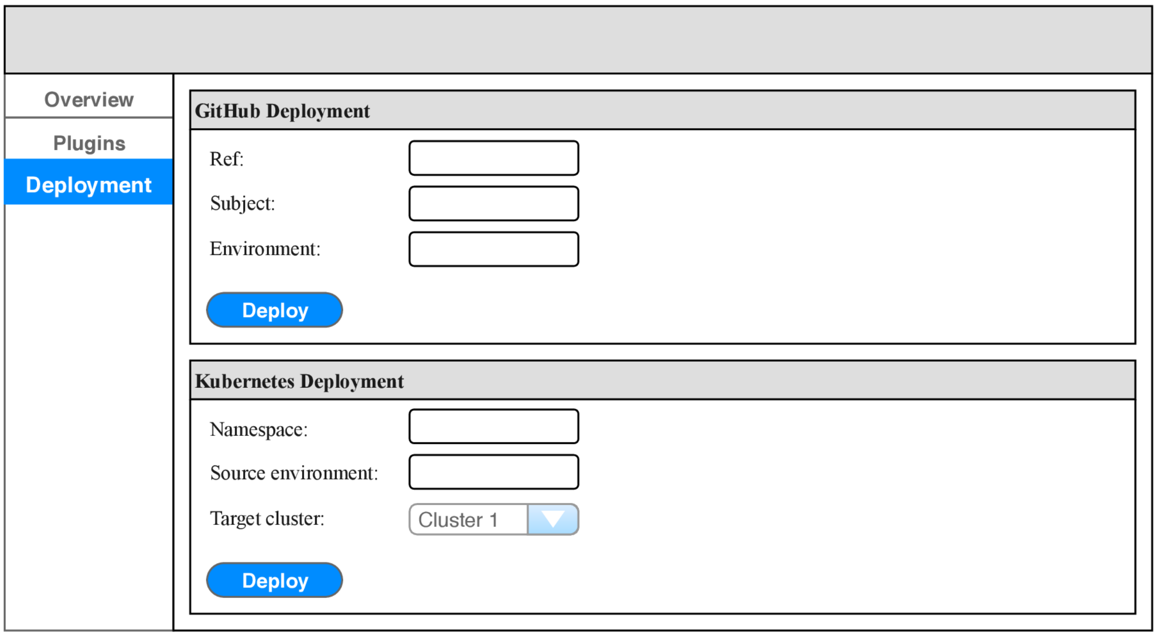

Figure 5.7: Wireframe of Deployment Component. One section is used for deploying application using GitHub deployments. Second section is used for deploying directly to Kubernetes.

Chapter 6

Implementation

This section talks about the implementation of the application. The structure of the source code created within this thesis is shown in Figure 6.1.

```
mmo ........................................................ Root of the repository
 └─ api .................................................... API part of the MMO
    ├─ protobuf
    │  └─ proto.proto .......................... Protobuf definition of gRPC server
    ├─ server
    │  └─ server.go ........................................ Server library of API
    ├─ static .................................. Directory with static frontend files
    ├─ proto.pb.go ............................. Generated gRPC server and stub
    ├─ proto.pb.gw.go .......................... Generated gRPC proxy gateway
    ├─ service_test.go ......................... Test of the service
    └─ service.go .............................. Implementation of server's service
 └─ ui ..................................... User interface of MMO – Angular project
    ├─ api .............. Typescript API client generated from the Swagger definition
    │  ├─ models ........................................... Generated models
    │  │  └─ ...
    │  ├─ api-client-service.ts ............................ Generated methods
    │  └─ index.ts
    └─ src ................................................. Angular source code
       ├─ app .............................................. Main Angular component
       ├─ assets ..................... Assets of Angular project – pictures, fonts, etc
       └─ environments ............................. Environments of Angular project
 └─ templates
    └─ kubernetes ............... Templates used for generating Kubernetes resources
       └─ ...
```

Figure 6.1: Directory tree of the *MMO* source code created within this thesis.

6.1 Implementation of the Backend

The backend part of the application is implemented in the programming language Go and it is present in the folder "api" in the root of repository. The structure of the source code is presented in Figure 6.1. File proto.proto in protobuf directory contains the definition of the server and its methods. This file is used for generating Go gRPC server, Go gRPC gateway and Swagger definition.

Backend of the application is run using command mmo ui. This also opens web browser with URL of frontend page. When server is run, configuration of the *MMO* project is loaded. This is later used for some server's methods like reading global plug-ins, services, its description and plug-ins. Along with configuration, GitHub personal key is loaded from environment variable $GITHUB_TOKEN too. GitHub key is used for integration of GitHub, in our case, Deployment API will be used for deploying applications.

Figure 6.1 shows how is source code of the *MMO* structured within repository. Directory api contains all files needed for running backend part of the application. Inside that, directory protobuf with file proto.proto is present. This file is used for generation of gRPC server and gRPC gateway proxy. Generated files are in the api directory and they are named proto.pb.go and proto.pb.gw.go. Generated gRPC server file has interface of the server that has to be implemented. Implementation of the mentioned sever is in the file service.go. This is shown in Figure 6.2.

Last file has to be mentioned is file server.go in the directory server. This file contains function that runs both gRPC server and HTTP to gRPC proxy.

Kubernetes resources are generated using Go HTML templates present in the Go standard library[1]. Example of the Go template can be seen in Listing 6.1. Go template engine takes template and an object of type interface{} as an input. Example of object can be seen in Listing 6.2. Resulting file created from mentioned template and object can be seen in Listing 6.3.

```
kind: Service
apiVersion: v1
metadata:
  name: {{ .ServiceName }}
spec:
  selector:
    app: {{ .ServiceName }}
  ports:{{range $index, $element := .Ports}}
    - name: {{ $element.Name }}
     port: {{ $element.Port }}
     protocol: TCP {{end}}
```

Listing 6.1: Go template used for generation of Kubernetes Service resource

```
{
    ServiceName: "auth",
    Ports: [
        {
            Name: "http",
            Port: "80",
        },{
            Name: "grpc",
            Port: "50051"
        }
```

[1]https://golang.org/pkg/text/template/

```
    ]
}
```

Listing 6.2: Object that can be passed to the Go template engine

```yaml
kind: Service
apiVersion: v1
metadata:
    name: auth
spec:
    selector:
    app: auth
    ports:
    - name: http
     port: 80
     protocol: TCP
    - name: grpc
     port: 50051
     protocol: TCP
```

Listing 6.3: Result of filling template in Listing 6.1 with object in Listing 6.2.

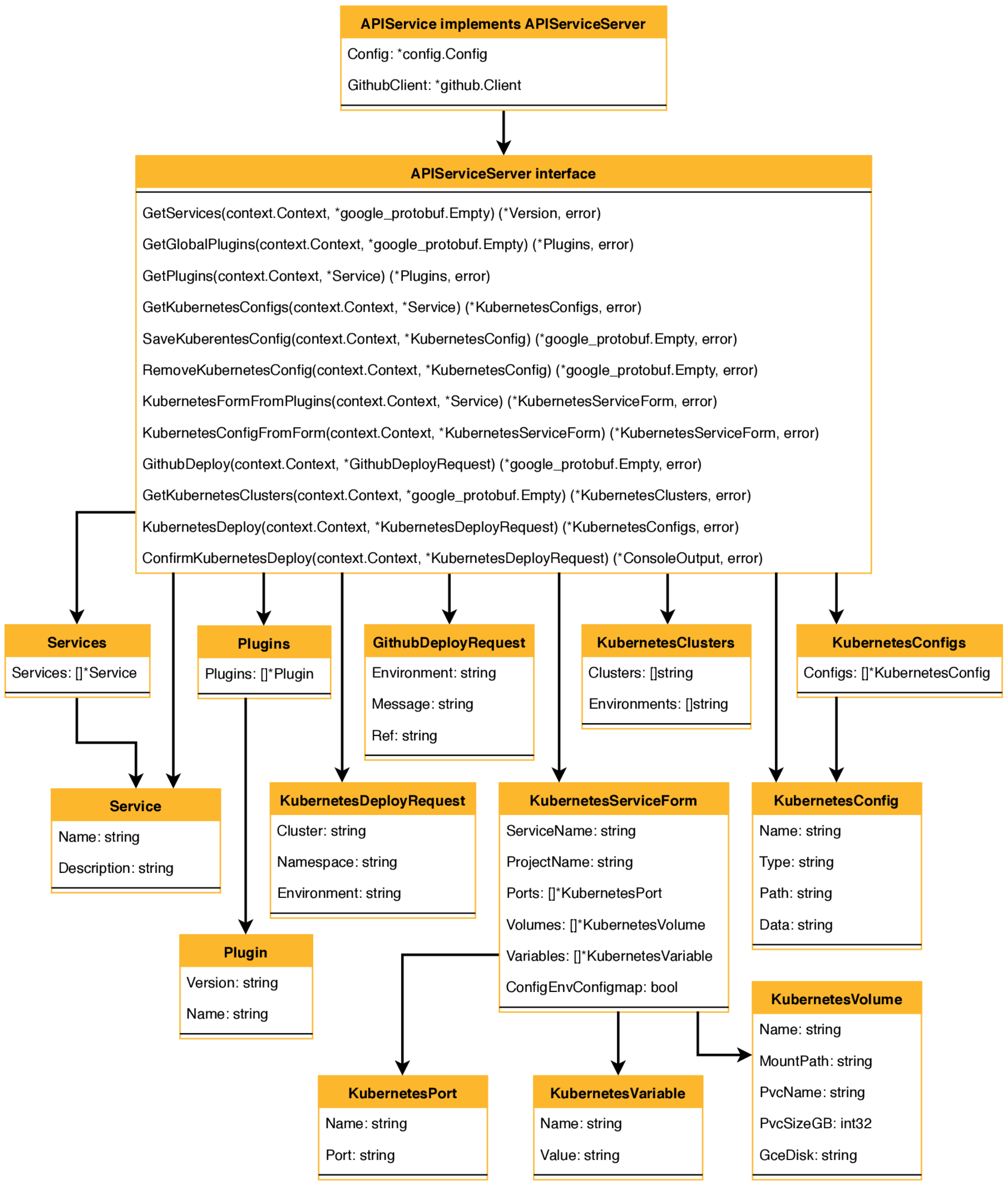
APIService implements APIServiceServer
Config: *config.Config
GithubClient: *github.Client
APIServiceServer interface
GetServices(context.Context, *google_protobuf.Empty) (*Version, error)
GetGlobalPlugins(context.Context, *google_protobuf.Empty) (*Plugins, error)
GetPlugins(context.Context, *Service) (*Plugins, error)
GetKubernetesConfigs(context.Context, *Service) (*KubernetesConfigs, error)
SaveKuberentesConfig(context.Context, *KubernetesConfig) (*google_protobuf.Empty, error)
RemoveKubernetesConfig(context.Context, *KubernetesConfig) (*google_protobuf.Empty, error)
KubernetesFormFromPlugins(context.Context, *Service) (*KubernetesServiceForm, error)
KubernetesConfigFromForm(context.Context, *KubernetesServiceForm) (*KubernetesServiceForm, error)
GithubDeploy(context.Context, *GithubDeployRequest) (*google_protobuf.Empty, error)
GetKubernetesClusters(context.Context, *google_protobuf.Empty) (*KubernetesClusters, error)
KubernetesDeploy(context.Context, *KubernetesDeployRequest) (*KubernetesConfigs, error)
ConfirmKubernetesDeploy(context.Context, *KubernetesDeployRequest) (*ConsoleOutput, error)
Services
Services: []*Service
Plugins
Plugins: []*Plugin
GithubDeployRequest
Environment: string
Message: string
Ref: string
KubernetesClusters
Clusters: []string
Environments: []string
KubernetesConfigs
Configs: []*KubernetesConfig
Service
Name: string
Description: string
KubernetesDeployRequest
Cluster: string
Namespace: string
Environment: string
KubernetesServiceForm
ServiceName: string
ProjectName: string
Ports: []*KubernetesPort
Volumes: []*KubernetesVolume
Variables: []*KubernetesVariable
ConfigEnvConfigmap: bool
KubernetesConfig
Name: string
Type: string
Path: string
Data: string
Plugin
Version: string
Name: string
KubernetesVolume
Name: string
MountPath: string
PvcName: string
PvcSizeGB: int32
GceDisk: string
KubernetesPort
Name: string
Port: string
KubernetesVariable
Name: string
Value: string

Figure 6.2: Diagram of structures and dependencies of Go package `github.com/flowup/mmo/api`. Every structure except `APIService` is generated from the models in Protobuf definition. Interface `ApiServiceServer` is generated from RPC methods in Protobuf definition too.

6.2 Implementation of the Frontend

Frontend is implemented in programming language Typescript using web application framework Angular. Like mentioned in 5.2, Angular is using reusable components for building application. Source code of the frontend is stored in the directory `ui` of the root of *MMO* repository as shown in Figure 6.1. This directory contains directory `api` which contains models generated from Swagger definition. Swagger definition is stored in the root of *MMO* repository and it is generated from the Protobuf definition. Second directory that can be mentioned is `src` which contains all source code related to Angular project – components and dialogs. At the root of `src` directory is directory `app`, which is main component of the application.

Library *Angular Material*[2] is used for integrating *Material Design* into the application.

App Component has *Material Toolbar*, *Material Navigation Drawer* and component based on the route that user has visited. This can be Overview Component, Deployment Component or Plugins Component. Structure of components within the `app` component can be seen in Figure 6.3.

```
app
├─ overview ........................................ Overview Component
│  └─ ...
├─ deployment ...................................... Deployment Component
│  └─ ...
├─ service ......................................... Service Detail Component
│  ├─ kubernetes ........... Dialogs for generating and editing Kubernetes resources
│  └─ ...
├─ global-plugins .................................. Global Plug-ins Component
│  └─ ...
├─ store .............. State management of the application using ngrx store library
│  └─ ...
└─ ...
```

Figure 6.3: Source code of the Angular project's components. Tree starts with main component of application named `app`.

A library ngrx-store is used for managing the state of application[3]. How this library works is described in Figure 6.4. Ngrx effects, reducers and store are all stored in `store` directory of `app` component.

Each component consists of three files:

- HTML template
- Sass style file
- Typescript Component file

HTML template is used for defining how component should look like. With special directives, variables from the Typescript file can be used. HTML template also supports more advanced features like conditions and loops. Style of HTML file is defined in Sass

[2] http://material.angular.io

[3] https://github.com/ngrx/platform

file. Sass is an extension of CSS that support features like variables, nesting of the CSS attributes, etc[4]. Typescript Component file is used for programming background logic of the component, for example, accessing ngrx store and retrieving data from it.

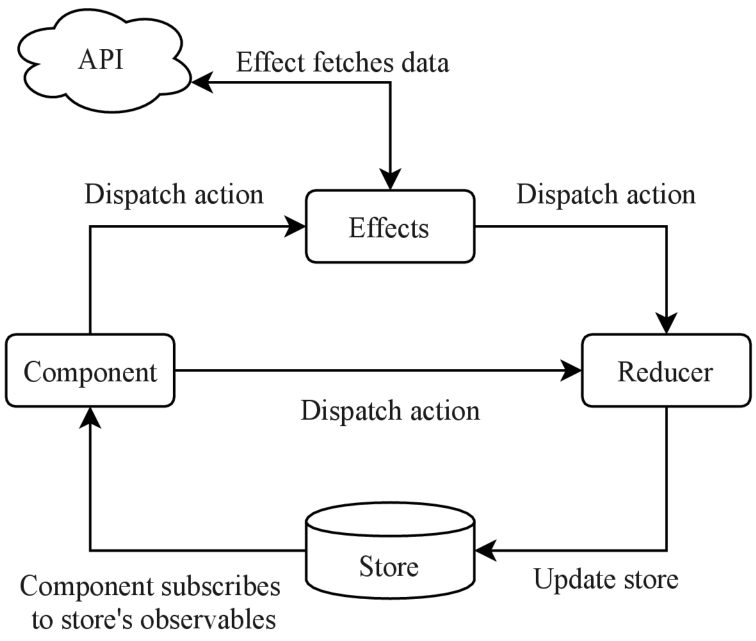

Figure 6.4: ngrx state management is based on actions. Actions can also transport a payload. Component is reading data from Store and can dispatch action that can affect the state of Store. Effects are side effects that can be triggered by action and effect, for example, can fetch data from API. Reducer is component that catches all actions and decides how to change state of the Store based on the action.

[4] https://sass-lang.com/documentation/file.SASS_REFERENCE.html

Chapter 7

Experiments

7.1 Time savings

Time savings that come from using user interface can be divided to two groups: time savings from Kubernetes resource management and time savings from the deployment of application from the user interface.

7.1.1 Time savings from Kubernetes resource management

Resources are usually copied from the examples of Kubernetes documentation or from different projects and then edited for intended purpose. Time can save be saved by generating resources with *MMO*. Time savings will differ for each user. Users who have experience with Kubernetes will save less time than unexperienced users. Another advantage of using *MMO* generator is that generator uses tested templates so there will be no errors after generation. User can bring some errors to resources when resources are copied from the documentation or from the project.

We have two user test groups for this experiment:

1. Beginner user – user knows the basics of Kubernetes. He can create Kubernetes resources based on Kubernetes documentation but he cannot check that resource is correct. This user will have hard time when something in the resources is not correct and he has to debug it.

2. Experienced user – user can create basic resources like Deployments and Services. He can check that resource he created is correct (Ports, container image, volume mounting, environment variables, etc.).

A task of the tested user is to create Kubernetes resource that contains Deployment and Service. Application is using two ports – one for exposing gRPC server and second one for exposing HTTP API. Application is storing data in the `/opt/data` and GCE disk (Google Cloud Compute Engine disk) has to be mounted to that directory. Environment variable named `DB_USER` has to be added with value `postgres`. Resource is created manually and second time it is generated using the MMO generator. Task was marked as failed, when user deployed three non-working configurations to Kubernetes.

Each testing group had three users. As shown in Figure 7.1, experienced user needed averagely 4m 5s to manually create the resource. 1 user had typo in the resource that was fixed after first deploy. Generation of the resource taken 1m 14s in average to the experienced users and resources created in this way are all correct. Beginner users had similar results when they generated the resources – difference was only little higher average time. Beginner users had problem to create Kubernetes resources from scratch – average time was over 17 min 49 sec but none of the created resources was correct.

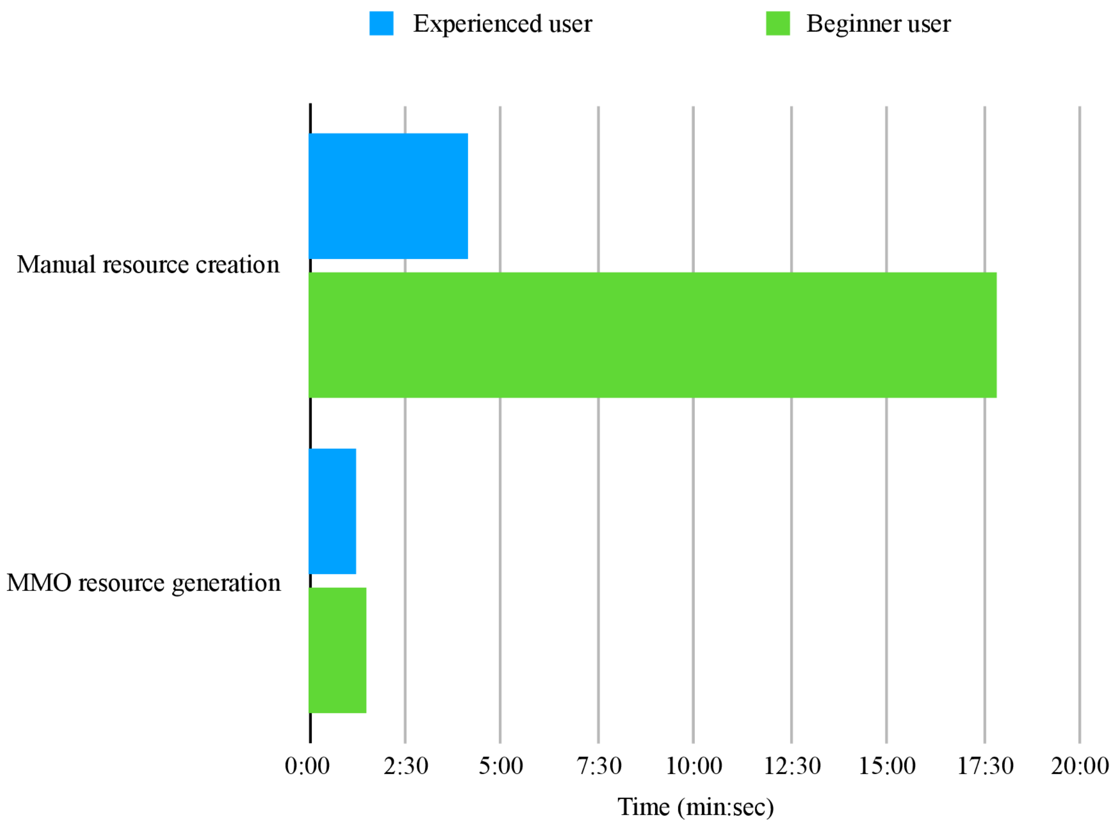

Figure 7.1: Chart showing the results of Kubernetes resources creation experiment

In the result, experienced Kubernetes users are able to generate Kubernetes resources approximately 226 % faster. Beginner users are able to generate configurations that are working out of the box so they much of their time and time of the DevOps engineers.

7.1.2 Time savings from using the MMO deployments

Time savings from using the *MMO* deployments come mainly from elimination of the mistakes that can be done when the application is deployed. Mistakes can happen easily when deploying application cluster. Mistakes are more likely when multiple applications are deployed to the different clusters and to different namespaces of the cluster. A result of the survey in 3.7 says that the most made mistake is deploying to the different namespace. Namespace must be provided via flag in `kubectl` tool and `default` namespace is selected when it is not provided. In other cases, namespace may be mistyped or namespace of another project can be typed by not paying attention. Mistake like this is not critical when nothing is running in the targeted namespace. This can be critical when existing application in mistaken namespace is put down by deploying different resources. This mistake can cause finance losses when production is put down. Similar mistake can happen when application is deployed to the different cluster. This is very likely when user is switching between different clusters and he forgets to switch to cluster where he wants to deploy application.

All these mistakes costs the user money or time. When no application is put down by deploying in the wrong environment, we always have to remove deployed resource from the environment. This can take various time depending on the status of environment. It takes less time to fix deploy mistake in namespace or cluster where nothing is running. On the other side, it may take long time to fix deploy mistake when application is deployed in the namespace where another application is running. User must be careful that he will not remove resources from original applications, because this can make application work improperly.

7.2 Standalone Platform for Deploying Microservices Projects

This experiment consisted of building a prototype of a standalone platform for managing microservice applications. The platform should be used for deploying applications after they are builr and testet in a continuous integration tool. The platform should serve to end-users, especially to developers and DevOps engineers.

The platform should be able manage projects and services. After creating the application you can create service and configure each service. Configuration of the service consists of similar options as those mentioned in Design section 5.1.1. We need information about which ports are used by application, if an application uses some volumes to persist data, which variables should be exported into the container, and which configurations should be mounted into the container.

The platform should have a webhook endpoint that would be accessible to continuous integration tools to deploy application into the cluster. Use-cases of the platform:

1. User signs in and creates project.
2. Services are added to project.
3. Each service is configured.
4. User configures cluster where application will run (Kubernetes cluster or Docker Swarm cluster) and provides credentials to this cluster.
5. User will setup continuous integration system to notify the platform that some service from the project should be deployed and information about changes is provided (can be name of the Docker image).
6. Platform will deploy application after notification.

Application will use web application framework Angular for a frontend and Firebase for a backend. Firebase is suitable for prototyping applications like this because of its simplicity and number of features it provides. The most important feature that we will use is Firestore which is real-time document database. The database will be used to store data about users, their projects, and services that are in the project. Configurations of all services will be also stored in Firestore. Storage in Firebase will be used for storing configurations that will be deployed as ConfigMaps to Kubernetes. We will also use Firebase Authentification for authenticating and authorizing users. Users will be able to sign-in using their GitHub account. Each user should have access only to projects and services that he owns. Firebase hosting will be used for hosting frontend of our platform. Last part of the Firebase that will be used is Firebase functions. Functions serve as support functions for Firestore database. We can create functions that will watch some resources in the database and trigger them when some event happens. For example, this type of the event can be write to the database. Firebase Functions should be used always when we have to do some operation that is CPU or memory intensive, or when the operation should not be done on the application's frontend. The technologies used in this implementation are shown in Figure 7.2.

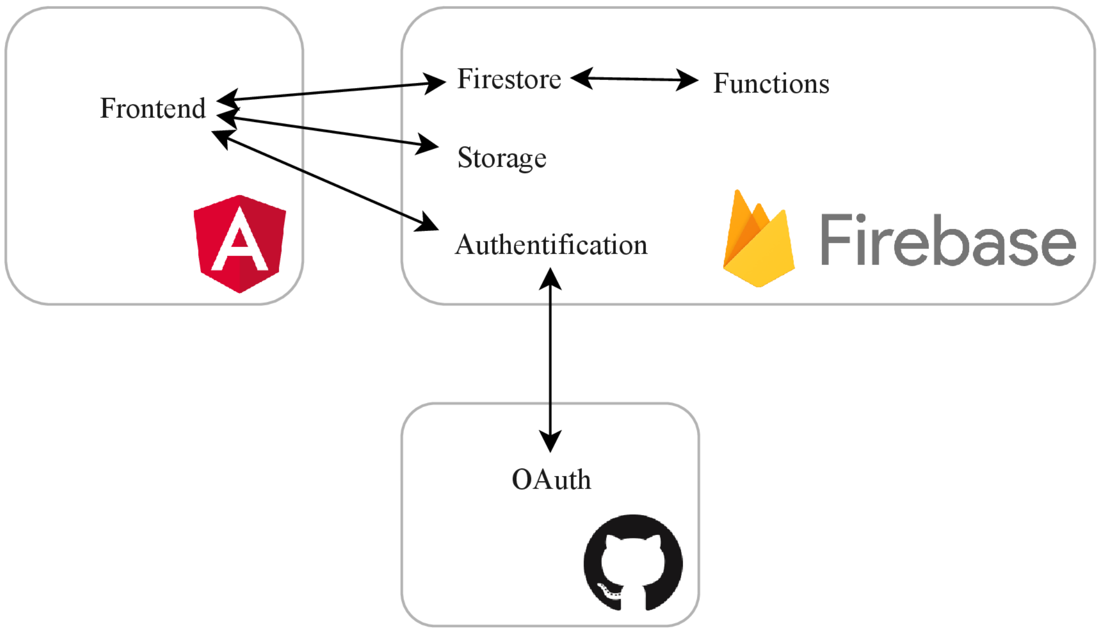

Figure 7.2: Application frontend is implemented using web application framework Angular. Users can authenticate with their GitHub account using Firebase Authentification. Data of the users are stored in the Firestore and Storage of Firebase. Firebase Functions are supportive functions for Firestore database.

Each service has multiple settings – used ports, mounted volumes, exported environment variables and used configurations. These settings are stored in Firebase Firestore database. Configurations are uploaded to the Firebase Storage. For deployment, user needs API key which is used for authentication and authorization. User can generate API key for his application in project settings. Webhook endpoint is implemented using Firebase Functions. When user sends deploy request to this endpoint, function will validate his API key and append this deploy request to Firestore. There is another Firebase Function that watches these deploy requests and does deployments according these requests. The deployment of application is done over Kubernetes API. Kubernetes resources are created based on the configuration of the service that should be deployed and information from the deploy request – Docker image name. These resources are sent to Kubernetes API with the credentials that are stored in the Firestore.

The application was implemented according to the design above. After doing user testing, it has been found out that experience the platform provides is not sufficient. Exact problems are:

- The application is just another platform that the developers and DevOps engineers would have to visit and have some basic knowledge about it.
- The application's feature that deploys application to the cluster on the webhook event should be part of the continuous integration pipeline.
- Configuration of the services is not tied to the version of the services. To make this possible, application's configuration would have to be stored in the repository instead of the platform's Firebase.

Platform would be useful, but in the case that it would integrate also continuous integration and deployment tools so developers would use only one tool instead of the two. Platform would have to be more advanced. Advanced options are necessary so applications can be set up to run in different environments (production too). Platform should cover stuff like monitoring, scaling, logging and more. Platform with this many features is big project that is not realizable as a bachelor thesis. Development would need team of the people and lot of planning.

7.3 User Experience

User experience tests were done to find out user experience flaws in application. First round of the user tests revealed some serious bugs in the application. All of these bug were fixed.

The most serious bug was that deployed application was always deployed to the default namespace of the Kubernetes, no matter which namespace was selected in the user interface. The second bug was a problem with refreshing the user interface in the browser, which just shown error "404 - Not found". This problem was caused by Angular which uses path in the browser for application routing. When browser path contained something after slash, HTTP static file server returned error, because it had not found that file in the static files. Last bug appeared when user tried to generate Kubernetes resource. Form with service configuration was not filled because server crashed while it was reading *MMO* service configuration.

First round of the user tests also bring enhancement to generated Kubernetes resources. Enhancement is adding healthcheck sidecar container to the Deployment of the gRPC server. Kubernetes offers native healthchecking of the container, which is used for terminating containers that cannot respond because they are unhealthy[1].

Second round of user tests have brought the idea to generate Configmap environment mappings and use them as environment variables in the Deployments. Some users mentioned in survey in 3.7 that they prefer] this approach. Second enhancement was loading of the source environments in the Kubernetes Deployment section of the application. This will save to users, because they do not have to check available environment in the application's repository.

[1]https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/

Chapter 8

Conclusion

The result of this thesis is a extension of the tool *MMO* that helps developers to deploy applications to Kubernetes cluster. Extension of the tool is in the form of an user interface. The most beneficial feature that came with the extension is generation of Kubernetes configurations for the deployment of applications. It is saving the precious time of both beginner and advanced users of Kubernetes. Other feature is useful for deployment of application using GitHub deployments or by direct deploy to Kubernetes cluster. The advantage of this feature is reduction of mistakes done by Kubernetes users, where they deployed their application to a different Kubernetes namespace or a different Kubernetes cluster.

MMO user interface can show much more information in comparison with command line interface. It can be extended with features that *MMO* provides as a command line interface tool. It could help the developer to focus his attention to the service that he is developing. Whole new features are possible with user interface. For example, one of the features can be running a service in a development mode, where source code of the service is tested and the results of the tests are shown in the user interface. Another useful feature would be watching source code and deploy application to Kubernetes cluster upon change. This would be useful when *MMO* would integrate the tool *skaffold* mentioned in Chapter 4. This is not possible at the moment because *skaffold* is a new tool in early stages of development and 4 alpha releases of the tool already rolled out between the first release and finishing this thesis. All of these changes would need redesign of the *MMO* tool as a whole to provide the best user experience to developers.

Bibliography

[1] Anonymous: *Protocol buffers*. [Online; visited 24/04/2018].
Retrieved from: https://developers.google.com/protocol-buffers/

[2] Anonymous: *The Open Container Initiative*. [Online; visited 03/13/2018].
Retrieved from: https://www.opencontainers.org/about

[3] Dua, R.; Raja, A. R.; Kakadia, D.: Virtualization vs Containerization to Support PaaS. In *2014 IEEE International Conference on Cloud Engineering*. March 2014. pp. 610–614. doi:10.1109/IC2E.2014.41.

[4] Fowler, M.; Foemmel, M.: Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*. vol. 122. 2006: page 14.

[5] Humble, J.; Farley, D.: *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Publishing Company. 2010. ISBN 0-321-60191-2.

[6] Kubernetes: *Kubernetes*. [Online; visited 04/10/2018].
Retrieved from: https://kubernetes.io

[7] Nelson, B. J.: *Remote Procedure Call*. PhD. Thesis. Pittsburgh, PA, USA. 1981. aAI8204168.

[8] Newman, S.: *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media. 2015. ISBN 978-1491950357.

[9] Rachel Potvin, J. L.: Why Google Stores Billions of Lines of Code in a Single Repository. *Communications of the ACM*. vol. 59, no. 7. 7 2016: pp. 78–87.

[10] Red Hat, I.: *Ansible documentation*. [Online; visited 04/02/2018].
Retrieved from: http://docs.ansible.com

Appendices

Appendix A

The Content of the Included Memory Media

A memory media attached to this thesis contains following files and directories:

- **mmo/** – a source code of tool *MMO* (parts that were designed and implemented within this thesis are listed in Figure 6.1).
- **mmo-demo/** – a *MMO* project for demonstrating functionality implemented in this thesis.
- **platform/** – a source code of platform that was created as an experiment in this thesis.
- **tex/** – LaTeX source code of this thesis.
- **README.md** – a file that contains instructions for building and running tool *MMO*.
- **thesis-print.pdf** – a final version of this document for printing.
- **thesis-wis.pdf** – a final version of this document for submitting to WIS.