TECHNISCHE
HOCHSCHULE
LÜBECK

Bachelor-Thesis

A Self-service User-interface for Creating Gitlab-projects with Kubernetes Integration

Vorgelegt von: Tom Christopher Böttger

Fachbereich: Elektrotechnik und Informatik

Studiengang: Informatik / Softwaretechnik

Erstprüfer: Professor Dr. Kratzke

Ausgabedatum: 02. November 2020
Abgabedatum: 02. Februar 2021

A. Hanemann
(Professor Dr. Andreas Hanemann)
Vorsitzender des Prüfungsausschusses

Aufgabenstellung:

As part of its computer science teaching and other (research) projects, the myLab at the TH Lübeck (THL) uses both GitLab and Kubernetes. The intention is to establish a Kubernetes cluster as a flexible shared hosting platform for research and student projects of any kind.

However, Kubernetes has the disadvantage of a very steep learning curve. Therefore, this service has not been widely used by students. Due to its limited usability, it is mostly used "behind the scenes" at the THL.

This bachelor thesis will address this problem and develop a "scaffolding" solution that generates preconfigured Kubernetes namespaces and example GitLab projects (Git repositories) that are built and deployed automatically using GitLab CI pipelines. These projects can serve as a starting point for further project work and should flatten the Kubernetes and CI pipeline learning curve.

- The scaffolding solution (in the following referred to as Scaffolder) shall be distributed and configured via Helm 3 and then deployed as an application in any Kubernetes cluster.
- Users should authenticate to the Scaffolder by using the OAuth 2.0 interface integrated into GitLab (The OAuth 2.0 interface of the THL GitLab instance should be used as a reference).
- The Scaffolder is to be designed stateless. It should not maintain a state over the namespaces and related user projects and therefore, cannot enforce limits on the number of projects per user.
- The Scaffolder is intended to create Gitlab repositories on behalf of the authenticated user. These repositories should contain:
  - Access credentials to the created namespace in the Kubernetes Cluster
  - A configured GitLab CI pipeline example (.gitlab-ci.yml) that allows automated builds and deployments
  - Deployment examples (to be selected by users via the Scaffolder interface) covering typical Kubernetes workloads (Deployment, StatefulSet, CronJob, Job), common languages (e.g. Java, Python, NodeJS, etc.), and possibly standard stateful services (e.g.~Redis, MySQL, etc.) in such a way that a user can quickly configure his source template (e.g. Python + Redis)
  - Exposing examples (also selectable by users) should cover Kubernetes Service and Ingress resources.
- The Scaffolder should create Kubernetes namespaces, provided that the particular namespace is still available. The namespaces must be configured as follows:
  - Resource quotas for CPU, memory, and storage for namespaces must be specified (values must be configurable)
  - Limit ranges for containers must be specified (values must be configurable)
  - The namespace should include a service account.
  - This service account should be bound to a read/write role that allows the service account to manipulate and view only resources within its namespace.

A suitable architecture must be developed for the Scaffolder. A setup composed of a single page web app and a REST-based backend is preferable. The Scaffolder (web app) should be easy and intuitive to use! Simplicity comes before functionality!

Professor Dr. Kratzke

TECHNISCHE
HOCHSCHULE
LÜBECK

Eigenständigkeitserklärung

Declaration of Originality

Böttger, Tom Christopher

Name, Vorname
Last name, first name

302177

Matrikelnummer
Matriculation number

Ich versichere hiermit, dass ich die vorliegende
I hereby declare that this

□ **Hausarbeit** term paper ☒ **Bachelorarbeit** bachelor's thesis □ **Masterarbeit** master's thesis

mit dem Titel
with the title

A Self-Service User Interface for Creating GitLab Projects with Kubernetes Integration.

eigenständig und ohne unerlaubte fremde Hilfe angefertigt habe. Nach den „Leitlinien zur Sicherung guter wissenschaftlicher Praxis" (Deutsche Forschungsgemeinschaft) habe ich keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und Entlehnungen aus anderen Arbeiten kenntlich gemacht. Für den Fall, dass die Arbeit zusätzlich elektronisch und/ oder digital eingereicht wird, erkläre ich, dass die schriftliche und die elektronische und/ oder digitale Form identisch sind. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

is my own original work and any assistance from third parties has been acknowledged. In keeping with the „Guidelines for Safeguarding Good Research Practice" (Deutsche Forschungsgemeinschaft/ German Research Foundation), I have clearly indicated and acknowledged all sources and resources as well as any borrowings from other works. In case of an additional electronic and/or digital submission of this work, I declare that the written form and the electronic and/or digital form are identical. This work has not previously been submitted either in the same or in a similar form to another examination office.

Ich bin damit einverstanden, dass die vorliegende Hausarbeit/ Bachelorarbeit/ Masterarbeit für Veröffentlichungen, Ausstellungen und Wettbewerbe des Fachbereiches verwendet und Dritten zur Einsichtnahme vorgelegt werden kann.

I agree that this work can be used for publishing, exhibition or competition purposes and can be inspected by third parties.

☒ **ja** Yes □ **nein** no □ **es liegt ein Sperrvermerk bis ______________ vor** there is an embargo period until

Lübeck, 14.01.2021

Ort, Datum
Place, Date

Tom Böttger

Unterschrift
Signature

TECHNISCHE
HOCHSCHULE
LÜBECK

Bachelor Thesis

A Self-Service User Interface for Creating GitLab Projects with Kubernetes Integration

Submitted by:	Tom Christopher Böttger
Department:	Electrical Engineering and Computer Science
Degree program:	Computer Science
First examiner:	Prof. Dr. Nane Kratzke
Date of Issue:	02th November 2020
Date of Submission:	14th January 2021

(Professor Dr. Andreas Hanemann)
Head of Examination Board

Task Definition

As part of its computer science teaching and other (research) projects, the myLab at the TH Lübeck (THL) uses both GitLab and Kubernetes. The intention is to establish a Kubernetes cluster as a flexible shared hosting platform for research and student projects of any kind.

However, Kubernetes has the disadvantage of a very steep learning curve. Therefore, this service has not been widely used by students so far. Due to its limited usability, it is mostly used "behind the scenes" at the THL.

This bachelor thesis will address this problem and develop a "scaffolding" solution that generates preconfigured Kubernetes namespaces and example GitLab projects (Git repositories) that are built and deployed automatically using GitLab CI pipelines. These projects can serve as a starting point for further project work and should flatten the Kubernetes and CI pipeline learning curve.

- The scaffolding solution (in the following referred to as Scaffolder) shall be distributed and configured via Helm 3 and then deployed as an application in any Kubernetes cluster.
- Users should authenticate to the Scaffolder by using the OAuth 2.0 interface integrated into GitLab (The OAuth 2.0 interface of the THL GitLab instance should be used as a reference).
- The Scaffolder is to be designed stateless. It should not maintain a state over the namespaces and related user projects and therefore, cannot enforce limits on the number of projects per user.
- The Scaffolder is intended to create GitLab repositories on behalf of the authenticated user. These repositories should contain:
  - Access credentials to the created namespace in the Kubernetes cluster
  - A configured GitLab CI pipeline example (.gitlab-ci.yml) that allows automated builds and deployments
  - Deployment examples (to be selected by users via the Scaffolder interface) covering typical Kubernetes workloads (Deployment, StatefulSet, CronJob, Job), common languages (e.g. Java, Python, NodeJS, etc.), and possibly standard stateful services (e.g. Redis, MySQL, etc.) in such a way that a user can quickly configure his source template (e.g. Python + Redis)

    - Exposing examples (also selectable by users) should cover Kubernetes Service and Ingress resources

- The Scaffolder should create Kubernetes namespaces, provided that the particular namespace is still available. The namespaces must be configured as follows:
    - Resource quotas for CPU, memory, and storage for namespaces must be specified (values must be configurable)
    - Limit ranges for containers must be specified (values must be configurable)
    - The namespace should include a service account
    - The service account should be bound to a read/write role that allows the service account to modify and view only resources within its namespace

A suitable architecture must be developed for the Scaffolder. A setup composed of a single page web app and a REST-based backend is preferable. The Scaffolder (web app) should be easy and intuitive to use! Simplicity comes before functionality!

Contents

1 Introduction

For a practice-oriented computer science education, it is not enough for students to develop and run applications only locally on their computers. That would not meet the demands of today's trends, in which an ever-increasing use of public clouds, service-oriented architectures, and standardized, open deployment methods play a role [1].

It is necessary to allow students to deal with current technologies and to enable the use of them with as little effort as possible during their studies and research. In addition to integrating these technologies into the curriculum, the first step is to create organization-wide service offerings within a datacenter.

For this purpose, the Technical University of Applied Sciences Lübeck established the myLab laboratory [2]. The aim is to offer teachers, scientists, and students managed services for any task, be it a research project, courses, theses, or even private projects. The offerings include virtual machines, JupyterLab (for interactive document-based computing), GitLab (for Git repositories, project management, and continuous integration and deployment), and Codepad (for quick creation and sharing of code snippets). Compared to public clouds, all services are free of charge and mentored by teachers and laboratory engineers to lower the hurdle.

However, a service for efficient development and hosting of web and cloud-native applications is still missing. In this context, efficient means that the service is, at best, well integrated into the existing GitLab service, does not waste unnecessary resources, and is easy to manage. Therefore, the offering of virtual machines is not adequate. In contrast, container orchestrators such as Kubernetes are already used internally as a flexible hosting environment for applications – but there is no widely used offering for this yet.

Two main reasons are considered responsible for the moderate use of Kubernetes in university settings:

- On the one hand, applications running in containers and the associated container orchestration introduce new technologies and principles. These are very different from the environment in which students usually develop and run their applications. Besides the actual development of an application, this leads to a considerable additional effort.

Furthermore, students have to take new concepts of service orientation into account when developing applications. The steep learning curve may consequently discourage many.

- And on the other hand, there is no self-service offering for creating Kubernetes environments and projects yet. The reason for this is that no available solution meets the individual requirements of the university setting. Example challenges are the multi-tenancy capability and the associated restrictions and limits. The service would have to be well integrated into existing services such as GitLab while introducing as few new concepts as possible to flatten the learning curve.

1.1 Motivation

The two summarised issues provide the incentive to work on this bachelor thesis and to develop a solution. Nevertheless, the reasons mentioned can be elaborated further and are by no means restricted to the university setting. Based on three example organizations, the following use cases explain how they could benefit from a solution.

The argumentation starts with the most apparent organization – the university. As technological change accelerates, so do the challenges facing teaching. The curriculum must adapt more frequently to new requirements of the world of work and research. To be able to guarantee practical relevance, it is essential to create service offerings for students as well as teachers and researchers. In the example of Cloud-Native Applications, the attention for this topic has been growing strongly since 2016 (see Fig. 1.1). The rapid growth is due to the emerging container-based approaches [1] and container orchestrators – including Kubernetes. Accordingly, it makes sense to familiarise students with these technologies and to promote their active use during their studies.

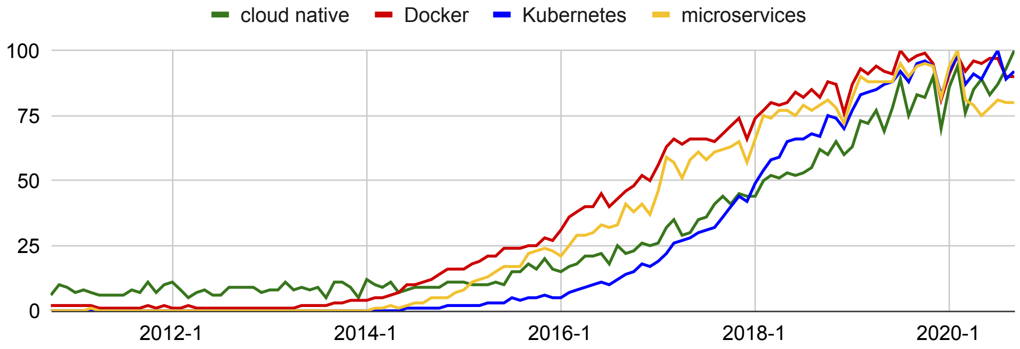


Figure 1.1: Search query trends for “cloud native”, “Docker”, “Kubernetes”, and “microservices” on Google over the last 10 years (10/2010 until 09/2020).

Another point is that new technologies have a relatively small community at the beginning. As a result, resources and tools aimed at beginners are not yet available. Furthermore, the integration into existing tools is often not yet or only insufficiently given. GitLab's Kubernetes integration can be mentioned as an example. The Git repository tool only allows the creation of Kubernetes clusters in the two public clouds of Google and Amazon or manual addition of an already created cluster. But advanced knowledge of Kubernetes is required to use this integration. Also, users must take care of the integration and deployment of the application themselves[1]. The simple creation of a project with Kubernetes integration, which uses existing organizational computing resources, is missing. These points lead to the fact that students may perceive the learning curve of applications running in containers as steep. Therefore, a transparent and well-integrated solution for generating projects can make sense.

A simple and free offer to host applications also encourages students to implement their ideas and projects outside of the courses. That enhances the learning effect even more by allowing students to pursue their interests and build a portfolio at the same time. The possible portfolio is especially helpful to graduates when applying for a job because this is often the only way to show that you have already gained experience [4] [5].

The following two example organizations are agencies and inhouse IT departments. Web development agencies, on the one hand, develop, manage, and maintain projects for many clients. Inhouse IT departments, on the other hand, do the same, but for its own applications and services, and on its own account. In both cases, they can benefit from an automated solution for creating Kubernetes based projects. A self-service offer would significantly reduce the configuration and administration effort. For example, Datadog already successfully uses an internal tool to provision Kubernetes environments to its developers automatically [6]. Developers could quickly create a project with Kubernetes integration and start development without administrative or operative overhead.

At the start of a project, the setup of all components usually takes much time. For example, the IT administrator must set up and configure the Kubernetes cluster and hand over the credentials to the developers. The developers then have to initialize the git repository, including the gitignore file and environment variables. Also, continuous integration and deployment must be configured and tested with a blank application. When all this got prepared, the actual development can begin. A self-service offer that combines these steps and automatically integrates the artifacts into the repository helps at this critical point.

[1] GitLab offers the Auto DevOps feature that takes care of the configuration of continuous integration and deployments for users. However, the process is not transparent to users and thus does not promote learning and knowledge of the technologies used. Furthermore, Auto DevOps requires a specific configuration of the cluster and the repository. Since this configuration is based on a cluster-admin role, multi-tenancy capabilities are limited [3].

But the generated project could contain even more. Developers could save time by selecting the desired technology stack in the self-service interface. The service then automatically generates a sample application for specifically the chosen technology stack. That not only lowers the learning curve for juniors but also allows automatic adjustment of the continuous integration and deployment to the specific technology stack.

Furthermore, Kubernetes offers the possibility of namespaces [7]. That means that administrators can divide clusters among several developer teams. For example, an agency can create a namespace for each client and an inhouse IT department can create a namespace for each internal application and service. The centralized management through namespaces instead of individual clusters allows organizations to monitor and control all resources more effectively. Additionally, clusters share internal resources between the namespaces, thus saving costs. Ideally, a self-service solution takes advantage of this namespace feature.

Last but not least, there is the advantage that many technologies related to the cloud-native stack are open and standardized [8] [9]. Developers can use the same technologies across various environments. They do not need to distinguish between a private datacenter in a university or company, and a public cloud. It makes working on this problem particularly appealing because it addresses a broad target group – which is likely to grow in the future. This also suggests the *Stack Overflow Developer Survey 2020*, in which Kubernetes and Docker as core technologies of the cloud-native stack landed in the top 3 of the most loved and wanted platforms [10].

1.2 Objectives

This bachelor thesis will address the problems and possibilities mentioned above. The goal is to develop a self-service interface that allows students to create new GitLab projects with Kubernetes integration. The generated projects shall be completely preconfigured. That means that the Git Repository contains a demonstration application, the Docker files, the Kubernetes manifest files, and the configuration for continuous deployment. The project should be customizable by the user, who can choose the desired technology stack during the creation process.

The creation of a simple, standalone project aims at flattening the steep learning curve of containerized applications. The focus should be on the essential technologies and, if possible, not introduce redundant convenience tools or concepts.

Furthermore, when generating a project, the service should automatically create a Kubernetes environment and integrate it into the project. To effectively distribute the available computing capacities, the service should make use of isolated namespaces. The isolation shall be solved using a role-based access system that moreover enforces defined resource contingents for each project.

1.3 Structure of This Thesis

In a first step, this work explains the term cloud-native and its related components in the *Fundamentals of Cloud-Native Applications* chapter. That includes containers as a runtime environment for applications and what container orchestration means. Furthermore, the concepts of continuous integration and deployment will get explained. These chapters thus provide the foundation for understanding this work and its results in detail.

Afterward, in the *Requirements Analysis* chapter, a stakeholder analysis will determine the requirements for a possible solution. The previous chapter about the fundamentals gives the possibility to consider technical details in the requirements. Thereby all functional, as well as non-functional requirements, are collected and documented so that the fulfillment of the requirements can be evaluated later. Based on the learnings of these chapters, the system and software architecture will get designed in the *Architecture* chapter. The software architecture will consist of the frontend and the backend system.

Once the software architecture is defined, the service will get developed. The *Implementation* chapter documents all implementation decisions. Among them are the crucial aspects, challenges, and limitations of the implemented solution. Lastly, the thesis will assess the fulfillment of the requirements in the *Testing the Requirements Fulfillment* chapter.

2 Fundamentals of Cloud-Native Applications

As previously mentioned in the introduction, many technologies around cloud-native applications are open and standardized. Consequently, the topic is about how to create and deploy applications, but not where to run them [11]. That may sound confusing at first because the word cloud is often associated with known public cloud providers. However, it instead describes the architectural features and principles for achieving the desired properties of cloud-native applications [1]. The three properties most frequently mentioned in cloud-native research are scalability, elasticity, and resilience [1]:

1. **Scalability** describes the ability of systems to respond to higher or lower loads by scaling horizontally (Scale Out/In) or vertically (Scale Up/Down) [12] without having to adapt the architecture of the cloud-native application [13]. Horizontal scaling means adding or removing an entire node in the distributed system. Vertical scaling means adjusting one or more specific resources within a node, for example, the CPU, memory, or storage [12].

2. **Elasticity** is the continuation of scalability. Herbst et al. define elasticity as "the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible." [14] The term elasticity is used extensively in the marketing of public cloud providers [14] since the combination of elasticity (on-demand availability of computing resources from cloud providers) and the pay-as-you-go pricing model results in better cost efficiency [15]. Conversely, this means that you use (and thus pay) only for the resources that you actually need.

3. **Resilience** illustrates the ability of an application to recover itself from failure and continue to function. As Microsoft states it in their .NET documentation: "It's not about avoiding failure, but accepting failure and constructing your cloud-native services to respond to it" [16]. The failures can include unexpected latencies, host and hardware failures, and temporary faults like blockages caused by long-running processes and short-term overloaded services [16].

To achieve these properties, developers can apply the following architectural styles and principles.

The first step towards cloud-native applications is to align the architecture with a **distributed system**. Cloud-native applications are, by definition, distributed [1]. That is because the traditional monolithic architecture prevents the achievement of the desired cloud-native properties. The modules of monoliths depend on shared resources, and modules cannot be executed independently [17]. That leads to significant disadvantages. Every change within a module requires the restart of the whole application, which can result in higher downtime [17]. The conflicting requirements of different modules cause inefficient deployment environments, since there may be modules that are computationally intensive and others that are memory intensive [17]. Furthermore, monoliths limit the scaling of an application when only single modules are stressed. Replicating the application leads to an inefficient allocation of resources to modules that do not have a high load [17]. Architects and developers must therefore consider and enable a distributed architecture from the very beginning. Distributed in this context means that the application is split up according to its individual domains. The resulting components then run distributed on different machines (or nodes) and collaborate by communicating via messages passed over the network [18, pp. 2]. In the context of cloud-native applications, the relevant properties of distributed software applications are scalability, concurrency, and independent failure of the components [18, pp. 3–6, 15].

There are many examples of architectural styles that implement the concept of distributed systems [18, pp. 56f.], but cloud-native applications in particular can be attributed to a specific architectural style that fulfills the characteristics of a distributed system. In 2017, Kratzke et al. found that the terms "cloud-native" and "native cloud" "show increasing momentum and are emerging from an evolutionary process starting with the service-oriented architecture approach, system virtualization, cloud computing, operating system virtualization (aka container) and ending in a recently most popular approach: **microservices**."[1] Compared to the service-oriented architecture, the more modern and "pragmatic" [1] microservices approach is not only the most popular [19] but also the approach most aligned with cloud-native principles [20]. The microservices architecture decomposes the application into minimal [17], loosely coupled, and independently executable [11] and deployable [21] services that get deployed fully automated [21]. Services are designed with scalability and constant failure in mind and hence should be stateless or as stateless as possible [22] [1]. A *shared-nothing* architecture in which each node can answer requests independently and autonomously with its own available resources is particularly suitable for this purpose [23]. The state is then handled by backing services (usually a database), which are replicated on each node [23]. The term "minimal" in the definition refers to the functional scope, which is only supposed to cover the underlying concerns [17]. A concern is organized around business capabilities and not on the technology layer so that you can have more independent and cross-functional teams [21]. That leads to the fact that different services can be heterogeneous and polyglot [17]. Therefore, particular standards have emerged to be able to leverage this heterogeneity: [1]

- Ensuring **loose coupling** of microservices so that services communicate with each other in a well-defined way without having to know how the requested functionality is implemented. As long as the interface of a service does not change, developers can change the implementation of one service without affecting the functionality of other services [24]. That facilitates the separation of concerns and the independence of individual microservices. Therefore, it allows services to be developed, tested, deployed, monitored, and consumed independently of each other [25]. Two approaches implementing loose coupling are the event coupling (via message exchange) and the data coupling (a common isolated state using a database) [1].

- Use of **simple and scalable communication protocols** to connect the heterogeneous services. The most widely used architectural style in the context of microservices is *Representational state transfer* (REST) [1] [19], which in turn is based on the stateless *Hypertext Transfer Protocol* (HTTP) and that "imposes several [architectural] constraints" to "provide uniform interface semantics" [26]. Both approaches have proven themselves over many years in distributed web services, namely the World Wide Web. Microservices employ HTTP "in a very pragmatic way to build distributed, large scale, massively (horizontal) scalable and elastic cloud[-native applications]" [1]. Services that are implementing the REST style are called RESTful services [24]. The services are asynchronously exchanging JSON-serialized [1], self-describing messages without needing to know or remember the state of the conversation [24].

- Encapsulating services and all its dependencies into **standardized self-contained deployment units** [1] so that services always behave the same and can be migrated efficiently and with less risk [22]. The use of containers is the logical consequence for this problem because they are much lighter, more efficient, and easier to manage compared to virtual machines [22] [27]. Container virtualization offers namespace isolation to give processes an isolated view of the system (e.g. file system and networking) and the ability to limit resource usage via Linux control groups [28]. The most popular container runtime [19] and now considered the de facto standard runtime is Docker's *containerd* runtime [1]. The runtime is highly standardized by the OCI runtime [29] and image [30] specification and got adopted by many cloud providers and open source projects – including Kubernetes [31]. That results in high portability and robustness within all compliant runtimes and adopters. We will take a practical look at Docker and its architecture in chapter 2.1.

That demonstrates that the distributed microservices architecture style is suitable as a basis for cloud-native applications. However, breaking down an application into minimal container-based microservices does not solve the problem of managing them, namely the automatic scaling and the resulting elasticity. It requires further principles that make it possible to achieve cloud-native properties.

Containers must be actively managed to guarantee the best possible resource utilization. For this purpose, **elastic platforms** are used which abstract the underlying infrastructure and on which resources can be requested via a uniform interface [32]. The common term for an elastic platform is a *container orchestrator* [22]. The managed system, consisting of physical nodes and the containers or applications, is called a cluster [33]. The fundamental functions of container orchestration platforms are: [33]

- **Establishing the desired cluster state** (e.g. automatic scheduling and scaling of the containers; resource allocation)
- **Providing high availability and reliability** (e.g. redundancy of components on different nodes; load balancing; health management consisting of fault detection and self-healing)
- **Ensuring security** (e.g. container image integrity verification; access management like attribute- and role-based access control; secret management)
- **Simplifying networking** (e.g. network isolation; dynamic port allocation and routing through the abstraction of a *service* [34])
- **Service discovery** (e.g. a service registry holding the network addresses)
- **Providing monitoring and governance** (e.g. resource usage metrics at the layer of containers, services, applications and nodes; auto-scaling; separation of the container's lifecycle and its logs)
- **Enabling continuous deployment without disruption** (e.g. allowing various deployment strategies like incremental, blue/green and canary deployment [35]; rollback solutions, declarative management of the desired cluster components and state)

The last point, that all cluster components are described declaratively, has the significant advantage that the complete system state can be versioned and described in a single repository. This principle is called *GitOps* (Git operations) [36]. As the name suggests, for instance, a pull or merge request within the Git repository initiates operational changes of the cluster. That means that the cluster adapts any change in the code (both of the infrastructure and the application). The automated continuous integration and deployment (CI/CD) pipeline is responsible for this adaptation by updating the desired system state. In addition, versioning of the cluster components also facilitates the restoration of previous system states and allows code reviews for infrastructure-related changes. Since the GitOps principle relies particularly on the CI/CD, we will explain how it works in detail using GitLab's solution as an example in chapter 2.3.

In production environments, Kubernetes currently leads as the most used container orchestrator [19]. It implements all fundamental functions mentioned above. Therefore, in chapter 2.2 we will present these concepts and features of container orchestrators using practical examples based on Kubernetes.

In summary, the necessary principles to achieve the cloud-native properties consist of a [22]

1. **microservices-oriented architecture** that decomposes applications into minimal, loosely coupled services that are massively scalable through simple communication and as much statelessness as possible.

2. The services are **packaged in containers** together with all dependencies, resulting in optimal reproducibility, portability, and isolation.

3. A **container orchestrator** again schedules, monitors, and scales all these containers to achieve optimal resource utilization, which is illustrated by the elasticity property of cloud-native applications. The cluster components are described declaratively to enable

4. **continuous integration and deployment**, which allows the automation of operational changes and having a single source of truth.

2.1 Containers with Docker

As already stated, *Docker* is the de facto standard tool when working with containers. You can "build, run, and share" [28] containerized applications with it. A comparison between virtual machines and containers helps to explain the components of Docker.

The main difference is that virtual machines (VMs) isolate different systems, whereas containers isolate separate applications or services (see Fig 2.1). Each isolated system requires a complete image of an operating system (the guest OS), which is managed by a hypervisor and runs on a host system. The hypervisor must virtualize and split the physical resources between each guest system. This virtualization of resources is not found in containerization because containers run directly as a process on the host system. In Linux, these processes are isolated via *namespaces*, and their resource usage is restricted by *control groups*. The Docker daemon is a RESTful service. It is responsible for the management of all Docker

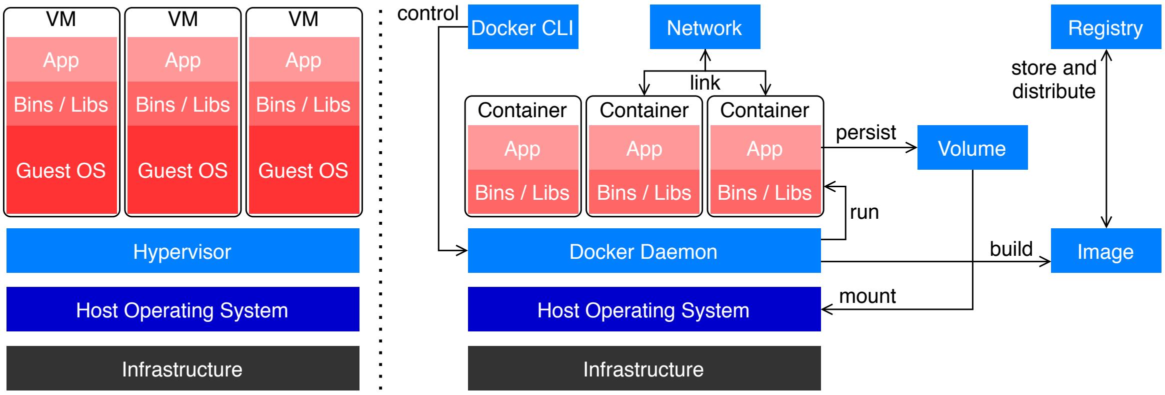


Figure 2.1: Comparison between system (left) and application (right) virtualization.

objects. These can be containers, but also images, isolated networks, and data volumes (see Fig 2.1). All these objects can be controlled by the Docker CLI, which sends the appropriate HTTP requests to the Docker daemon, or directly through the REST API of the Docker daemon. In this introduction we will use the Docker CLI commands.

Before you can start and run a container, you have to build it. For this purpose, Docker provides the **image** component. The Docker documentation says that "an image is a read-only template with instructions for creating a Docker container" [28]. Usually, you use other images as a basis and then extend it with your customizations, for example installing the application's dependencies and copying the source code into the image. On the other hand, a **container** is a running or runnable instance of an image. The distinction is because a container can have multiple states. For example, you can create a container but not start it yet (*created*). Then you can start the container (*running*) and also pause every process inside of the container (*paused*). If you stop the container (*exited*), it not loses its state. As long as the container has not been removed, you can start it again (*restarting*).

We will now build a simple, minimal *Hello World!* HTTP service with the JavaScript runtime *Node.js* [37] and the web framework *express* [38]. We take the application code, shown in Listing 2.1, as given. Just with this little application, you can already illustrate three essential concepts of Docker. The first is to expose the application on port 80, so it is reachable from outside of the container. The second concept covers environment variables. The application reads the environment variable `GREET_NAME` in line 5 and sets the constant `greetName` accordingly. The application also has an endpoint to retrieve the data of the log file that gets updated with every request made to the *Hello World!* endpoint. As long as the container does not get removed, it will maintain its state. But to be able to see the logs even after the removal and recreation of a container, we will store the log file persistently on the host machine. That is solved by the third concept using a data volume mounted into the container. Volumes can only be deleted explicitly and are, by default, detached from the lifecycle of a container.

```javascript
const express = require('express')
const fs = require('fs')
const app = express()
const port = 80
const greetName = process.env.GREET_NAME || 'World'

app.get('/', (req, res) => { // Hello World! endpoint
  const logEntry = `${Date.now()}: {${req.ip}: ${greetName}}\n`
  fs.appendFile('log.txt', logEntry, function (err) {
    if (err) throw err
    console.log(`Saved log: ${logEntry}`)
  })
  res.send(`Hello ${greetName}!`)
})

app.get('/log', (req, res) => { // log endpoint
  fs.readFile('log.txt', function (err, data) {
    if (err) throw err;
    res.send(data.toString())
  })
})

app.listen(port, () => { console.log(`App listening on port:${port}`) })
```

Listing 2.1: Hello world service in JavaScript with express. (app.js)

Listing 2.2 shows the Dockerfile with all steps to build a docker image for the Hello World service. Each line in a Dockerfile starts with a keyword which stands for the respective building operation. The file shown here demonstrates the most essential and for this work relevant operations.

The first step is to select a base image from which to build the custom image. In this case, a JavaScript program needs to be executed. So Node.js is a suitable runtime. The official node image[1] has all the necessary tools to run JavaScript programs – including the Node Package Manager (`npm`). Similar images for any other programming language are

[1] https://hub.docker.com/_/node (visited on 10/09/2020)

available in the Docker Hub[2]. The FROM operation selects the base image in the form of <repository>:<version> .

```
FROM node:12
EXPOSE 80
ENV GREET_NAME=World
WORKDIR /hello-world-service
RUN mkdir data
# VOLUME ["/hello-world-service/data"]
COPY package.json . # Dot (.) indicates the present working directory
RUN npm install
COPY app.js .
ENTRYPOINT ["node", "app.js"]
```

Listing 2.2: Dockerfile for building the hello world service image. (Dockerfile)

Since the Hello World service has an HTTP interface, the container has to be configured to be accessible from outside of it. The EXPOSE operation is a best-practice for the documentation of service ports but has no actual functionality. It should show users of the image the intended port which needs to be exposed. The actual publishing of the service on the host machine happens when the container is started. This is done with the --publish <port-host>:<port-container> flag.

When starting the Hello World service, the application initializes its constants. The service reads the GREET_NAME environment variable and sets the greetName constant accordingly. So that the constant is never empty, the ENV operation sets a default environment variable for GREET_NAME . This environment variable is set for each operation in the build process of the image and during the runtime of the container. Every defined environment variable can later be changed, for example when starting a container or at runtime by interacting with the Docker daemon. Furthermore, additional environment variables can be created and modified that are not included in the Dockerfile. For this reason, the environment variables defined in the Dockerfile are normally used as a default setting for the application running inside of the container.

Next, the WORKDIR operation is used to specify the working directory in which any future operation should be performed. This improves the readability of commands and the maintainability of the Dockerfile. If the specified path does not exist, it gets automatically created using mkdir commands. So the WORKDIR operation can be seen as a combination of mkdir and cd . As with the change directory (cd) command in Linux, a distinction is

[2] https://hub.docker.com/ (visited on 10/09/2020)

made between absolute and relative paths. If the string starts with a slash (`/`), it indicates an absolute path starting from the root path. Without a slash, it points to a relative path starting from the present working directory.

The `RUN` operation can now execute a command in the desired working directory. Since the service stores the logs under the path `data/log.txt` , this path has to be created first. In line 5, the `mkdir` command creates the `data` directory for storing the logs. This is the path a volume can be mounted at for persistence. Docker offers several ways to create volumes. As indicated in the comment in line 6, a volume can be created automatically for each container and mounted at the data path inside of the container. However, this has the disadvantages that the volume is empty for each new container, it is assigned a random name, and it can not physically be provisioned by a container orchestrator. The two other, and in this case more reasonable solutions, are manually created volumes, which get a desired name and are managed by Docker, and bind mounts, which mount a path from the host system into the container. The former can be used by creating a volume with the Docker CLI command `docker volume create hello-world-data` and mounting it inside of the container using the flag `--volume hello-world-data:/hello-world-service/data` . In the latter case, the flag `--volume <your-host-directory>:/hello-world-service/data` is used to mount a path from the host system into the container. However, the path from the host system must be absolute to avoid possible ambiguity between volume names and paths (since the `--volume` flag is the same for bind mounts and volumes). Example use cases are volumes to share data between different containers, to persist data, and to load configuration files from the host system into the container. It should be noted, that even the first solution with a Dockerfile-defined and randomly generated volume can be reasonable too. That is because application data should preferably not be written and read in the container itself. The I/O performance is much lower compared to native programs or volumes because you work in the so-called writeable layer of the container [39].

The next step is to install the dependencies of the service. As mentioned above, the service is based on the web framework express which must be installed first. The `COPY` operation copies the `package.json` file (a simple, `npm` -specific file listing all dependencies) from the host system into the image. The path of the file to be copied is relative and originates from the path of the Dockerfile. That means that the `package.json` file must be located in the same path as the Dockerfile from which the image is built. Afterward, the `RUN` operation in line 8 executes the command `npm install`[3]. Once all required packages are installed, the `COPY` operation copies the source code of the service from the host system into the container.

[3] In production environments you should use the `npm ci` command and copy the `package-lock.json` file into the image too. The command is faster, stricter and reduces inconsistencies by using only the versions specified in the `package-lock.json` file. See here for more info: `https://docs.npmjs.com/cli/ci.html` (visited on 10/10/2020)

The reason why this happens only after the dependencies have been installed and not the complete folder is copied in line 7 is because images are based on a layer system. Each `RUN`, `COPY` and `ADD` operation in the Dockerfile adds a new layer to the image. That allows the individual layers to be cached and the build time of an image can be significantly reduced. In practice, this means that if the layer from line 7 (i.e. the dependencies of the service) does not change, then this layer and the following one do not need to be rebuilt. Correspondingly, the dependencies do not have to be reinstalled in the image after every change to the source code. One best-practices is to copy frequently changing files (in terms of continuous software development) into the image only late because the cache gets invalidated for all subsequent operations [40].

The last step is to give the image a default command. Either the `ENTRYPOINT` or `CMD` operation is possible for this purpose. This example uses the `ENTRYPOINT` operation in line 10 which makes the image executable. The defined default command, as long as it is not overwritten manually at startup, is automatically executed at every startup and thus represents process number 1. This process also receives the termination signal accordingly. Both the `ENTRYPOINT` and `CMD` operations use the so-called exec form with the JSON syntax `ENTRYPOINT ["executable", "param", "param"]`. The first entry in the array is always an executable and the following entries are the parameters. The difference between `ENTRYPOINT` and `CMD` is that you can optionally define both operations in the Dockerfile at the same time while using the `CMD` operation as a means for default parameters. For example, an executable file and parameters can be enforced during startup of a container using the `ENTRYPOINT` operation and further overwritable default values can be passed in the form `CMD ["param", "param"]` using the `CMD` operation.

The Dockerfile is now complete. The `docker build -t hello-world:0.0.1 .` command builds and tags the image for the Hello World service. An image tag can be thought of as a counterpart to a Git tag and is used to reference an image version. In this case, the repository name is “hello-world-service” and the version is “0.0.1”. The dot (.) in the build command indicates the directory for the build context, where it will look for the Dockerfile, and from where it will copy the files when building the image.

To create and execute a container of the Hello World service, the persistent volume for the data directory needs to be created first with the `docker volume create hw` command. It creates a volume with the name “hw”, which can be mounted in the next step when starting the container. The `docker run -e GREET_NAME="Tom" -p 80:80 -v hw:/hello-world-service/data hello-world:0.0.1` command finally creates and starts the container, and also overwrites the default `GREET_NAME` environment variable.

The remaining question is how to store and distribute the image – both to other developers and onto servers. For this purpose, there are container image registries. They use the tags to reference container images. Accordingly, an image must always be tagged before you can push it into a registry. The tag has the form `<host:port/repository:version>`. Here, “host” always represents the hostname, the fully qualified domain name, or the IP address where the registry can be reached. To push and pull images, developers can simply execute the `docker push <tag>` and `docker pull <tag>` commands with the tag in the form defined above.

2.2 Container Orchestration with Kubernetes

The manual pulling of a container image and its start with multiple flags is time-consuming and does not scale for many containers, applications, and servers. Chapter 2 has already listed the functions that Container Orchestrators provide. This chapter now explains Kubernetes' concepts and extends the *Hello World* example from the previous chapter using Kubernetes workloads.

2.2.1 Architecture

Kubernetes consists of a Master/Worker architecture (see Fig. 2.2) [41]. Each (physical or virtual) machine is called a node, and all nodes together form a cluster. The master node consists of the control plane (blue colored, see Fig. 2.2), which is the container orchestration layer. That control plane manages the lifecycle of all containers. The controllers are responsible for responding to node failures, ensuring that the correct number of replicated pods is always running, connecting containers with services via endpoints, and automatically creating default accounts for new namespaces. The scheduler then selects a node to run pods that have not yet been assigned to a node using various criteria. Furthermore, the key-value database *etcd* stores the services, configurations, and secrets. All these internal components of the control plane are accessed via the API server. That applies to both *kubectl* (the command line interface) and the worker node components.

On the other hand, the worker nodes run the containerized applications. Each worker node consists of the kubelet, which is an agent responsible for the management and health of local running containers, and the kube-proxy, which maintains the iptables entries for endpoints and services. Also, to be able to run containers, each node provides a container runtime.

Kubernetes objects (also often called manifests [42]) describe the state of resources in the cluster in declarative form. For example, each object can represent a containerized application, available system resources, or rules on how an applications can and must behave.

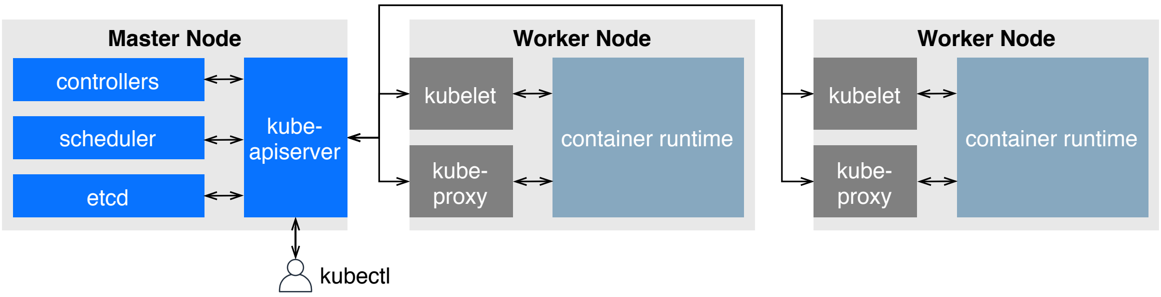


Figure 2.2: Example Kubernetes cluster and its components [41].

All objects together form the desired cluster state. Kubernetes constantly tries to establish and maintain this desired state. Internally it keeps track of two states for each object. It holds the `spec` field, which expresses the desired state of an object, and the `status` field, which allows monitoring the current state of an object. The control plane is responsible for continuously comparing these two object fields and adjusting the state of an object, if necessary so that the two fields match. The objects are usually described using YAML (an acronym for "YAML Ain't Markup Language" [43]), a human-readable data serialization language. YAML is a superset of JSON. The Kubernetes client (kubectl) converts YAML-defined objects to JSON strings before making requests to the Kubernetes API. Each object to be created consists of a field for the API version to be used (`apiVersion`), the type of object (`kind`), the metadata for identifying objects (`metadata`), and the desired state of the object (`spec`). The latter differs in its structure depending on the API version and type of the object [44].

2.2.2 Namespaces

However, the complex architecture of Kubernetes has the disadvantage that it involves a relatively large amount of overhead, both from an operational and financial point of view. Therefore, Kubernetes usually makes more sense for large to very large applications. In order that teams with smaller applications can also benefit from the advantages of Kubernetes, a multi-tenant solution can make sense. Using a cluster with multiple teams leads to more efficient use of resources and takes advantage of economies of scale from which otherwise only large corporations can benefit. The challenges are the isolation between teams or applications and the fair distribution of resources among applications. To overcome these challenges, Kubernetes offers namespaces. These are "virtual clusters backed by the same physical cluster" [7] but still share the resources of all nodes between all namespaces. That means that a set of nodes does not dedicate itself to a single namespace. They can thus isolate environments between multiple teams and projects. An important point, also for the solution to be developed, is that namespaces cannot be used to represent hierarchical structures, since namespaces cannot be nested into one another.

Namespaces enable further principles for optimized isolation and fair sharing of resources. Each namespace provides a service account by default, which clients can use to authenticate themselves towards the API server. These service accounts allow assigning rights to clients. For example, you can authorize a client via a service account to read certain resources but forbid it to modify them. The two relevant authorization alternatives of Kubernetes are attribute-based access control (ABAC) and role-based access control (RBAC) [45]. With ABAC, you can create a policy that directly associates a service account with a rule allowing certain operations for a specified resource. RBAC, on the other hand, requires the creation of a role beforehand. In contrast to ABAC, this role can contain many rules that allow certain operations for specified resources and is either cluster-bound (ClusterRole with rules applying for all resources in the cluster) or namespace-bound (Role). The RoleBinding resource then associates the role with a service account.

Further, limits and quotas, applicable to namespaces, solve the challenge of fair sharing of resources. Resource quotas can divide the physical resources available in a cluster among the namespaces. Thus, they restrict resource usage within individual namespaces. But it does not necessarily have to include only physical resources such as the number of requested CPUs, memory, and storage. The restrictions can also include the number of Kubernetes-specific objects [46]. The latter can be helpful because the cluster and nodes have a definite capacity of resource objects they can manage [47]. Unlike namespaces without resource quota, developers who want to create Kubernetes objects in a restricted namespace must always specify how many resources each object requires. That is the only way the controller can decide whether it can create the object in the namespace, or it has to reject it due to excessive resource requirements [46]. To prevent creation failures due to undefined resource requirements, Kubernetes offers default values for containers, pods, and volumes via the LimitRange object [48]. As the name suggests, it additionally enforces minimum and maximum limits for every object's resource request. That has another advantage that not one container can claim all resources for itself.

2.2.3 Workloads

The term pod has already been mentioned multiple times, but a definition is still missing. The Kubernetes documentation says that "Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. A Pod [..] is a group of one or more containers, with shared storage/network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context" [49]. In terms of Docker, this shared context means that the containers run in the same Linux namespace and control groups with a shared filesystem. The containers inside of a pod are therefore tightly coupled. In a non-cloud context, this concept is the counterpart to applications running on the same physical or virtual machine.

A single container per pod is the most common scenario. However, there are also other use cases where the actual application requires additional containers. For example, there are so-called init containers, which can get executed with different images than the application itself. They prepare the environment like volumes by running utility programs. Before the application container can start, the init containers must run to completion and thus "offer a mechanism to block or delay app container startup until a set of preconditions are met" [50]. Another example could be sidecar containers that run alongside the application container and constantly update the contents of the shared volume. Imagine a web server that serves static files that get updated by a file puller sidecar container.

Typically you do not work with pods directly. There are higher-level workload objects that automatically create and manage pods. These workload objects are always based on pods and thus require pod templates that are part of the desired state of a workload. The workload controllers then take control and monitor the pods created on behalf of workloads. The following list describes the most common and for this thesis relevant workloads:

- **Deployment**: The Deployment workload provides controlled handling of updates, scaling, and rollback functionality. It is based on the ReplicaSet workload, which is a lower-level workload that "ensures that a specified number of pod replicas are running at any given time" [51]. Deployments are an ideal workload for stateless applications as they do not maintain a state for the administered pods. The pods thus are treated as disposable and independent units [52].

- **StatefulSet**: The StatefulSet workload provides the same functionality as the Deployment workload, but additionally "maintains a sticky identity for each of [its] Pods" [53]. That means that a pod will retain a unique network identifier and its persistent storage across rescheduling. Also, it guarantees an ordered deployment and scaling through ordinal indexes. That means, for example, when you deploy a set of two pods, the StatefulSet controller will always start deploying the first pod (and its storage) with an ordinal of 0. It waits until the pod is running and only then starts the next one with an ordinal of 1. When scaling down the set, the controller performs the deployment in the opposite direction. It always removes the pod with the highest ordinal first. When scaling the set up again, each pod will resume in its previous state by keeping the same storage. This workload is especially helpful for stateful applications like databases.

- **Job**: The Job workload covers short-lived tasks, which, unlike long-lived services, terminate after a definite time. The workload is typically useful for asynchronous tasks that are CPU-bound and which the main application should not run itself. Running a job thus can prevent blocking the event loop of the actual application and allow much more diverse tasks to be executed. Like any other controller, the job workload controller creates pods but also ensures that a specified number of them terminates successfully. This number is dependent on the type of the job. Kubernetes offers three main types of jobs. The first type is a single pod that needs to terminate successfully once. If that pod fails it will restart it until it terminates successfully[4] or until the backoff limit is reached. In the latter case, the job fails. The second type is a job that runs a pod for a specified number of completions sequentially. The last type is a job that runs a specified number of pods in parallel and succeeds if all pods terminate of which at least one pod successfully terminates. In case not all pods can get created directly, which might be the case in restricted namespaces, the controller will not create additional pods if at least one pod has been successfully terminated.

- **CronJob**: Just like the time-based scheduler in Linux, namely cron, the CronJob workload runs tasks on a repeating schedule. This schedule is written in the cron table format. The CronJob controller does nothing else than creating Job workloads periodically. This can be helpful for creating backups and other maintenance tasks.

2.2.4 Services

Containerized applications can run distributed and replicated in multiple pods and on multiple nodes using the Deployment workflow. However, the resulting endpoints of all pods must also be discoverable to be able to send requests to them. The challenge is that pods are disposable and each of them gets a new IP address when it gets started. To solve this problem Kubernetes offers the Service object. It abstracts a set of pods to a logical unit that can be accessed by clients. That is possible because pods are designed to be interchangeable so clients can send their request to any pod. The service object provides a stable DNS record and automatically balances the load between all pods. The service controller knows to which endpoints it must forward the requests using selectors. These selectors must be defined in each pod template and must match the selector from the service (see Fig. 2.3, the `app` keyword represents a selector) [54].

The Service object can now be used to send requests to replicated applications. But this works only within the cluster because the service has a type of `ClusterIP`. That means that the service exposes itself to a cluster-internal IP address. Two common ways to be able to access services from outside of the cluster are making use of node ports or Ingress objects. When setting the Service's type to `NodePort`, the service controller allocates a port on

[4] Provided that the restart policy is set to `OnFailure`.

each node in the cluster. Service requests can now be made to every node on the allocated port. Each node then proxies the request to the corresponding service. Since this method creates new entry points for each service in a particular port range (the default range is from 30000 to 32767), it usually makes it necessary to proxy these entry points again to be able to reach them under a unified address. For this purpose, Kubernetes offers the Ingress object that exposes all services at the same address and consolidates all routing rules in a single resource [54].

All Ingress objects rely on an Ingress controller that must be manually deployed in the cluster first. That can be a proxy server like Nginx, which is also the officially supported one by the Kubernetes team. Unlike a service, the Ingress object provides additional features such as TLS termination, name-based virtual hosting (see Fig. 2.3), and HTTP URI-based routing [55].

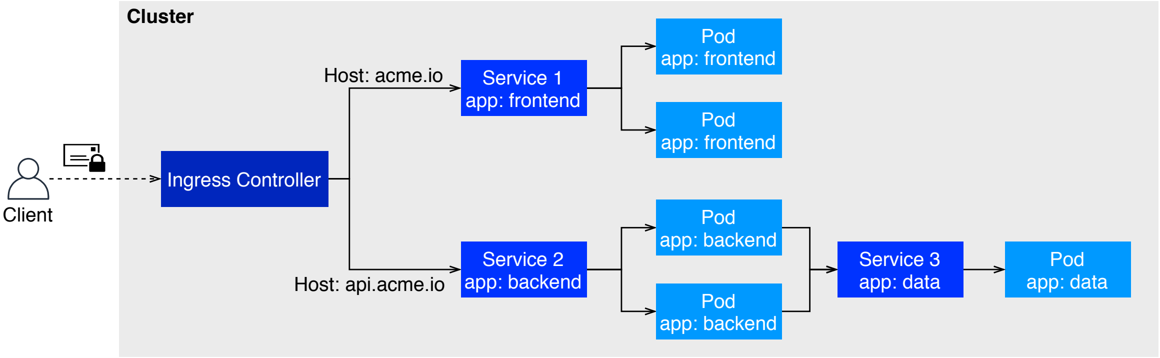


Figure 2.3: Example Ingress application resolving to two virtual hosts [55].

2.2.5 Hello World Service Example

Finally, we will combine the above Kubernetes concepts with the help of the Hello World example. The example covers the three concepts (environment variables, volume, and publication of the service) we already know from the *Containers with Docker* chapter. Accordingly, the Hello World service should be accessible under a hostname from outside the cluster on port 80. The communication between client and cluster should be encrypted using TLS. The Ingress object (see Listing 7.1, Line 48) serves this purpose. In total, one instance of the Hello World service shall run in the cluster, which stores its logs in a volume. That is solved by a StatefulSet, which contains a template for a volume (see Listing 7.1, Line 37). The volume gets created together with the pod and it always gets mounted by the same pod – even after rescheduling and scaling. That is due to the sticky identity provided by the StatefulSet. In addition, the underlying ReplicaSet ensures that the container in the pod gets restarted if it fails. In this example, we want to set the environment variable `GREET_NAME` to “Gandalf the Grey”. The container specification solves this through the `env` keyword (see Listing 7.1, Line 31). The connecting piece between the Ingress object and the pods is the Service object. The service adds each pod with the selector `app: hello-world-pod` (see Listing 7.1, Line 10) to its endpoints list and balances the load between them. In this case, it is only one pod, but the replica number can be upscaled.

To deploy the app in the cluster, we can simply run the `kubectl apply -f hello-world.yaml` command. The Kubernetes controller will create all resources according to the specification. If you now want to update the Hello World service, you can change the `hello-world.yaml` file and run the `apply` command again. For example, we can update the service with another name to greet. When changing the `GREET_NAME` environment variable to "Gandalf the White" and, again, running the `apply` command, Kubernetes will stop and remove the outdated pod and then recreate it with an updated one. The application will still retain its logs in the volume. But the disadvantage of having volumes for the logs is that it does not scale. Scaling the service would mean creating multiple different volumes for each pod. To solve this issue, it is a good practice to outsource the logging into a dedicated service, which records the logs of every pod in the cluster. That demonstrates the cloud-native principle that your services should always be as stateless as possible to be scalable and resilient.

2.3 Continuous Integration and Deployment with GitLab

CI/CD is all about automating development practices and minimizing the risk of bugs and conflicts. CI stands without doubt for continuous integration, but the acronym CD can be interpreted in two ways. One is continuous delivery, and the other is continuous deployment [56]. Why this thesis chooses the variant of continuous deployment, will be explained in a moment.

Continuous integration sets the goal of merging new code from development branches to the main branch as often as possible. By having developers writing tests, these tests can get executed automatically each time developers push new code to the repository. That is what the commonly called CI pipeline is responsible for. After building the application and running the tests, it then validates whether the tests were successful or if the new code causes errors. Only if the pipeline has passed successfully, the new code can be merged into the main branch. Also, developers must first merge any update that takes place on the main branch, before they can merge their development branch into the main branch – regardless of the success of the pipeline. Doing the integration as often as possible reduces discrepancies between development branches and their different dependencies. The mantra is that the whole process of building and testing should be automated and without human influence [56].

Continuous delivery or deployment now extends the principles by a further stage. The pipeline just mentioned is usually divided into different stages. Continuous integration introduced the first two stages, namely the testing and building stages. The pipeline's principle is that it only executes the next stage if it successfully completed the previous stage. Accordingly, a third stage can be introduced, which automatically deploys the code to the servers. The difference between continuous delivery and deployment is that the former requires human intervention, which is the manual execution of the deployment stage [56]. Continuous deployment, on the other hand, deploys code from the main branch to the servers automatically. Thus the main branch always represents a single source of truth. That is especially advantageous when working with Kubernetes since it is possible to describe the entire system declaratively with just the manifests. The CI/CD pipeline can thus automatically change the system state without operational effort. As described earlier, this principle is called GitOps. By having the infrastructure as code, developers can now conduct code reviews, write comments, and use `git diff` for operational changes. In addition, the infrastructure's version control also makes it easy to roll back to previous system states [36].

To use GitLab's CI/CD integration, all you need to do is add a `gitlab-ci.yml` file to the repository, listing the different stages and their procedures. The principles outlined here are not limited to GitLab, but also apply to other Git hosting providers, and differ only in implementation details. Listing 2.3 describes a single stage, namely the testing stage of a Node.js application. Each stage consists of one or more jobs, and each job, in turn, consists of a script that uses standard bash commands (Bourne shell) [57]. You can specify many more conditions and properties for a job, like caching behavior, parallelism, and when stages should be executed automatically. Every time a developer performs a Git push, it automatically initiates the pipeline to start. Each job then gets executed in a dedicated container. The containers download the repository and checkout on the corresponding branch from where the push was executed. Afterward, each container executes its job's script. If the exit code of a job is non-zero, it means that the pipeline has failed and all subsequent stages will not get executed [58].

```yaml
Test app: # one job in the 'testing' stage
  stage: testing
  image:
    name: node:12
  script:
    - npm ci # install the dependencies
    - npm run test # run the tests
```

Listing 2.3: Minimal GitLab CI example pipeline for the testing stage. (gitlab-ci.yaml)

3 Requirements Analysis

This chapter will gather the requirements for the solution to be developed through a stakeholder analysis. The advantage of a stakeholder analysis is that the requirements do not necessarily have to focus on the university environment, but will also consider other environments and groups. Since the bachelor thesis has a fixed deadline, it makes sense to prioritize the requirements and focus on the most important ones. For the prioritization, the *MoSCoW* method will be applied. That means that there will be four priority classes, namely *Must-Have*, *Should-Have*, *Could-Have*, and *Won't-Have*. The Won't-Have category only applies to the completion time of the bachelor thesis and does not categorically exclude a requirement forever. Therefore, requirements can also leave room for future enhancement opportunities.

3.1 Stakeholder Analysis

First, the requirements defined by the bachelor thesis itself (see *Task Definition*) get recorded for each stakeholder. Since these requirements are decisive for the success of this bachelor thesis, they are given the highest priority. Afterward, possible further requirements get analyzed and recorded. The associations of individual requirements with the respective stakeholders are merely assumptions. That helps to put oneself in the position of a stakeholder group, to identify requirements more efficiently, and to easier understand the reasons behind requirements. The appendix finally lists all functional (see Table 9) and non-functional (see Table 10) requirements in tabular form. The tables contain the atomic requirements and are thus higher in number and level of detail than the following stakeholder analysis.

3.1.1 End Users

The end-user group describes all the people who are supposed to use the solution. These are, for example, students, researchers, and software developers.

As a user, you want to authenticate yourself quickly and easily. Ideally, you don't want to register just for this one solution. Therefore it makes sense to use a single sign-on solution where the users are already registered. For this purpose, there is the requirement that users can log in through GitLab's OAuth 2 interface (see Req. F1). That also gives you the ability to offer the solution only to people who are authorized to do so. In the case of this bachelor thesis, those people are who already have a GitLab account at the university.

If you, as a software developer, decided to host a new project in a Kubernetes cluster, you would have to do several steps manually. That includes creating a new git project and asking a server administrator to create a cluster or namespace. You would also need to set up the project for the cluster environment. These steps result in the requirements for a self-service solution. Users should be able to create both a GitLab project (see Req. F2) and a namespace in the cluster (see Req. F3) on their own. While configuring the project in the self-service user-interface, a user should be able to select the desired technology stack (see Req. F2.5–F2.9), and the solution then initializes the project with a corresponding template application (see Req. F2.10). This template application should additionally be ready to run directly after the project is created (see Req. NF8). That involves setting up a CI/CD pipeline (see Req. F2.2) with appropriate Dockerfiles, Kubernetes manifests (see Req. F2.8) and additional files to prevent secrets leakage (see Req. F2.3). Further, the user should be introduced to the structure of the project and its artifacts (see Req. F2.11) to lower the hurdle even more.

As a user of such a solution, you also want your application to be accessible under a specific domain. The domain name should not be generated randomly but should be based on the name of the project. Kubernetes uses DNS to resolve the address of resources. For example, it uses the scheme `<service>.<namespace>.svc.my-domain.io` for services [59]. The self-service solution is supposed to tie in with this schema by making the project name a subdomain of the cluster domain (see Req. F4). Therefore, a project can only get created if the desired name is available for both a GitLab project and a namespace (see Req. F5). Additionally, the solution shall set up an ingress rule with TLS termination to be able to request services in a web browser (see Req. F4.1).

In addition to concrete functions, users also want non-functional requirements to be fulfilled. It can always happen that the user loses the connection after confirming to create a new project. The solution should be able to handle a connection loss and still provide the user with information about the cluster address and the next steps (see Req. NF5). Users also want the solution and especially its authentication to be secure (see Req. NF7). It shall thus execute the authorization flow on the server-side and encrypt credentials. Also, the solution should be usable on different end devices via the web browser. That is only possible if the user-interface automatically adapts to different screen sizes (see Req. NF13).

The overall goal of the solution is to flatten the learning curve and help users get started with Kubernetes. Therefore, it should not rely on Kubernetes' custom resource definitions (see

Req. NF9). Nevertheless, it should not prevent learning. Thus, the solution should display the generation steps in a comprehensible way so that users can understand what steps are needed to integrate a Kubernetes environment into a project (see Req. NF10).

3.1.2 Server and Kubernetes Cluster Administrators

Since the solution should rely on virtual clusters using namespaces, an administrator has an interest in ensuring that users can only work within the namespace intended for them. The solution must apply authorization rules accordingly (see Req. F3.3). For the account bound to the namespace, the access data must also be created (see Req. F3.5) and provided to the user. The latter is to be solved in a secure way using an environment variable in the GitLab project (see Req. F2.1). Furthermore, it can be necessary for various reasons that administrators can associate a namespace with a specific person. The solution should thus associate a user with the namespace to be created (see Req. F6). Another useful maintenance feature that can relieve administrators of work is the deletion of inactive namespaces (see Req. F8).

It is also relevant for administrators that they can easily install the solution in a Kubernetes cluster. For this purpose, a Helm chart should be used to enable the installation with just one simple command (see Req. NF1). To be able to use the solution in various clusters and organizations, the installation should be configurable (see Req. NF2). Also, the solution should only create new projects if it first checks whether there is enough capacity available. So if a project cannot be deployed in the cluster, the project should not be created either (see Req. NF14). A crucial aspect for administrators is to give third-party applications as few rights as possible to minimize the risk of threats. Therefore, the solution should run without administrator rights for the GitLab server and only use the user rights provided by the OAuth authorization (see Req. NF6).

3.1.3 Software Maintainers

As a software maintainer, you probably have more interest in the non-functional requirements. In order to make optimal use of existing human resources, the technology stack should be selected so that the organization's members are able to maintain it. Therefore it makes sense to align the overall application landscape to one language. In the case of this thesis, the implementation language for the backend is set to Python (see Req. NF3). Furthermore, the requirement is that the solution should be designed as simple as possible. That means that it should be stateless and thus make it simple, horizontally scalable, and robust (see Req. NF4). To be able to maintain and extend the solution, it is also vital that the implementation gets tested well. The requirements thus include automated unit tests for all endpoints (see Req. NF11).

3.1.4 Organization

The organization also represents specific interests to an internal self-service platform. A Kubernetes cluster for shared development and hosting is a cost issue. Of course, financial means are limited, and therefore unlimited computing capacity cannot be made available to end-users. Individual users must be limited in their use of resources. Thus, the solution should apply resource quotas for CPU, memory, and storage usage for each self-serviced namespace (see Req. F3.1). Moreover, a Kubernetes cluster has a maximum number of objects it can manage. To prevent users from creating large numbers of workloads with minimal resource requirements, lower and upper limits should also be applied to individual objects (see Req. F3.2). All quotas and limits should be configurable to the individual needs of organizations (see Req. F3.4).

When organizations offer a service for hosting within a Kubernetes cluster, they should point out their terms and conditions. These may include legal information, what the use of secrets entails, what rules to follow when using ingress, and information about logging and restrictions that apply. The solution shall inform the user of all conditions and request their confirmation (see Req. F7).

Additionally, the solution should be white-labeled to benefit from it not only in the specific use case of the Technical University of Applied Sciences Lübeck. That allows other organizations to easily customize the logos, labels, and terms and conditions to suit their own needs (see Req. NF12).

3.2 General Design Decisions

The requirements conclude a client-server architecture. A purely client-side application is precluded. The reasons for this are that the clients should not be given the possibility to create new namespaces themselves. That would require cluster-wide rights, which only the server should have. Also, authentication and authorization can be solved more securely with the OAuth 2 authorization code flow (which requires a server) than with a purely client-side application.

The user interface should be accessible as a web app in the web browser. There are three possibilities for the delivery of the web app:

1. **Dynamic websites**: The server renders the HTML files dynamically based on the data available to it and then delivers the files to the client. Any interaction takes

place only by exchanging HTML files between the Web server and the client. Thus, the complete logic and state gets handled by the server. That is an adequate solution for displaying personalized content to users. However, this architecture requires a template engine for dynamic generation and does not fulfill the cloud-native principles due to its monolithic nature.

2. **Single-page applications (SPAs)**: The web server delivers only a single, minimal HTML file – no matter which route the client requests. The Document Object Model (DOM) gets built using JavaScript in the client's web browser. This principle applies to the entire web app. When the user clicks on a link or button, the browser does not send a request to the server. Instead it alters the DOM. If the single-page application needs to display user-specific data, it requests it directly from the backend's microservices (typically through HTTP APIs). The services then send back the requested data (typically in JSON format), and the website updates the DOM according to the received data. Typically, a SPA framework also implements lazy-loading, or progressive loading, for better performance and lower data usage.

3. **Static websites**: As with SPAs, the web server only delivers static files to the client. The difference is that the static pages are fully rendered, and there is a separate HTML file for each route. When the user clicks on a link, the client requests a new HTML file from the server. The delivered file then contains the complete, rendered DOM. As with SPAs, user-specific content can be displayed using JavaScript and HTTP APIs. In contrast to SPAs, however, the website can be displayed without executing JavaScript in the web browser. That has the advantage that the website is more likely to be indexed by search engines [60]. But this point is to be neglected since the self-service solution is supposed to represent an internal platform[1].

For the reasons mentioned above, the decision was made in favor of a single-page application. The backend shall implement a HTTP-based API to which the SPA shall send its requests. The backend shall be responsible for the authorization procedure, the creation of a project including the namespace integration, and the scaffolding of the initial source files. All system components (which need to be derived in the next chapter) shall run in the same Kubernetes cluster for which new projects will get created. This implies that the components shall be containerized.

[1] Also, there are tools available to generate static sites from SPAs. See `https://nextjs.org/` and `https://nuxtjs.org/` (visited on 10/21/2020)

4 Architecture

The achievement of the requirements depends considerably on the system and software architecture. The system architecture should give a high-level overview of the system components and their interaction. Before being able to describe the system architecture of a possible solution, we must determine these components. In chapter 4.2 we will identify (business) activities and aggregate them into domains. This analysis leads to the architectural approach that the solution should apply. Accordingly, a diagram will represent all the system components that have been derived. Afterward, we will design the software architecture of the individual components in chapter 4.3. The design should not yet specify every detail so that adjustments are still possible during the implementation phase. That is because at the beginning of a project, the uncertainty is still relatively high, and by working in an agile way, you can validate and adapt design decisions iteratively.

4.1 Preconditions

An architectural design is always based on certain assumptions. These must be made in advance to create an architecture reliably. There are two external systems in total. One is the Kubernetes Cluster, and the other is the self-hosted GitLab server. For both systems, certain conditions must now be established. Based on these conditions, a system and software architecture can then be created.

One requirement is that all system components shall run in the same Kubernetes cluster as the projects to be created. That has the advantage that the scaffolding application can assume the same preconditions for the projects to be generated. It ensures that generated projects are most probably executable in the cluster because they expect the same technologies and cluster configuration as the scaffolding application requires itself. The following list now elaborates on Kubernetes cluster components, which eventually represent preconditions for the first external system:

- **Ingress controller**: The single-page application and the API of the backend must be publicly available to be able to download and use them. For each public service, Ingress resources get created, which represent rules that an Ingress controller then implements. Consequently, the cluster needs an Ingress controller. According to the Cloud Native Computing Foundation (CNCF) survey from 2019, the Nginx Ingress controller is by far the most used one [61]. It is also the only open-source controller maintained by the Kubernetes team [62] and supports, among many other features, hostname- and path-based routing. Therefore this controller should be used for the solution to be developed.

- **Certificate issuing**: The TLS termination function of the Ingress controller relies on certificates that must be issued by a certificate authority. TLS termination encrypts the data traffic between clients and the Ingress controller (which acts as a reverse proxy for all services in the cluster). The cert-manager tool can automatically handle the administration, order, and renewal of certificates. It integrates well with Kubernetes' Ingress concept by watching the annotations of Ingress objects [63]. Further, it supports the Automatic Certificate Management Environment (ACME) protocol [64], which makes it possible to obtain certificates from Let's Encrypt – a certificate authority from which a user can order up to 50 certificates per week for free [65]. The cert-manager, set up with Let's Encrypt as the cluster issuer, is another precondition that the cluster must meet. The reason for this is because many features of modern web browsers require a secure context, which involves TLS encryption [66]. And anyway, it is a good practice to encrypt the data traffic.

- **Admission controllers**: The usual authorization layer of the Kubernetes API checks if a user is allowed to request a specific resource type. Admission controllers, on the other hand, represent an additional verification layer that determines whether a user is allowed to make a request with this particular specification. Further, they also react to incomplete specifications by completing them with default values specified by an administrator. That is especially helpful for compliance with the limits and quotas in which users have to stay. The upstream Kubernetes version activates all needed admission controllers by default [67]. But since downstream versions can customize this behavior, it must be checked that the following controllers are activated:

  - *LimitRanger* (Enforces limit ranges for pods)
  - *ResourceQuota* (Enforces compliance with namespace resource quotas)
  - *ServiceAccount* (Automates service account management)
  - *DefaultStorageClass (Adds storage class to persistent volume claims)*
  - *NamespaceLifecycle* (Rejects requests in non-existent namespaces and prevents system namespace deletion)

  - *AlwaysPullImages* (Deactivated by default. If deactivated, users can start pods using any image that is present on the node without authorization checks.)

- **Non-privileged containers**: The Kubernetes API server should always get started with the flag `--allow-privileged=false`. In the upstream version, this is set to false by default [68] to avoid privilege escalation. It is especially relevant in a multi-tenant environment.

- **Role-based access control**: Users of the self-service shall only get granted limited rights, so they stay inside of their namespace. The solution should be based on a role-based model and therefore requires the activation of role-based access control (RBAC). When starting the Kubernetes API server, you must activate it by passing the `--authorization-mode=RBAC` flag [69]. Depending on the Kubernetes distribution, the procedure can differ.

The second external system for which preconditions need to be elaborated is the GitLab server. The CI/CD pipeline must be enabled as the solution makes use of automated builds and deployments. That includes jobs (`builds_access_level=enabled`) [70], shared runners [71], and the container registry [72]. Since there are many ways to install these components [73], we will not go into detail here. The shared runners are needed for running containers in the pipeline. These, in turn, build the container images and load them into the container registry so that users can use them within the Kubernetes cluster. The advantage of using GitLab's built-in registry is that it is well integrated into the CI pipeline through environment variables and offers managed multi-tenancy functionality. Nevertheless, an independent registry could also be configured, for example, one that runs inside of the cluster.

Another requirement is that users can authenticate themselves via GitLab's OAuth interface while also authorizing the web application to create new projects on their behalf. Authorizing the web application to access user-based resources is essential. With the permission of a user, the web application can then use GitLab's API as if the user would use the interface himself. Therefore, the GitLab administrator must first register the web application and allow it to use GitLab's OAuth interface [74].

4.2 System Architecture

The requirements already specify that the solution should be based on a client-server architecture (see section 3.2). Now it is necessary to find and evaluate a suitable architecture style for the server-side application. Based on this style, we will then design the system architecture.

The solution to be developed should represent a service for students and, in general, software developers. A service-oriented architecture might seem to be appropriate. There are several alternatives to this style. On the one hand, there are the two extremes – a monolithic application based on a single consistent data model, and many loosely coupled applications, called microservices, communicating over HTTP. And, on the other hand, these two very puristic styles get complemented by the more pragmatic miniservices style (see Fig. 4.1) [75]. These miniservices are also referred to as modular monoliths because they inherit features from both extremes. Figure 4.1 illustrates that the loose coupling between individual miniservices is partially broken, for example, by some services accessing the same database. Another distinguishing feature is the differentiation criterion between different services. Miniservices get divided more coarsely based on individual business domains rather than individual business capabilities. Nevertheless, like microservices, miniservices communicate with each other via an HTTP API.

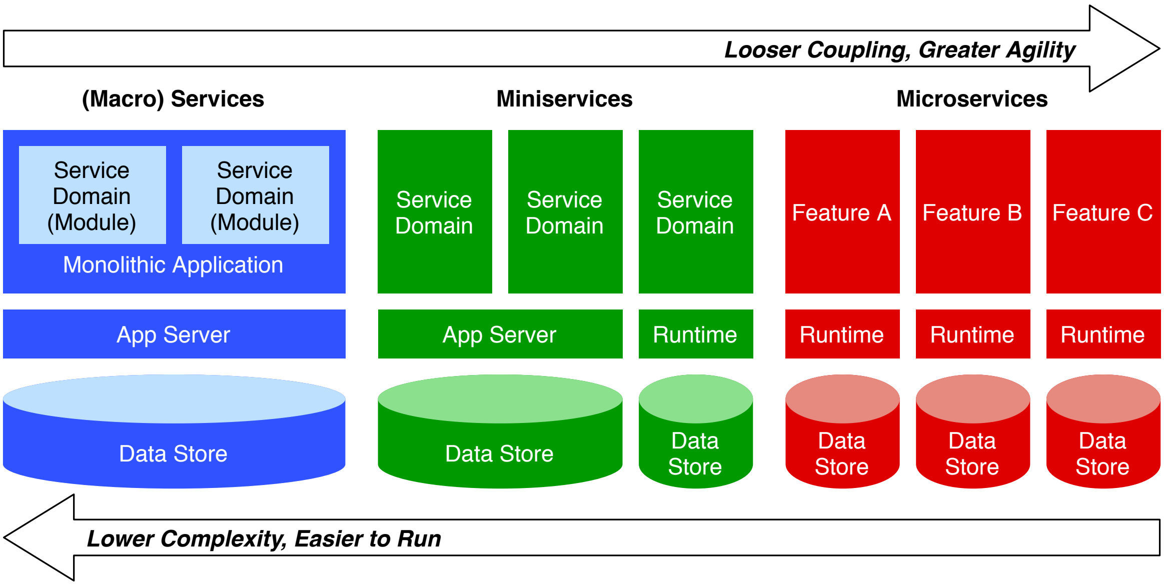


Figure 4.1: Comparison of different service-oriented architectural styles [75].

The question now is which architectural style is more suitable for the backend system of this bachelor thesis. To answer it, we will look at Domain-Driven Design. The concept is often mentioned in the context of microservices to define the boundaries between services [76]. It avoids designing systems solely based on technical requirements. That would otherwise lead to the division of responsibilities between different technical architectural layers. The layers combined express the model of a monolith, and the distribution of these layers is not desirable because it introduces a lot of complexity. Martin Fowler also defines this in his first law: “Don’t distribute your objects” [77]. In Domain-Driven Design, the goal is to break down the silos of technical experts and functional departments, and thus enable collaboration and agility.

A recommended method to identify and define individual aggregates is event storming. An aggregate is a way to draw clear vertical boundaries between services. It consists of several events that occur within a domain. The method first collects all actors. Each actor then triggers different events over time. An event can trigger additional events, which are typically restricted by policies that, in turn, depend on data. After the collection of all events, they can be associated with an individual aggregate [76]. Figure 9 shows the results of an event storming for the solution to be developed. In total, we can identify four aggregates. The solution must deliver the single-page application, interact with the Git repository provider (in this case GitLab), assemble the source code of the new project, and integrate it in a new virtual Kubernetes cluster. Each of them already gives an overview of their functionality since every event of an aggregate has to be processed. The aggregates now allow us to focus on the things that actually happen and not on data structures or other technical aspects. That is crucial for the discussion about which architectural style to choose.

Theoretically, a monolithic application covering all domains would be possible, as well as the separation into four microservices. However, it is noticeable that serving the SPA is completely independent of the other three domains. There is no policy between them that would automatically lead to subsequent events in the last three domains. Only the interaction of the user himself leads to events in the last three domains. Nevertheless, there are many automatic subsequent events for which the user is only implicitly responsible and which get handled internally. In other words, you can draw a distinct vertical line between serving the SPA and the rest of the domains. The clear distinction between the interaction with the Git repository provider, the assembly of the source code, and the integration into a virtual cluster, however, is not that obvious.

The idea is to apply a hybrid model consisting of a distinct microservice for serving the SPA, and a monolithic service. The latter should cover the requirements of the three remaining domains. This idea can be justified with the following factors:

- At the beginning of a project, the strict boundaries that usually separate microservices are still uncertain and unstable. They only establish themselves gradually over time, which is why it is worth starting with a monolith and breaking out individual microservices from it after a while [78]. Afterward, they can evolve independently and cope with new demands. That is also the opinion of Sam Newman, who is “convinced that it is much easier to partition an existing system than to do so upfront with a new one. You have more to work with. You have code you can examine, you can speak to people who use and maintain the system. You [..] have a working system to change, making it easier for you to know when you may have got something wrong or been too aggressive in your decision-making process. You also have a system that is actually running [and which you can use as a baseline from a performance point of view]" [79].

- The microservices architecture style was introduced to handle complex systems. However, the style also brings its own complexity. New challenges can include automated deployments, failure handling, concurrency, and eventual consistency. This overhead is also known as microservices premium [80]. Several architecture consultants and software engineers, including James Lewis, Sam Newman, Thiyagu Palanisamy, Evan Bottcher, and Martin Fowler, therefore claim that microservices should be considered when the system is too complex to be managed as a monolith [80]. In contrast, the idea proposes to develop only three domains within a monolith. Furthermore, the system should apply the self-service principle and will thus be based on only one actor, the user himself. Therefore, it can be said that the possible solution is not complex and that it should be easy to manage it in a monolith.

- One of the goals of microservices is to achieve higher agility through cross-functional teams, loose coupling, and the minimization of complexity within a service compared to the monolith [21] [81]. However, this argument does not make sense when working on this bachelor thesis since only one developer is working on the solution. There is no need to distribute business units or teams, and there is no need to work in parallel. The application should solve only one single, very concrete use case. The estimated implementation time is four weeks. Thus, the development phase does not extend over a long period of time, and there is no need to quickly, independently, and iteratively deliver new components and features in a large system.

- Probably the most influential argument for microservices is the good horizontal scalability. However, this is not relevant to the project, as the number of users can only be as large as the number of members of the organization offering the service. That is because each organization would deploy an independent instance of the service in its own cluster. Besides, the available cluster capacities would probably be exhausted before a high demand would overwhelm the service. Furthermore, the service is to be developed stateless anyway, which would also apply to a monolithic architecture.

- Monolithic does not necessarily mean that the domains and corresponding modules must be tightly coupled. As with microservices, the modules can be loosely coupled through well defined and stable interfaces. In particular, the fact that the service is supposed to be stateless makes loose coupling easier, since the modules do not have to share a common persistence model.

Figure 4.2 shows a system architecture that applies this hybrid model. Since all system components shall run in the Kubernetes Cluster, we will introduce the Kubernetes-specific components directly in the diagram (see the parentheses). As written in the chapter about the preconditions, an Ingress Controller is part of the system architecture. It acts as a reverse proxy between the separate services and the end-users. The separate services consist of the web server for serving the SPA, and the Scaffolder API server. The latter comprises the three aforementioned domains, which are to be developed together in a monolith. The Scaffolder application uses two external services: the GitLab and Kubernetes API servers. To be able to access these external services, the application needs the necessary rights. Therefore the whole system is located in a separate namespace for which a default service account with cluster-wide rights will get created. That means that, as long as the application runs inside of the namespace, it automatically has access to all resources in the cluster via the Kubernetes API Server. The namespace also contains a secret for the so-called "client secret", which is required for the OAuth protocol to request a token from the GitLab API server. In addition, the Scaffolder application connects to an in-memory database so that the service alone is truly stateless and can be seamlessly scaled and reinstated. The pod of the Scaffolder API server could also be instantiated via a horizontal pod autoscaler (a Kubernetes-specific component that automatically scales resources) and would then satisfy all cloud-native properties. However, this is not required at present.

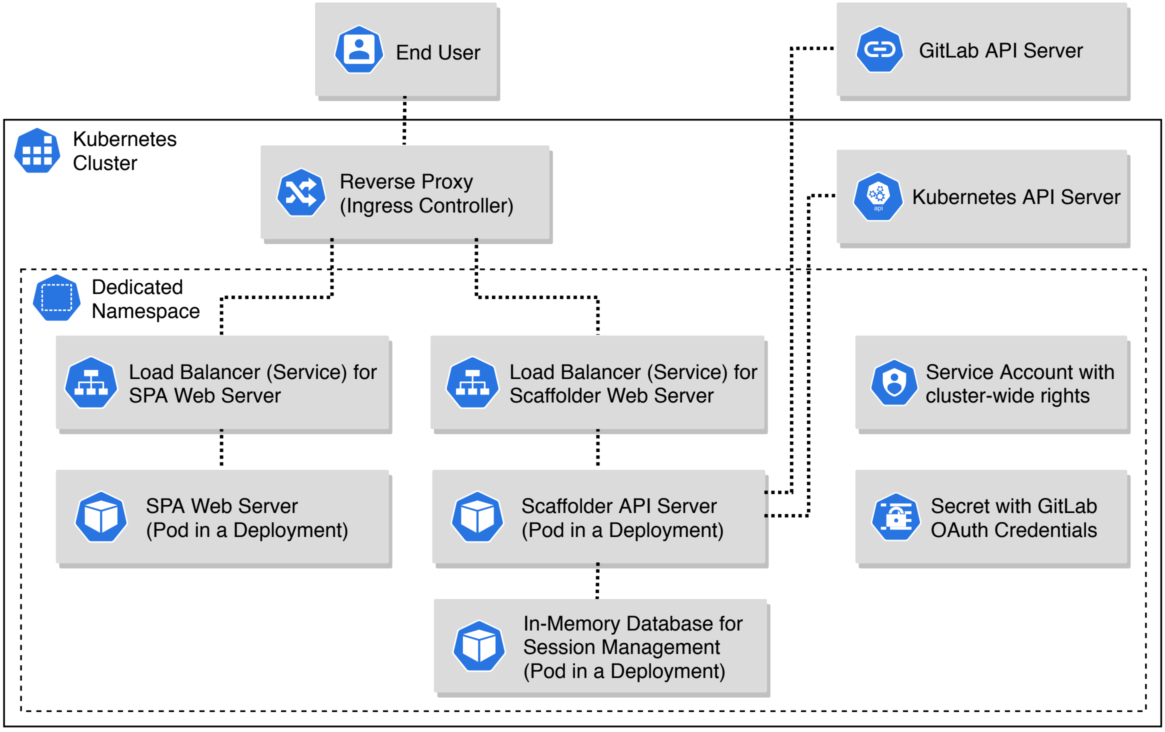

End User
GitLab API Server
Kubernetes Cluster
Reverse Proxy (Ingress Controller)
Kubernetes API Server
Dedicated Namespace
Load Balancer (Service) for SPA Web Server
Load Balancer (Service) for Scaffolder Web Server
Service Account with cluster-wide rights
SPA Web Server (Pod in a Deployment)
Scaffolder API Server (Pod in a Deployment)
Secret with GitLab OAuth Credentials
In-Memory Database for Session Management (Pod in a Deployment)

Figure 4.2: High-level overview of the system architecture.

4.3 Software Architecture

The software architecture essentially consists of two components. As shown in the system architecture, there are two containerized applications. That is, on the one hand, the user interface (frontend), which gets served by a web server, and on the other hand, the Scaffolder API server (backend), which implements the logic. Section 4.3.1 derives endpoints and tasks from the events of the event storming, and explains the project scaffolding of demonstration applications. Furthermore, section 4.3.2 presents a wireframe for the user interface.

4.3.1 Backend

The backend implements a web server that publishes specific endpoints that the end-user can request through the web app. A request to such an endpoint means that the backend has to perform a task. These tasks can easily be translated into events that have already been identified during the event storming. Therefore, these tasks get listed in table 4.1 below with the corresponding endpoints.

Method	Endpoint	Task(s)
GET	Check session	Check if session valid/available
GET	Login	Forward user to the GitLab login page
GET	Authorize	Request the access token from the GitLab API server; Save new session
GET	Logout	Invalidate session
GET	Check capacity	Check if the cluster has sufficient capacity for a new project
GET	Check name	Check if the project name and namespace is available

POST	Create project	Create new namespace;
		Configure new namespace;
		Create new GitLab project;
		Configure new GitLab project;
		Assemble the project files according to specification;
		Upload the project files

Table 4.1: Endpoints of the Scaffolder API server

“Check capacity” is not listed in the events. This task comes from a non-functional requirement that new projects should only be created if the cluster capacity allows it. Altogether, the tasks can be divided into three major categories. These categories are: managing the authorization, the project creation, and the creation and assembly of the source code for the demo application, which the user specifies during the creation process. The latter is also called scaffolding (hence the name “Scaffolder API Server”). It uses a template engine that generates source code based on templates and the user’s project specification. More on this in the paragraph about the project scaffolding.

Authorization through OAuth 2.0

GitLab offers three different ways for users to grant rights on their GitLab account to a client application [82]. Client-only applications can use the implicit grant flow, which issues an access token to the client immediately after authorization. Since the self-service solution is based on a backend server, this flow is not necessary. It has security risks anyway since the access token is stored directly on an end device. Another method is the “resource owner password credentials” flow. It allows the user to enter the access data (including the password) for the GitLab account directly in the client application’s user interface. With these credentials, the backend server can then request an access token. This method is the least secure method since the client application can attain any rights scope. Furthermore, it does not provide security features such as two-factor authentication. The recommended method is the authorization code grant flow. Figure 4.3 illustrates how it works. When the user tries to log in to the Scaffolder app, he or she gets redirected to the GitLab authorization page (via the login endpoint of the Scaffolder app). If the user then grants the Scaffolder app the rights to his GitLab account, the GitLab server redirects him to the “Redirect URI”[1]. That is, in this case, the authorization endpoint of the Scaffolder app. The task is then to use the code to request an access token for the GitLab API. Once the token is received, the Scaffolder API server can access the user’s resources.

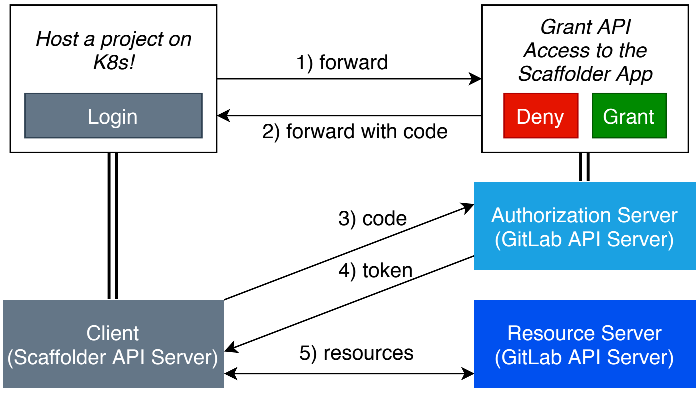

Host a project on K8s!
Login
1) forward
2) forward with code
Grant API Access to the Scaffolder App
Deny
Grant
3) code
4) token
Authorization Server (GitLab API Server)
Client (Scaffolder API Server)
5) resources
Resource Server (GitLab API Server)

Figure 4.3: Illustrating the OAuth 2 authorization code grant flow.

[1] As mentioned in the preconditions, the GitLab server admin must set this URI when registering the scaffolder app.

After the successful authorization, the access token must be stored in a session. For this purpose, the system architecture provides the in-memory database. It stores key-value pairs, each representing a session. The access token represents the value, which can be accessed by a randomly generated string as the key. This randomly generated string then gets set as a cookie in the user's browser. The browser sends the cookie with each request back to the Scaffolder API server so that it can identify the user.

Project Creation

Before the Scaffolder API server can create a new project, it must check two things. First, it must check if the cluster capacities allow the allocation of additional containers. That means that the number of active pods added to those that the new project would request must not exceed the maximum number of possible pods (of all cluster nodes together). Furthermore, the creation of a new project must not exceed the physical cluster resources (e.g. CPU, memory). This task gets executed when the client requests the "check capacity" endpoint. And second, the app has to check if the desired name is still available – both for the namespace and as GitLab project name. Normally, the check of one of them would be redundant, because it would fail anyway if the server requests the creation for one of them. But since the name is to be checked directly while the user types, there must be the additional endpoint "check name".

Once these two criteria have been checked, the project generation can start by sending a request to the "create project" endpoint. The following list describes all tasks to be executed:

- The endpoint represents a POST method and therefore receives a body. The content (project name and specification) thus must be parsed.
- The Scaffolder API server must each send a request to the Kubernetes API server to
  - create a new namespace,
  - apply a limit range for new objects in that namespace,
  - apply resource quotas that get enforced in that namespace,
  - create a restricted role in that namespace,
  - associate that restricted role with the default service account of that namespace,
  - and retrieve a list of all secrets in that namespace (including the token and certificate of the default service account).

- In order for the user and the CI pipeline to access the namespace, the Scaffolder API server must generate the credentials before. It does that by decoding the base64 encoded secret of the default service account and creating a kubeconfig file from it.
- The Scaffolder API server must each send a request to the GitLab API server to
  - create a new project,
  - set an environment variable in that project with the content of the kubeconfig file so that the CI pipeline runners can gain access to the namespace,
  - and create a project-dependent deployment token that can be used to read the project's container registry.
- The demo applications use the CI pipeline to automatically build containers from the source code, which then get pushed into the container registry. In order for the container puller (in the Kubernetes cluster) to have access to this registry, it needs a secret containing the access data. The scaffolder API server thus requests the Kubernetes API server to create a secret within the created namespace. This secret contains the previously created deployment token.
- Depending on the project specification, another secret has to be created. If the user wants a database, the Scaffolder API server has to generate a randomly generated password for this database and store it in the namespace. That allows the demo application to retrieve the database password via the environment variables.
- The Scaffolder API server must assemble all source code files based on the desired project specification. More on this in the next paragraph.
- The final step is to commit all source code files to the Git repository. The Scaffolder API server must convert the files to a JSON payload and then *POST* them to the GitLab API server.

Project Scaffolding

One of the main goals of the solution to be developed is to flatten the learning curve of Kubernetes by introducing typical Kubernetes workloads based on a simple demo application. The idea is to let the user select his desired technology stack by three layers. Depending on what the user needs, he can then easily choose from components he is already familiar with. This way, the user does not have to familiarize himself with Kubernetes-specific terms before even starting a project.

When defining the layers, the challenge is to find a use case that is simple, but at the same time can cover all typical Kubernetes workloads and can also be completely customized from a technology perspective. That also means that the user can exclude individual layers that he does not need. A possible use case could be a simple CRUD application (CRUD stands for create, read, update, and delete), which allows altering a list of not further specified names. The user could then select his desired components from the following three layers:

1. **User Interface**: The user interface layer implements a web app that allows users to alter the list in a web browser. It accesses the list via the application's HTTP API.
    - VanillaJS (Static website)
    - Vue.js (Single-page application)
    - React (Single-page application)
2. **Application**: The application layer implements a web server that exposes an HTTP API with endpoints to alter the list. This layer is mandatory.
    - Java (with Spring Boot)
    - Python (with FastAPI)
    - Node.js (with Express)
3. **Data Storage**: The data storage layer provides a storage solution to save the list. By default, this storage is ephemeral, but the user shall have the option to select persistence so that the data gets stored in a persistent volume. If the user excludes this layer, the application will store the list in a language-based in-memory data structure. In that case, the application layer would be stateful.
    - MySQL (Relational database)
    - MongoDB (NoSQL, document-based database)
    - Redis (NoSQL, in-memory data structure store)

With the help of the above-mentioned layers, it is possible to cover the Deployment and StatefulSet workload. Furthermore, they allow the demonstration of typical CI/CD examples. The missing job and CronJob workloads can be covered by two additional use cases. For example, the user could select that the names list should be cleaned up automatically (**Cleanup layer**). A CronJob then runs on a specific schedule and shortens the list to a certain number of entries. In addition, there may be another endpoint in the application that allows it to start a job that adds many names to the list (**AddMany layer**). This endpoint can also be accessed via a button in the user interface. When the user presses the button, a job gets executed, and n entries get added to the list. This job may not be a typical use case for jobs because it could also be executed synchronously and is not necessarily CPU-bound. However, it should be sufficient to demonstrate the job workload, as it shows how to define job objects and execute them within an application via the Kubernetes API server. This way, the simple use case of a CRUD app can cover all workloads from the requirements.

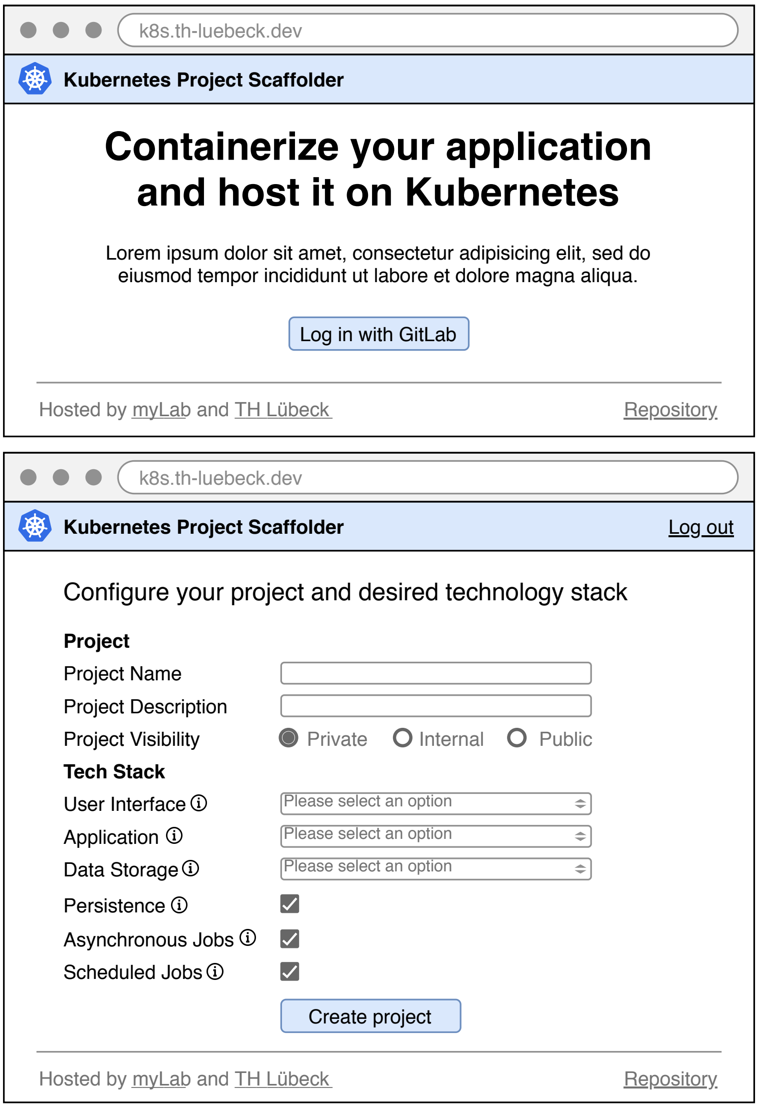
The well-defined HTTP interface of the application layer allows for easy exchange of the web app within the user interface layer. Each web app implementation always accesses the same interface. At the data storage layer, this is less easy because each database exposes its own interface. The database cannot be exchanged without adapting the source code of the application. This could only be avoided if an additional service gets placed before the respective database. It would abstract the database protocol and expose a unified HTTP interface. If we choose a specific programming language in which this service should get implemented, it could significantly reduce the amount of work involved. However, this has no added value for the user. On the contrary, it would introduce unnecessary concepts and would not allow the user to access the database directly from the application. Therefore each implementation at the application layer must implement an individual connector for the database. That leads to an increased workload for the author but is then easier to comprehend for the user. The same applies to starting jobs from within the application, which must get implemented for each application individually.

After implementing and testing the individual technology stacks, the source code must get converted into templates that a template engine can process. That means that the Scaffolder API server has access to a folder where all source code templates are stored. It can then use the template engine and project specification to generate an individual demo project for a user.

4.3.2 Frontend

The frontend shall be based on a single-page application (SPA) that communicates with the Scaffolder API server via HTTP requests. SPA frameworks typically already provide a general project structure. In this case, the author will use Vue.js as the SPA framework as he has the most experience with it. In addition, a component-based CSS library will be used to minimize the development time for the layout and design. The following wireframes in figure 4.4 show how the user interface requirements will get covered.

Although it is called a “single-page application”, the user interface consists of several pages. As already explained, only one page gets loaded initially, and then the DOM gets altered by the JavaScript code. Depending on whether the user is logged in, the app will show the corresponding page. If not logged in, the app will show the welcome page. This page includes



k8s.th-luebeck.dev

Kubernetes Project Scaffolder

Containerize your application and host it on Kubernetes

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Log in with GitLab

Hosted by myLab and TH Lübeck   Repository

k8s.th-luebeck.dev

Kubernetes Project Scaffolder   Log out

Configure your project and desired technology stack

Project

Project Name

Project Description

Project Visibility ● Private ○ Internal ○ Public

Tech Stack

User Interface ⓘ Please select an option

Application ⓘ Please select an option

Data Storage ⓘ Please select an option

Persistence ⓘ ☑

Asynchronous Jobs ⓘ ☑

Scheduled Jobs ⓘ ☑

Create project

Hosted by myLab and TH Lübeck   Repository

Figure 4.4: User interface wireframes of the Scaffolder application.

a short explanation about the service and a button that allows the user to log in with their GitLab account and authorize the Scaffolder application to create a project. Once the user is logged in, the project configuration page appears. It allows the entry of project-specific details and the specification of the previously discussed layers. The info icons next to each layer are used to display tooltips. As soon as the user hovers the mouse over them (or clicks, if on mobile devices), a description and a hint in which use case the layer can be useful gets shown. The first three layers use a dropdown list instead of radio buttons. This way, the offered technologies can easily get extended in the future without changing the user interface. Also, it allows the user interface to request a list of possible options from the backend server and adapt its dropdown list accordingly.

5 Implementation

As described in the architecture chapter, the challenge is that some of the technologies are still unknown, and requirements and decisions need to be validated as the work progresses. Therefore, an agile way of working was chosen for the implementation phase. That means that individual work packages were derived from the requirements for each week. However, these work packages are only specified in their functional requirements by the architecture chapter. The reason for this is because it represents a substantial risk to put too much time into the exact software design, only to find out during the implementation phase that the decisions were wrong and that it is not realizable in such a way. The individual software components are interdependent, and therefore it is wise to evaluate implementation decisions for each work package as needed and validate them with the start of the next work package. The same applies to the initial requirements. In the course of the work, it has become apparent that requirements have to be adapted. Therefore, the purpose of this chapter is to describe the actions of the last weeks and to discuss the decisions made.

5.1 Project Structure

The project consists of a mono repository that contains the source files for the backend, frontend, and the Helm chart. Since these areas were each implemented in different languages and frameworks, they get examined individually and independently in the following sections.

5.1.1 Backend Services

The requirement for the backend was that it should be implemented in Python for better integration into the current system landscape of the university. In the beginning, the choice of the web framework was up for discussion. There are two popular frameworks in Python – on the one hand the full-stack framework Django and on the other hand the microframework Flask [10]. The difference between full-stack and microframeworks is that the full-stack frameworks offer significantly more features out of the box. That means that they offer, among other features, the Model-View-Controller pattern with server-side templates and object-relational mappers with database connectivity. However, since the Scaffolder API is not intended to render dynamic pages, but only to provide a JSON-formatted HTTP interface, we do not need the large feature set of a full-stack framework. In the end, the author chose the microframework *FastAPI*. Its API is much inspired by Flask, but additionally, it offers numerous advantages. As the name suggests, the resource usage is lower. It also provides request and response validation, OpenAPI documentation generation, and improved IntelliSense (respectively auto-complete and typing)[1].

The first step was to implement the API controller. The backend specification in chapter 4.3.1 already provides the HTTP endpoints. Table 4.1 thus results in the following endpoint handlers:

Listing 5.1: The API controller implementing the HTTP endpoints. (Backend/scaffolder/-main.py)

```
app = FastAPI()

@app.get("/api/login")
def login():
        ...

@app.get("/api/auth")
def authorize(state: str, code: Optional[str] = None):
        ...

@app.get("/api/check-session", response_model=responses.CheckSession)
def check_session(response: Response, sid: Optional[str] = Cookie(None)):
        ...

@app.get("/api/logout")
def logout(sid: Optional[str] = Cookie(None)):
        ...

@app.get("/api/options", response_model=responses.TechStackOptions)
def options():
        ...

@app.get("/api/check-capacity", response_model=responses.CheckCapacity)
def check_capacity():
```

[1] https://fastapi.tiangolo.com/ (visited on 12/20/2020)

```
        ...

@app.get("/api/check-name", response_model=responses.CheckName)
def check_name(name: str, sid: Optional[str] = Cookie(None)):
        ...

@app.post("/api/project", response_model=responses.ProjectDetails,
                                                        status_code=201)
def create_project(spec: requests.ProjectSpecification,
                                        sid: Optional[str] = Cookie(None)):
        ...
```

For better readability, the above listing omits the imports and concrete implementations of the controller methods. These methods do not implement any functional requirement themselves but instead call functions from service modules. The authorization and scaffolding functionalities were outsourced to these service modules. They are located in the `/scaffolder/services` directory and are imported as singletons. Although, strictly speaking, they do not fulfill the singleton pattern as known from the gang of four. But since modules in python get initialized only once, modules always reference to the same imported module object [83]. However, the API Controller is the only module that imports the service modules and thus serves as the link between the modules. Specifically, this means that, for example, the `check_name()` endpoint handler must use the appropriate service modules to check whether a user is logged in and whether the project name is available in both the user's GitLab account and the cluster.

Authorization and Session Management

The first important service implements everything related to authorization and session management. The module is located in the `/scaffolder/services/auth.py` file. Once imported by the API controller, it initializes two thread-safe Redis database connections. Application states that have a retention time longer than a single request must be implemented thread-safe. The reason is that the API controller is run by an "Asynchronous Server Gateway Interface" (ASGI) server, which calls the handlers in concurrent threads. Back to the Redis connections. The first connection is made with virtual database number 0 and is responsible for holding the sessions. In this case, the module only uses Redis' hash map to store the session IDs and the associated GitLab API access credentials. The second connection to virtual database number 1 stores randomly generated strings, which only serve another security aspect during authorization. All keys, both in database 0 and 1, have an expiration time, which is defined in the `config.py` file and can automatically be set via environment variables. For the sessions, the expiration time gets reset with each user interaction. The keys with the random strings, on the other hand, expire after a fixed time without exception. Without the context of the individual module methods, the purpose of these Redis connections may now be somewhat obscure. Therefore, the following OAuth 2.0 flow example provides an overview of the crucial methods.

Once a user presses the login button, the browser requests the `login()` endpoint, which in turn calls the `auth.get_login_redirect_url()` method to retrieve a new redirect URL. This URL contains information so that the GitLab OAuth API knows which application requests access rights to the user's account. Additionally, it contains a randomly generated string called "state". When redirecting the user back to the Scaffolder application, the GitLab API includes the same state parameter to the authorization redirect URL so that the Scaffolder API can check whether this string exists in the database. The OAuth flow implements this approach to prevent cross-site request forgery attacks [84]. Without the state parameter, any other website (or attacker) could potentially trick the user to request the authorization endpoint and thus changing the application's state.

After the user has successfully authorized the Scaffolder app in the GitLab user interface, GitLab redirects the user to the authorization endpoint of the Scaffolder API. The `authorize()` endpoint uses the `auth.is_state_param_present(state)` method to check if the state parameter is present and then creates a new session. For this purpose, the module implements the `auth.get_new_session(code)` method. The code parameter that gets passed is a one-time code to request an API token. The method then requests a new API token for the user from the GitLab API using the code, app ID, and app secret. If the request is successful, the method creates a new random session ID and stores the user's API credentials in a session (with the session ID as the key and the credentials as the value) in the Redis database. Finally, the endpoint sends a response, including a session cookie (with `sid` as the key and the session ID as the value) and a redirect to the Scaffolder's home page. The corresponding session cookie only allows connections that are originating from the same domain (*SameSite=Strict*) and that are secure (*Secure* flag, meaning only TLS encrypted connections). Moreover, it cannot be queried or manipulated by the script since browsers only include it in the HTTP request when making requests to the origin server (*HttpOnly* flag).

When the user returns to the Scaffolder's home page, the SPA automatically sends a request to the `check_session()` endpoint. The `auth.is_session_valid(sid)` method then checks if the session ID, which the request includes as a cookie, is valid. The method fetches the user's API credentials from the database and verifies that it can retrieve the user data via request to the GitLab API. If the user data is successfully retrieved, the Scaffolder API replies with the corresponding `CheckSession` response, which includes the user name.

The user interface can now display that the user is logged in by writing the name into the navigation bar. Additionally, that also allows displaying a log out button. In case the request to the GitLab API was not successful, the Scaffolder API deletes the user's cookie when sending the negative `CheckSession` response. Another case when the cookie gets deleted is when the user logs out. In this case, the user sends a request to the `logout()` endpoint. It calls the `auth.delete_session(sid)` method to delete the session from the database and then redirects the user back to the home page with an invalidated cookie.

Communicating With The GitLab Server

After the user logged in, he can fill out the form to specify a new project. As soon as the user enters the desired name of his new project, the SPA automatically sends a request to the Scaffolder API to check the name's availability. For this, the Scaffolder API controller implements the `check_name()` endpoint, in which it needs to communicate with the GitLab API server and the Kubernetes API server. The corresponding module to communicate with the GitLab server is located in the `/scaffolder/services/git.py` file. The `git` module implements the method `is_name_available(access_token, name)`, which requests a list of all user projects and then compares the existing names with the desired name. An essential point to mention here is that this method is the only one that uses the Python wrapper for the GitLab API. The reason is that the number of user projects can be relatively large and the GitLab API thus returns the projects only in segments. The wrapper provides automatic pagination so that the method can easily iterate through all projects. However, the wrapper has the disadvantages of not offering IntelliSense and not allowing project-specific method calls via a project ID like the GitLab API, so it was excluded for further use in the other methods. Instead, they simply call the HTTP endpoints of the GitLab API. In hindsight, this decision can surely be questioned. However, the GitLab API is well documented so that the author sees no disadvantage in using the project endpoints directly without API wrapper.

The next step of the user is to create the project. The web app sends a request with the appropriate project specification to the `create_project()` endpoint. The endpoint handler is responsible for creating a GitLab project, a Kubernetes namespace, and the source code for the demonstration app. It does that by simply calling the methods of each service module. For creating the new GitLab project, the `git` module implements the `git.get_new_project()` method. When the handler calls it, it passes the project specification, the user's GitLab API credentials, and the cluster details (Kubeconfig of the service account and the namespace) as arguments. The `git` module then creates a new GitLab project on behalf of the user, two new environment variables for the CI/CD runners (respectively `KUBECONFIG` and `K8S_NAMESPACE`), and a deployment token to allow the cluster to pull container images from the project-specific container registry. After the handler receives the source code of the demonstration application, it calls the `git.commit_and_push_files()` method to upload the files to the repository. In turn, this method sends all the files as a list of JSON objects to the GitLab API server.

Communicating With The Kubernetes Cluster

As soon as the Scaffolder web app gets opened by the user, it asks the Scaffolder API whether the cluster has enough resource reserves available for a new project. If this is not the case, the frontend informs the user and does not allow any project creation. When the corresponding `check_capacity()` endpoint gets requested, the handler calls the `cluster.is_capacity_sufficient()` method of the `cluster` module (`/scaffolder/services/cluster.py`) and asks if at least 5 percent of the cluster capacity is still available. The implementation is a bit more complex because there is no endpoint for requesting the overall resource usage. Therefore, the capacity must be queried for each cluster node in order to offset the summed capacity with the consumption of all pods in the cluster. An open-source script was used for the implementation (see the docstring of the `cluster.get_allocated_resources()` method for more information). The method only considers the requested resources, not the limits. Since individual applications are presumably rarely used at full load, it should also be possible to overcommit resources. Thus, more students could use the cluster and exceed their requested resources for a moment if needed.

As described in the previous paragraph, when the user types in the desired project name, the `check_name()` endpoint checks whether the project name is still available. Since namespaces in the cluster shall be based on the project name, their availability must also be checked just like in the `git` module. Therefore, the `cluster` module implements a similar `is_name_available(name)` method, which checks whether the desired namespace is available in the cluster. Also, the method makes sure that the namespace does not start with `kube-` since these are reserved for system-internal namespaces.

Kubernetes namespaces must always meet the criteria of a fully qualified domain name (FQDN) since pods include the namespace in their full network address. However, since user input does not necessarily fulfill domain name constraints, the method still has to convert the passed name into a so-called slug. The idea is that each new project gets a namespace that corresponds to the slug of its project name. The namespaces must be unique and can therefore be used as a subdomain for the ingress rules. A slug removes all special characters and replaces them if necessary. Spaces get filled with hyphens, and the string gets written in lower case. For example, the string “Lübecker Straße” would be converted to “luebecker-strasse”. However, the conversion to a slug is not only needed here but also by other modules. Therefore, a utility module (`/scaffolder/services/util.py`) with the `get_slug()` method has been implemented to ensure consistency. In addition, this module also implements the `get_random_str()` method to generate random strings. It is needed, for example, when generating the previously mentioned state parameter, a session ID, or database keys.

The previous section also mentioned that, in order to create a GitLab project, the Kubeconfig file must be passed to the appropriate method. Only then the CI/CD runners can execute commands in the namespace. But for that, the namespace has to be created and configured in the first place. For this, the `cluster` module implements the `get_new_namespace()` method. As arguments, it receives the namespace's name and, to be able to associate the namespace to a user for administrative purposes, the email address of the user. It then creates a new namespace by rendering a YAML-formatted template and applying it to the cluster (comparable to the apply command of the Kubernetes CLI). The configuration of the namespace consists of applying resource quotas (`ResourceQuota`), limits to the value ranges of resources (`LimitRange`), a role with restricted rights (`Role`), and a role association to the default service account of the namespace (`RoleBinding`). After the method successfully created and configured the namespace, it returns the content of the Kubeconfig as a string.

There are two ways to programmatically create individual Kubernetes resources in the cluster. The first way is to initialize Python objects using modules from the `kubernetes.client.models` package. Especially for large resource specifications, this bloats the code a lot, but again gives the possibility that you can insert variables in the resource specification. The other approach is using the `yaml` module provided by the Kubernetes API wrapper. It allows YAML-formatted strings to be converted to a resource specification and created directly in the cluster. It also allows the conversion of multiple related resources from a single string. This alternative feels more natural in use and increases the readability. The downside is that you can only convert static strings with it. Since the Scaffolder application depends on a template engine anyway (to create the demonstration application), it can also be used in this module to dynamically insert values in the YAML resource. Therefore, the author decided to create the resources using only templates, which are rendered and then converted by the `yaml` module. All templates used by the `cluster` module are located in the `templates/namespace_creation` directory. Jinja was chosen as the template engine. The reason for this is that the templates should also support logic to adapt the demo application's source code to the user's needs. That leads to a stricter separation between the templates and the Scaffolder's source code. For example, a user could want to run asynchronous jobs and thus the demo application's template must add the corresponding part of the source code when rendering. Due to the logic support, the popular engines Mako and Jinja remained. However, unlike Mako, Jinja offers a syntax that is more similar to the Go `template` package. Since the Helm charts are also based on the Go syntax, Jinja was ultimately chosen.

As mentioned, keys get generated during the creation of a GitLab project and the associated source code. Among them are, for example, the access data for the container registry and, if specified by the user, the database credentials. The `cluster` module must now store these as secrets in the user's namespace. Since the types of secret resources and correspondingly their resource definition differ, there must be different methods for creating secrets. The `set_registry_credentials()` method creates a secret of the type `kubernetes.io/dockerconfigjson`. It renders the `registry_credentials_secret.yaml` template by inserting an encoded `dockerconfig` into the template. The private `__get_docker_config_json()` method creates the required string by rendering the `dockerconfig.json` template using the user's registry credentials. The `set_secret()` method, on the other hand, is simpler, as it only creates a secret of the type `Opaque`. The associated `secret.yaml` template receives only a name, a key name, and a key value and then gets applied to the cluster.

Creating Demo Applications

When the user opens the web app, it automatically asks the Scaffolder API for the options available to the user. The Scaffolder API controller implements the `options()` endpoint, which simply calls the `scaffolder.get_tech_stack_options()` method. It returns all possible technology stack options. That allows the backend to be developed and extended independently of the frontend.

However, the main purpose of the `scaffolder` module is to generate the source code for the demonstration application. The templates for all applications, Kubernetes manifests, and other source code files are located in the `/templates/demo_application` directory. The `get_source_code_files()` method creates a list of files that each get stored in a dictionary in the form of `{ "file_path": str, "content": str }`. In addition, if the user wants a database in his stack, it also creates the credentials for the database, which get stored in another list of secrets. The caller of the method, in this case the endpoint handler, receives both lists. It thus is responsible for creating the secrets and uploading the source code to the repository.

But the said method is not responsible for the rendering of the templates itself. Instead, it creates filter functions, which depend on the user's requirements, and then calls the `__generate_files_from_templates(filter)` method. This, in turn, renders only the templates that fulfill the filter. The filter gets applied to the directory path of a template. To understand how a filter function works, you need to know the directory structure of the templates. The `templates/demo_application` directory contains folders for the individual components that the user will later find in his project repository. For example, these are folders for the application, the user interface, the cron job, and the Kubernetes manifests. In each of these folders, equivalent to the names of the technology options, are subdirectories that contain the source code for all options. Accordingly, the filter functions must filter for these subdirectories and delete the specific technology option name from the directory path. An example of a user selecting a project with `go_fiber` as the option for the application layer would be that the filter function includes the `/application/go_fiber` subdirectory but excludes all other subdirectories within the `/application` path. The resulting path of each source file then omits the technology option. For example, the template path `/application/go_fiber/main.go` results in the source path `/application/main.go`.

Another challenge is to adapt the demo application's source code to the corresponding database. For this purpose, each application template contains a `data_access` folder in which the source code files for the individual database connectors are located. Accordingly, the filter function for the application source code got extended for the `data_access` directory. For example, if the user selects `mysql` as database, only the `mysql.*` file gets rendered. This extension was possible for all templates except the `java_spring` template. Since the Spring framework is very opinionated in the choice of the directory structure and the implementation of the so-called "data repositories", simply too much source code had to be adapted for the respective database connector. A single template for the `java_spring` option would involve a lot of logic in the template and the Scaffolder's source code itself. Also, the template's directory structure would change with different database connectors. The author thus decided to split it into four independent templates for better maintainability and readability.

Additionally, the `scaffolder` module implements the `save_source_code_files()` method, which is used during development. If the `IN_CLUSTER` environment variable is not set, then the application assumes it is in development mode. If that is the case, the application will not create a new GitLab project and namespace but save the source code in a new directory on disk. That allows developers to work on templates faster and iteratively without wasting resources each time.

Containerizing The Scaffolder API Server

The Scaffolder app should run directly inside of the Kubernetes cluster. Therefore, a container image must be created beforehand. The Dockerfile, as described in chapter 2.1, provides the blueprint for the image. Each time the source code gets updated and pushed to the project repository, the GitLab CI pipeline builds the container image of the Scaffolder app according to the `/Backend/Dockerfile`.

The notable features of this Dockerfile are that it locks the base image version to a `major.minor` version and does not use `latest` as the image tag. It ensures that there are no breaking changes in the Scaffolder's base image that can cause problems. Also, the source code gets copied to the image in two separate steps. First, the `requirements.txt` file gets copied, which contains all dependencies. Afterward, these get installed by the `pip install` command, and accordingly, as explained in chapter 2.1, the layer gets cached. Only then the actual source code gets copied. Since the source code changes much more frequently than the dependencies, this layer caching principle saves a lot of build time. Another best practice to further minimize the risk of privilege escalation is to run the application inside of the container only as a non-root user. Therefore, the Dockerfile includes a command that creates a new user. It affects the way how files should get copied into the container image, which is why the `--chown` flag is now set in the `COPY` command.

Unlike the example from Chapter 2.1, the `ENTRYPOINT` command is not used. That is because the Scaffolder app is not an executable program for the command line. With the `ENTRYPOINT` command, one usually specifies an executable program and then uses the `CMD` command to specify its default parameters. Therefore the author decided to use the `CMD` command in the `exec` form (meaning as an array with an executable as the first entry). The respective `CMD ["uvicorn", "scaffolder.main:app", "--host", "0.0.0.0", "--port", "8080"]` command specifies to start the web server as the first process in the container.

5.1.2 Frontend Web App

The single-page app, on which the frontend is based, uses two key frameworks. The first is the JavaScript framework Vue.js[2], which can be used to build component-based user interfaces. Together with the Vuex[3] state management library, it implements the model-view-viewmodel pattern. The Vuex store represents the model and implements the connection to the backend. Each component consists of a script (`<script/>`; representing the modelview) and a Markup template (`<template/>`; representing the view), which binds the data from the script. However, since this work is not about a frontend framework, this section will only explain the choices made about the components themselves. As the second framework, the choice fell on Bootstrap. It is a component-based CSS (Cascading Style Sheets) design system. Developers can use it to build websites quickly, as they only need to select the components and slightly adjust them if necessary. It also offers a direct grid system, which

[2] https://vuejs.org/v2/guide/ (visited on 12/21/2020)

[3] https://vuex.vuejs.org/ (visited on 12/21/2020)

enables responsive websites – that means the automatic adaptation to the screen size. For Vue.js there is the library Bootstrap-Vue[4], which is used in this project as an integration of Bootstrap within Vue.js. It provides native event and property binding for the Bootstrap components.

State Management

Before discussing the individual components, here is a brief explanation of how the web app manages its state and communicates with the Scaffolder backend interface. Continually passing properties from one component to another nested one is considered a bad practice. That is because this requires to synchronize the copied state between components via events. One solution is to make use of the store pattern, which essentially supplies each component in the tree with the same instance of an observable store. In this case, the `store.js` file exports the store object, which then gets imported by the `main.js` file (see listing 5.2). Setting the `store` property of the `Vue` instance makes the store available globally as a singleton. The store instance thus gets injected into each component. These components can then dispatch "actions" and commit "mutations". That leads to better maintainability and gives the ability to observe and manipulate a unified state via development tools.

```
import store from './store'

new Vue({
  render: h => h(App),
  store: store
}).$mount('#app')
```

Listing 5.2: Code snippet from the main file of the frontend demonstrating the store injection. (main.js)

In total, the state of the store comprises twelve variables, most of which are self-explanatory. The only variable that may need to be explained at this point is the `showCreateProjectPage` variable. Normally, in larger projects with many different pages, you would implement a router. It allows the use of different URLs and navigation across multiple pages. However, the Scaffolder's SPA only consists of a welcome page (`<Welcome/>`) and the page where the user can specify his project (`<CreateProject/>`). Therefore, the author chose a pragmatic solution and solved the navigation based on a state variable. The app switches between the two pages via the `v-if="showCreateProjectPage"` directive (see `/src/components/App.vue`).

[4] https://bootstrap-vue.org/docs (visited on 12/21/2020)

What is more important are the three actions that the store implements and that components dispatch. Using the axios HTTP library, each action sends one or more requests to the API server. Depending on the response, the actions commit the appropriate mutations to change the state. The `INITIALIZE_APP()` action checks the user session and cluster capacity, and queries the available technology options. How the resulting data gets processed will be discussed below in the context of the particular component. The `CHECK_PROJECT_NAME_AVAILABILITY()` action uses the `name` parameter to check if the name is still available to the user. The third and final one is the `CREATE_PROJECT()` action, which sends the project specification to the API server. After the successful project creation, it commits the mutation of the `createdProjectUrl` so that the respective component can display a link to the new project. In addition, the store actions allow incrementing a load counter for each request to the API server. When an API request gets completed, the counter gets decremented again. Also, the `catch()` blocks catch any errors and then execute the `SET_ERROR()` mutation so that components can notify the user that something went wrong.

Components

A great advantage of component-based frameworks is that you can easily reuse the individual components. That can pay off even in smaller projects like this one since, for example, the form components repeat themselves. In addition, the framework allows you to bind the entered data directly in the script without having to work inconveniently with query selectors. All components lie in the `/src/components/` directory. The `App.vue` component represents the root component of the SPA since it comprises a div with the ID `app` as its topmost element. When building the SPA, this div gets injected into the `/public/index.html` page. The `App.vue` component's script implements the `mounted()` method, which automatically executes once the component is loaded into the virtual DOM. It dispatches the previously mentioned `INITIALIZE_APP()` action. Depending on the state, the component displays the `Welcome.vue` or the `CreateProject.vue` component. Also, it shows the `Footer.vue` and `NavBar.vue` components in any case. The latter receives the login status from the store and, if applicable, the username. The NavBar component can then display the username and a logout link to the logout endpoint of the API server.

The `Welcome.vue` component only displays text and a button. The latter can take different forms depending on the state. After the `App.vue` component dispatched the initialization action, the page is in a loading state until the requests have been answered. The `Welcome.vue` component represents this state by using the `b-spinner` component inside of the button. After loading is complete and there was no error, the component either displays the "Log in with GitLab" or "Configure Your Project" button. It can also happen that it shows an infobox instead of the button. That is always the case if an error has occurred or the cluster capacity is no longer sufficient.

The `CreateProject.vue` component lets the user specify the project as well as the technology stack. The page mainly consists of an input form, a checkbox to confirm the terms of use, and a button. In addition, it implements a so-called modal that displays the terms of use in an overlay and an output field that informs the user about the project creation. The following table 5.1 shows the implemented components on which the input form relies.

Component	**Input**	**InputSelect**	**InputCheckbox**
Input Type	Text	Dropdown List	Checkbox
Properties	label value placeholder valid debounce	label value tooltip options	label value tooltip link
Events	input	input	input

Table 5.1: Individual components for the CreateProject page.

What stands out is that all of them implement and emit the input event when the user makes an input. Also, each component includes the value property. The reason is that components like the `CreateProject` page can now bind a local variable to, for example, the `Input` component using the `v-model` directive (see listing 5.3). Accordingly, both values are always synchronized. Other notable properties of the `Input` component are the `valid` and `debounce` values. The former is used to indicate that the desired project name is not available anymore. If this value is not true, the Input component will display its text area with a red border and give a brief hint that the name has already been taken. The `debounce` value ensures that the input event is not emitted immediately with every character input but with a delay. In the listing 5.3, the component emits the event as soon as the user has not entered any characters for 800 milliseconds. That reduces the number of requests to the API server and, accordingly, the number of requests to the GitLab and Kubernetes API servers.

```
<Input
  label="Name"
  placeholder="Your project's name"
  :valid="validProjectName"
  debounce="800"
```

```
    v-model="projectName"
  />
```

Listing 5.3: Using the Input component within the CreateProject component's form. (CreateProject.vue)

Containerizing The Web App

Similar to the backend API server, the web app also needs to be containerized so that it can run inside of the Kubernetes cluster. The difference is that the Dockerfile will now rely on a multi-stage build. That means that the Vue-CLI[5] builds the static artifacts first, and then these are copied to a second container image that provides a web server. In the end, the resulting container image will be much smaller because it only builds upon the second image. That is why one should try to find the smallest base image for the second build stage. The author has chosen Nginx as the web server. The reasons are its popularity, simplicity, and availability of a small container image (namely the version based on Alpine Linux[6]). Copying between the individual build stages works by naming these stages. In this case, the first stage is called `BUILDER` (through the `FROM node:12.19.0-buster AS BUILDER` command) and the second stage explicitly copies from the previous one by setting the `--from=BUILDER` flag in the `COPY` command.

With the help of the dotenv package[7], environment variables can be set via the `.env` file. The Node.js runtime will load these variables into the `process.env` object when the web app gets built. They are used to customize the footer and HTML meta tags. However, this leads to the problem that the built container image is no longer customizable. Environment variables no longer have any influence on the SPA and its static files when starting a container.

5.1.3 Helm Package

The final step of the implementation is to prepare the Scaffolder application, both the frontend and backend, so that administrators can easily install it in a Kubernetes cluster. For this purpose, a Helm Chart is to be developed. According to the official documentation, "a Chart is [comparable to a] package [and] contains all of the resource definitions necessary to run an application, tool, or service inside of a Kubernetes cluster" [85]. When the administrator installs the Chart in the cluster, then the running instance of the Chart is called a

[5] https://cli.vuejs.org/ (visited on 12/22/2020)

[6] https://github.com/nginxinc/docker-nginx/blob/master/stable/alpine/Dockerfile (visited on 12/22/2020)

[7] https://www.npmjs.com/package/dotenv (visited on 12/22/2020)

release. There can be several releases of a Chart inside of the same cluster. In this case, however, this is not desirable. Therefore, the Chart and the commands must be adapted so that there is always only one release. Thus the Chart's templates do not specify variables to be derived for the namespace or name of resources. Instead, the Readme comprises the command for installing the Chart with a concrete release namespace.

In total, the Chart consists of 13 different Kubernetes resources. That may seem like a lot for such a small application. However, this also includes the services, and configuration resources such as Secrets, Configmaps, Roles, and Rolebindings. As specified in the architecture chapter, there are three Deployments, each for the API server, the Redis database, and the SPA's web server. All resources are customizable via the `values.yaml` file, from which the variables in the resource templates derive their values. So administrators can customize these values to their individual needs and then install the Chart using the command from the Readme.

A few decisions were made in the course of creating the Chart. The default service account must be associated with a ClusterRole so that it can create namespaces. Also, this role must hold all rights to almost all API groups. The reason is that service accounts can only create new roles with certain rights if they have at least the same rights as the new role. Accordingly, one service account can not escalate its privileges. So the rights of the ClusterRole are very extensive.

The `values.yaml` file has been split into four categories, plus one category that would be dropped if the Scaffolder's container images were published. The latter is the `registryCredentials` object, which defines the credentials for the GitLab registry, which is where the two images for the Scaffolder application reside. Once the images get published to Docker Hub or another publicly available registry, this object is no longer needed. The other four categories consist of the `userInterface`, `application`, `userNamespace`, and `ingress` object.

Each the `userInterface` and `application` object specifies the respective container image, the image pull policy, and the `resources` object (known from a regular Kubernetes Deployment). Furthermore, the administrator can use the `userInterface.enableGzipCompression` value to toggle the Gzip compression of the Nginx web server. For this purpose, the `nginx-configmap.yaml` template was created, which contains an Nginx configuration file. Since Vue.js' artifacts are relatively large, it is recommended to turn on compression. Instead of 1378,46 KB, the web server will transfer only 303,16 KB. The object for the application server additionally comprises the `application.env` object. It specifies the information about the GitLab and Kubernetes API server, and the Redis connection.

The `userNamespace` object defines the rules that should apply to user namespaces created by the Scaffolder app. That includes default values for the image pull policy, the storage class for persistent volumes, and the TLS certificate issuer. Also, it contains the resource quotas and limits for the user namespaces.

The last category, defined by the `ingress` object, describes the Fully Qualified Domain Name (FQDN) and the TLS certificate issuer. Administrators can therefore set an individual domain under which the Scaffolder application should be accessible. Of course, the DNS entry must then also resolve to the IP address of the ingress controller.

Another challenge in developing the Chart involved the secrets. The cluster needs a `dockerconfig.json` to pull an image, and the Redis database needs a `redis.conf` file to secure the database with a password. For both purposes, the `_helpers.tpl` file implements one function each to create the content for the respective secret. These functions rely on Helm's template functions[8], for example, when converting values to their Base64 representation.

5.2 Challenges and Decisions

Most of the implementation decisions have already been addressed. However, some general decisions were still made that should be briefly discussed here.

5.2.1 User Namespace Constraints

A user namespace must ensure that its users cannot break out of the namespace and thus influence other users or even the entire system. This property is achieved via role permissions. Chapter 5.1.1 has already mentioned that the Scaffolder API server creates a role for this purpose, which it then associates with the namespace's default service account. The definition of the role resource has become quite extensive, as it is not possible to exclude individual resource types. For example, if a role is not to have access to pods, the definition must instead explicitly name all other resource types from that API group. This case, unfortunately, occurred with the most comprehensive API group, the core group, because it includes the `Namespace` resource. Users should only be given the right to execute methods on namespaces with the get, list, watch, and delete verb. As a result, all desired resource types of the core API group, including all sub-resource types like `services/proxy`, had to be listed explicitly. That ensures that users cannot change the namespace metadata, but can still delete their own (and only their own, since users represent a namespace-bound role) namespace. In addition to namespaces, users can perform all read methods on the `ServiceAccount`, `PersistentVolume`, `ResourceQuota`, `LimitRange`, `RoleBinding`, and `Role` resources.

[8] https://helm.sh/docs/chart_template_guide/function_list/ (visited on 12/23/2020)

There was also the question of whether users should be allowed to change the `Ingress` resource on their own, as this would risk clashing FQDN's from multiple users. However, users should be allowed to change ingress rules for their services independently. Therefore, no further adjustment is made to the role permissions, as the permissions system cannot selectively exclude the modification of the ingress FQDN.

In addition, a namespace must adhere to certain physical and quantity-related resource limits, or quotas. The `ResourceQuota` object enforces these namespace constraints. The physical quotas refer to the CPU, memory, and storage. That means that, for example, the CPU usage sum of all pods in a namespace cannot exceed the defined CPU quota. The quantity-based constraints, on the other hand, refer to the resource types themselves. In this case, the number of `PersistentVolumeClaims` and `Pods` should be limited. On the other hand, users can theoretically create all other resources in unlimited numbers. In particular, limiting the number of pods within a namespace is important because the cluster can only manage a finite number of pods and because there are no lower limits on the resources of a pod - but more on that later.

A Deployment, or any other resource that creates pods, includes the resource requirements in its container specification (namely the `resources` object). On the one hand, there is the `resources.requests` object, which represents the guaranteed physical resources and after which the *kube-scheduler* decides on which node it will create the pod, or whether it can create the pod at all. Furthermore, there is the `resources.limits` object, which specifies the degree to which resources are allowed to burst out until they are throttled (or terminated if too much memory has been allocated). That means that, by specifying higher limits, physical resources can be overcommitted. This is especially useful if you expect application utilization to fluctuate. Suppose a namespace is limited to 1000 millicores. A user deploys two pods which each request 500 millicores. Assuming that a pod runs at 75% utilization on average but has occasional peaks, for example, a 25% higher limit might make sense. That results in better overall utilization of the cluster. However, it is difficult to find the appropriate values for the requests and limits directly for the current environment. It gets even more difficult when you consider the update strategy of deployments. For example, if a deployment specifies `RollingUpdate` as the strategy, then at least one replica must be added during the update process. If the quotas are too low, this quickly leads to problems and pending statuses. The default values have to be evaluated and tuned with higher utilization and especially with more active usage. Therefore, the quota values in the Helm Chart are chosen subjectively for now.

Due to the fact that users must now always explicitly specify the resource requests and limits in their deployments, the Scaffolder API server should additionally apply a `LimitRange` object for each newly created namespace. It sets default values for the `resources.requests` and `resources.limits` objects within deployments. Also, it gives the ability to specify maximum and minimum values. Actually, the minimum values should be specified to prevent users from creating too many containers in their namespace. However, in the course of the work, it turned out that the TLS certificate issuer creates the pod that requests the certificate inside of the user namespace. This pod is scheduled with very low resource requests and would thus violate the `LimitRange`. Therefore, the author decided to solve the problem exclusively with the already mentioned quantity-based restriction of pods.

5.2.2 Frameworks and Databases for the Technology Options

The requirements only expect the technologies for the demonstration application to be "common". The author understands this to mean that the technologies to be used should be widespread and popular. The requirements already indicate possible options, namely the programming languages and the databases. However, the author made the final decision for the individual languages, frameworks, and databases at his own judgment. Various sources[9] [10] [11], including the number of stars on GitHub, were used for this purpose, as well as the subjective assessment of the popularity of individual technologies. The Scaffolder app finally implements the technologies shown in table 5.2.

User Interface	**Application**	**Data Storage**
None	Python (FastAPI)	In-Memory (Application-specific)
VanillaJS	Node.js (Express)	Redis
	Java (Javalin)	MySQL
	Java (Spring Boot)	MongoDB
	Go (Fiber)	

Table 5.2: Technology stack choices

The Java template is the only case where two different frameworks are implemented – a microframework (Javalin) and a full-stack framework (Spring Boot). At first, only the Spring version was to be implemented, as the author assumed that its popularity would be particularly high. However, difficulties turned out. A single template had to be implemented for each database connector. This has already been addressed in chapter 5.1.1. Furthermore, the Spring application is difficult for beginners to understand and especially to debug in a cluster environment, and generally, the resource usage is relatively high. Therefore, the Javalin template was added afterward.

[9] `https://insights.stackoverflow.com/survey/2020` (visited on 12/28/2020)
[10] `https://db-engines.com/en/ranking` (visited on 12/28/2020)
[11] `https://www.techempower.com/benchmarks/` (visited on 12/28/2020)

5.2.3 Container Image Building

The repositories that the Scaffolder app creates contain a `.gitlab-ci.yml` file in their root directory. It defines the four stages for the continuous integration and deployment pipeline. Overall, it includes the build, deploy, cleanup, and shutdown stages. The GitLab runners automatically execute the first two stages as developers push new code into the repository. The runners themselves run with Docker as containers.

The usual method of building images inside a Docker container is using Docker-in-Docker (dind), which means launching the Docker binary as a container image. However, this has two drawbacks. First, it poses security risks because the Docker container must launch with privileged rights. So it would have access to the host system. And secondly, it degrades performance, not least because the Docker daemon has to be started first. A good alternative is Kaniko[12]. It is a tool to build container images within container environments without depending on the Docker daemon. Accordingly, the Kaniko image can also be executed entirely in userspace. For the reasons mentioned above, the pipeline thus uses Kaniko instead of dind. It also uses Kaniko's layer caching feature, which again leads to a considerable time advantage during the build.

The deploy stage then applies all Kubernetes manifests in the cluster using the `bitnami/kubectl` container image. Furthermore, the user can manually run the cleanup stage to delete all these applied manifests again. The shutdown stage can also be executed manually and deletes the namespace from the cluster. This step is not reversible and is used to free up capacity from the user side when the project is finished.

5.2.4 Reverse Proxy

The Scaffolder app exposes two services. One for the API server and one for the Nginx web server. Both services should be accessible under the same physical address. In this case, it is the address of the Ingress controller, which acts as a reverse proxy. The Ingress resource offers name-based virtual hosting, which means routing HTTP requests to multiple hostnames on the same physical server. However, the Scaffolder website accessed by the browser would then have to send requests to multiple domains, resulting in what is known as cross-origin resource sharing (CORS)[13]. This would require additional headers and HTTP methods to be implemented. Therefore, the author decided to use a simpler alternative based on the HTTP path. As soon as the Ingress controller receives a request that starts with the `/api` path, it forwards it to the API server. All other requests under the `/` path get forwarded to the Nginx server. To avoid security risks from the SPA's script, the API server sets the

[12] `https://github.com/GoogleContainerTools/kaniko` (visited on 12/29/2020)

[13] `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS` (visited on 12/29/2020)

session cookie with the `HttpOnly` flag.

However, path-based routing causes difficulties during development. If the developer starts both servers locally at different ports, then the browser perceives them as different origins. That again leads to CORS. However, the SPA should not know under which domain it is running, not least because administrators cannot set a “host” environment variable for the Nginx container. Therefore, for developers, the `/sources/frontend/local-dev-reverse-proxy` directory contains a Node.js module, which can be started via the command `npx dprox`. It starts a local reverse proxy that can be configured via the `proxy.config.js` file. Developers can now run both servers locally, guaranteeing the same behavior as when running them in a cluster.

5.2.5 StatefulSet

The Scaffolder project initially also required StatefulSets to be covered as a Kubernetes resource. This type of resource is particularly useful for scaling databases. In order to include a simple example of StatefulSets, it was necessary to find a database that could scale easily. This was the first challenge, as databases are generally difficult to scale. After some tests, the decision was made to use Redis (see `/preparation/redis-demo/deployment-files` directory). However, implementing the database connectors for a replicated database and different languages proved to be difficult. In addition, the manifest files are relatively extensive, as they contain scripts and ConfigMaps that configure the replicated database. Beginners, in particular, would feel overwhelmed by the large number of files, configuration artifacts, and new concepts of headless services. In addition to the newly introduced Kubernetes concepts, users also need to learn the database-specific concepts regarding horizontal scaling – in this case, Redis replication. However, the purpose of this work is to flatten the learning curve and not unnecessarily introduce many new concepts. Therefore, the decision was finally made to remove the StatefulSets from the requirements.

5.2.6 Beta Phase

One goal was to develop a functioning app as quickly as possible so that we could start the beta phase quickly and gather feedback. That was well worth it, as valuable feedback could be incorporated during the implementation period. Therefore, many thanks to Professor Kratzke for making the beta phase possible and for contributing helpful suggestions.

The user interface was revised and simplified in the course of the feedback. Users can now no longer set the GitLab project visibility themselves. It is always set to private by default. The tooltips that contain hints about the usage and usefulness of individual layers also received links to the documentation of the Kubernetes resources used. In addition, the configuration page includes another category "Advanced Option" in a dropdown area. Users who have advanced requirements can add persistence and asynchronous or scheduled jobs manually in this area. The initial smaller list of options makes beginners less overwhelmed, the project size becomes smaller, and fewer resources will be needed.

The generated `README.md` has received an additional chapter "Connecting to the cluster" to also address options such as the Kubernetes IDE "Lens". Feedback also came in regarding the project structure. For larger projects, developers quickly lose track of the Kubernetes manifests if they are located in each layer's subdirectory and have the same name. Therefore, the Scaffolder app now creates an additional directory for the manifest files with a distinct naming scheme. The "Project Structure" chapter mentions the used Kubernetes concepts and adds links to their documentation.

Some consistency issues in the `.gitlab-ci.yml` file and Dockerfiles have also been detected during the beta. Therefore, the `.gitlab-ci.yml` file now includes a `variable` object that lists all versions for both the `kubectl` image and the versions for the container images to be built. Also, all `Dockerfile` templates were checked for their image tag. The Scaffolder app now only generates Dockerfiles that use a small image version with alpine or slim in their tag.

5.3 Limitations

Unfortunately, some architectural decisions turned out to be limitations in the course of the work. The following sections thus explain the effects of individual decisions.

5.3.1 Cohesive Demo Application

The interconnected layers of the demonstration app limit the user's requirements. For example, the Scaffolder app always requires the application layer to be selected. That means that there is always an application server with which the other layers interact. So users cannot configure a project that only requires a simple HTML website. Or they can't configure a project that requires only the periodic execution of a container. Also, the Scaffolder app only allows for one database. What about the case when a user wants to use a SQL-based database and, additionally, needs a Redis instance for caching?

There should have been a more intense discussion about the cohesive example with the CRUD app from the beginning. Its pros and cons should have been evaluated because this specific use case makes the generated code more complicated than it should be. That could potentially lead to confusion and increased onboarding time for users. Moreover, the question arises whether it makes sense to generate a full example, considering that users would adapt the source code to their own needs and delete most of it again afterward. It also complicated the development by requiring unnecessary bugs to be fixed. For example, there were problems with database connection libraries, of which not all were implemented in a thread-safe manner. This problem had to be solved to make the Add-Many-Names and Cleanup-Names jobs work properly since they send many requests in a short time.

5.3.2 Maintainability of the Templates

The cohesive use case also results in a larger maintenance effort for the Scaffolder app. The codebase is larger, and, most importantly, there are significantly more dependencies in the generated apps. Updating the source code and dependencies of templates would only be possible with automated testing, as there are too many different combinations for developers to test. Manual modification and testing would look like this: Render the template to create the project artifacts; Modify the source code or/and dependencies; Test for the correct functioning of the CRUD app; Merge the changes into the template. However, there are no automated tests for this process, only the Postman Collection (see `/sources/Backend/api/postman_collections` directory) for manual testing of the CRUD App endpoints.

5.3.3 Dependency Locking

Another problem arose with the dependencies for Node.js applications. Typically, a file records the versions of packages so that they do not differ between different developers and the production system. Therefore, this so-called lock file should also always be kept in the Git repository. However, the problem is that the templates for these lock files include all dependencies. The Scaffolder app then filters out the unneeded dependencies as soon as the source code of the demonstration app gets generated. However, in the case of Node.js, this is not possible or only with a disproportionate effort. The Node package manager (npm) requires two files: `package.json`, which specifies the project and dependencies, and `package-lock.json`, which sets the versions of the dependencies and their dependencies. If only the `package.json` file is present, the package manager always installs the latest minor version (according to the SemVer[14] scheme). Only by means of the `package-lock.json` file, the versions can be determined reliably. However, in this case, the file consists of almost 1700 lines and numerous dependencies. The creation of a template for the lock file is thus connected with high effort. The author does not see the effort as justified and decided to leave the template unedited. That, in turn, means that the demonstration application for Node.js also installs dependencies that are not needed.

[14] https://semver.org/ (visited on 12/30/2020)

5.3.4 Binary Files

Another limitation is that the templates must be formatted in UTF-8 so that the template engine can render them. As a result, for example, the Maven wrapper for the Spring template and favicons for the user interface cannot be rendered. The disadvantage is not particularly bad, but it should still be mentioned here.

5.3.5 Single-Page Application

Chapter 5.1.2 on containerizing the web app has already touched on the fact that after building the Nginx container image, the source code, and thus the SPA is static. Within the Helm Chart, an administrator can no longer do any customization. This leads to the problem that information like the organization name, the imprint, links, and the terms of use cannot be changed by future users of the Chart. If users still want to do this, they have to pull the source code and rebuild the container image.

In addition, the SPA now introduces a second place where the state must be managed. The strict separation of frontend (SPA) and backend (API server) resulted in a larger interface than actually necessary. For example, the SPA sends three separate requests to the API server when it gets opened in the browser. Of course, the SPA then has to manage the state of these requests. In hindsight, the author feels that it might have made more sense to work with dynamic templates rendered and delivered by a monolithic web server. That would reduce the HTTP interface and state management overhead.

5.3.6 Certificate Issuing With Let's Encrypt

The number of new projects essentially is limited to a maximum of 50 per week when using Let's Encrypt as the certificate authority. That is because Let's Encrypt issues new certificates for only up to 50 domains per week for a single account. The problem here is that the responsible cert-manager runs centrally in the cluster under a single email address. To avoid the limit, users could also instantiate a cert-manager in their own namespace, though that would decrease their available resource contingent. And it would not only decrease it for the time of the issuing procedure but for as long as the user wants the domain to be secured because the cert-manager also manages the renewal of certificates.

6 Testing the Requirements Fulfillment

Now that the implementation is complete, this chapter evaluates whether the bachelor thesis meets the requirements. The analysis uses the tables created in Chapter 3 for the functional and non-functional requirements. Each entry includes a priority, description, and fit criterion. The author then evaluates the fulfillment using the criterion. In order to stay within scope, this chapter does not cover the evaluation process for each requirement. With 42 requirements, that would be significantly too much. Therefore, the following paragraphs provide an overview of the requirements that were met and those that were not, or only partially met.

The good news is that the implemented solution fulfills all requirements with the priority "must-have". These alone were 22 requirements. In addition, the two "won't-have" requirements, F10 and F11, are not considered further as they were excluded from the outset. Of the functional requirements, only one was not fulfilled. It is requirement F8 with the priority "could-have", which requires the solution to delete inactive namespaces automatically. Fulfilling this requirement has proven to be time-consuming during the course of the work, as there is no interface that returns the last activity of namespaces in a simple way. A possible solution would need to check each namespace individually. This process requires the check of each resource's last update time and if it is within the desired "active period". Checking the activity of pods is not sufficient since usually higher-level resource types manage these. There is also the added difficulty that users can install CronJobs and thus keep the namespace active forever. That would then require a separate rule. Overall, it can be said that finding rules that describe the inactivity of a namespace is an elaborate process.

Of the non-functional requirements, a total of two were not met and one was only partially met. However, these are not explicitly listed in the task definition of the bachelor thesis and therefore only have the priority "could-have". Requirement NF10 requires that the solution shows the individual creation steps to the user for transparency reasons. The idea was that this would enable users to understand what is actually happening in the background and thus increase the learning effect. However, the fulfillment of this requirement would have needed either WebSockets or server-sent events. In addition, the architecture would have to be extended to include an event queue in which each service module publishes the completed tasks. The user would then subscribe to that queue through the SPA. However, when it turned out that the creation of a new namespace and the GitLab project happens almost instantaneously, the author decided that it makes no sense to print so many events to the user without explanation.

The next unmet requirement is NF11. It says that the solution should be well tested using automated tests. The required criterion is full test coverage of unit tests for the controller endpoints. Testing of all handlers would also cover the service modules since they are called by the handlers. The service modules, in turn, rely on communication with external interfaces (GitLab and Kubernetes). In order for the service modules to be meaningfully tested without external influences or problems, the unit tests must mock the external interfaces. This problem is usually solved via dependency injection. As soon as modules are initialized, they are provided with synthetic objects that mock external interfaces. However, the author did not gain any experience with this principle before the bachelor thesis and did not consider it in the initial software architecture. Instead, the author took other measures to test the Scaffolder solution. For testing the controller endpoints, a Postman collection resides in the `/sources/Backend/api/postman_collections` directory. It consists of queries with sample parameters, or bodies, for each endpoint of the Scaffolder API. After implementing major features, the author performed system tests. That means that the author fully integrated the Scaffolder solution into a production-like system environment and tested that the requirements were met. That includes the entire process from logging in to forwarding the user to the GitLab project and testing the live demonstration app. During system testing, the author manipulated the external systems to cover as many cases as possible. For example, already occupied project or namespace names were entered, or the cluster capacity was artificially reduced. The system tests were then followed by the beta test with external persons. As chapter 5.2.6 describes, the beta test was mainly for validating and adapting non-functional requirements in order to increase acceptance. Lastly, the creation of the templates presented a challenge in this work. Therefore, the `scaffolder` module saves the rendered source code locally when the Scaffolder app does not run inside the Kubernetes cluster. Developers can thus more easily validate and test changes to the template without actually having to create a new GitLab project.

The last requirement that is only partially fulfilled is NF12. It requires the individualization of the user interface. Chapter 5.3.5 about the limitations has already explained the implementation of this requirement and its limitation.

7 Conclusion

The core task consisted of answering two questions. How can an organization offer a container environment to its members in a simple way and with as little administrative effort as possible? And the second question: How can the entry hurdle and the learning curve for containerization and container orchestration be flattened? This bachelor thesis answered both questions with the help of a software solution.

7.1 Outcome

The developed web app implements all initial functional requirements from the task definition. It implements a self-service portal where users can request new Kubernetes virtual clusters. Without the intervention of administrators, the organization can provide computing power to its members. Users can directly authorize themselves using their already existing GitLab account. The so-called Scaffolder solution creates the new virtual clusters in the form of new restricted namespaces. The restrictions provide the organization with the ability to limit the computing power provided. They also serve for restricting the rights of users within the server cluster. This self-service portal gives users an environment in which they can run containers and also make their services publicly accessible.

The portal answers the first question but does not flatten the learning curve of containerization and container orchestration. On the contrary – simply providing a Kubernetes cluster would pose a new challenge to users. Therefore, during the creation process, the user has the option to select a technology stack to be used in the project. At its core, the stack consists of three layers. The user interface layer defines the technology with which the web app or website will be developed. The application layer represents a web server with HTTP API. And the data storage layer defines the database that will persist the application's state. Using this project specification, the Scaffolder App can additionally create a new GitLab project on behalf of the user. This project contains sample source code for an application that demonstrates how each Kubernetes principle works. The concrete example allows users to understand the Kubernetes manifest files. The project's Readme file guides users. Moreover, the project includes a configuration for the CI/CD pipeline. For each code update, the pipeline rebuilds the application containers and pushes them to the image registry. After the images are published, the pipeline also executes the appropriate commands to apply the Kubernetes manifests from the repository in the cluster. These principles introduce users to Kubernetes and promote a DevOps culture.

However, difficulties also arose during the course of the work. For example, the cohesive demonstration app that the Scaffolder app creates may limit the possible use cases for new projects. In addition, the demonstration app itself has a certain size. Users need to familiarize themselves with its code and later discard it in favor of their own application. Also, the concrete demonstration example makes it more difficult to maintain the templates. Another limitation arose from the choice of a single-page application. Administrators cannot customize the frontend to their own organization's needs through environment variables while installing the Scaffolder app. That is because the single-page application consists only of static code that must be built beforehand.

7.2 Outlook

So far, the solution has only been tested with a few people. Moreover, the beta phase only made use of a single-node cluster. In the future, more people with different use cases should test the solution and, especially, evaluate it. Only with more feedback, the Scaffolder app can be further developed efficiently. What developers should question for future improvement are the choices of the single-page application and the cohesive demonstration application.

In addition to the already implemented functions, the solution offers numerous extension possibilities. The following list gives a brief overview of thoughts and ideas for future features:

- *Namespace Limits Per User*: Actually, the bachelor thesis excluded the requirement for namespace limits per user because the solution should be implemented stateless. The statelessness was achieved. But the Scaffolder app adds the user's email address in the metadata of each newly created namespace. Users cannot modify this metadata. Thus, the Scaffolder App can use the metadata to check how many namespaces a user already owns and limit the number accordingly.

- *Admin Notifications on Events*: It may happen that the cluster capacities are exhausted, or other incidents occur. In that case, the solution could automatically notify the administrator via email.

- *Automatic Container Image Tag Updates*: In its current state, Kubernetes does not detect when an image changes. The CI pipeline does build the container images with each new code push, but these are not necessarily deployed automatically. The generated Readmes explain to users how to get Kubernetes clusters to download the new image. It would be better to have a solution that customizes the Kubernetes manifests within the pipeline so that deployments get updated. Possible solutions could involve kustomize[1] or a make file.

- *Automatic Deletion of Inactive Namespaces*: As described in Chapter 6, finding rules for when a namespace is considered inactive is an elaborate process. Also, the implementation is anything than trivial since each namespace and resource must be checked individually. What might be more practical in the university's case is to simply reset the cluster after the end of each semester.

- *Supporting Multiple Git Hosting Providers*: So far, the Scaffolder app only supports GitLab for creating a new project. But you could also use all kinds of other Git hosting providers. For example, you could add GitHub and use its so-called actions for the CI/CD pipeline. The templates would then have to include another GitHub-dependent workflow file in addition to the `gitlab-ci.yml` file.

[1] `https://kustomize.io/` (visited on 01/04/2021)

Acknowledgements

Throughout the work on this bachelor thesis I received great support.

First of all, I would like to thank my supervisor, Mr. Professor Nane Kratzke, who was always able to help me in the shortest possible time. In particular, I appreciate his very constructive and insightful feedback, which greatly helped my research.

I would also like to thank Mr. Professor Andreas Schäfer for his support as the second examiner and his advice on organizational decisions.

Furthermore, I would like to thank my father, who highly supported me in any way throughout my studies and showed understanding when I preferred to program instead of doing the laundry.

Appendix

Hello World Service Example with Kubernetes

Listing 7.1: Kubernetes manifest file for the service, stateful set and ingress object of the Hello World Service Example. (hello-world.yaml)

```yaml
apiVersion: v1
kind: Service
metadata:
  name: hello-world
spec:
  ports:
    - port: 80
      targetPort: 80
  selector:
    app: hello-world-pod
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: hello-world-stateful-set
spec:
  selector: # for the underlying replica set controller
    matchLabels:
      app: hello-world-pod
  serviceName: hello-world # must be created before the stateful set
  template: # pod template
    metadata:
      labels:
        app: hello-world-pod
    spec:
      containers:
        - name: hello-world-container
          image: "registry-address/hello-world:0.0.1"
          env:
            - name: GREET_NAME
              value: "Gandalf the Grey"
          ports:
            - containerPort: 80
          volumeMounts:
            - name: data-volume
              mountPath: /hello-world-service/data
  volumeClaimTemplates:
    - metadata:
        name: data-volume
        labels:
          app: hello-world
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: "100Mi"
---
kind: Ingress
apiVersion: networking.k8s.io/v1
metadata:
  name: hello-world
spec:
  rules:
  - host: hello-world.io
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: hello-world
            port:
              number: 80 # port of the service
  tls:
```

```
  - hosts:
    - hello-world.io
    secretName: hello-world-io-certificate # must be issued beforehand
```

Requirements

The requirements appendix contains the functional and non-functional requirements. These were elaborated in the *Requirements Analysis* chapter and are listed here with a unique identifier (F: Functional Requirement; NF: Non-Functional Requirement) in tabular form. They also indicate the priority (M: Must-Have; S: Should-Have; C: Could-Have; W: Won't-Have) in the "P" column. The *Testing the Requirements Fulfillment* chapter uses the "Fit Criterion" column's description to evaluate the fulfillment of both the functional and non-functional requirements. Depending on whether the requirement was fulfilled, the "F" column is filled with a y (yes), n (no), or p (partially).

Functional Requirements

ID	P	Description	Fit Criterion	F
F1	M	Users shall be able to authenticate themselves via GitLab's OAuth 2 interface	The solution uses the university's GitLab server for authentication via OAuth 2	y
F2	M	The solution shall create GitLab projects on behalf of an authenticated user	After the successful authentication, the user can create a new GitLab project with a chosen name through the solution	y
F2.1	M	The created GitLab project shall contain access credentials for the K8s cluster	The created GitLab project contains an environment variable with access credentials for the K8s cluster	y
F2.2	M	The created GitLab project shall contain a CI/CD pipeline example	The created GitLab project contains a gitlab-ci.yml file with automated container image builds and deployments	y
F2.3	S	The created GitLab project shall prevent secrets leakage	The created GitLab project makes use of the .dockerignore and .gitignore files to exclude secretive files from being exposed	y

Table 9 continued from previous page

ID	P	Description	Fit Criterion	F
F2.4	C	Users shall be able to choose a project description and the project visibility	The solution offers input fields for the description and visibility, and applies them to the GitLab project	y[1]
F2.5	M	Users shall be able to select a desired tech stack for the project to be created	The solution offers selectable technology options for the initial template project based on a user interface, application, and data storage layer	y
F2.6	M	The options for the desired tech stack shall cover common languages	The technology options cover Java, Node.js, and Python as selectable languages	y
F2.7	M	The options for the desired tech stack shall cover common databases	The technology options cover MySQL, MongoDB, and Redis as selectable databases	y
F2.8	M	The options for the desired tech stack shall cover typical K8s workloads	The technology options cover K8s Deployment, StatefulSet, Job, and CronJob workloads	y[2]
F2.9	M	The options for the desired tech stack shall cover typical K8s exposing methods	The technology options cover K8s Service and Ingress resources, and make the application reachable from outside the cluster	y
F2.10	M	The solution shall initialize the created project with source code according to the desired tech stack	The solution initializes the project with a template application consisting of all selected technologies	y
F2.11	S	The solution shall guide users through the created project template	The created GitLab project contains a Readme file (Getting Started) that describes the structure and setup process	y
F3	M	The solution shall create namespaces for an authenticated user based on the project name	The solution creates a K8s namespace for the new corresponding GitLab project	y

[1] The project visibility option got canceled, see section 5.2.6.
[2] The StatefulSet option got canceled, see section 5.2.5.

Table 9 continued from previous page

ID	P	Description	Fit Criterion	F
F3.1	M	Namespace users shall be restricted in the usage of cluster resources	The solution applies resource quotas (CPU, memory, and storage) to the namespace	y
F3.2	M	Namespace users shall be constrained in the creation of single resources	The solution applies limit ranges and default values for the resources object of deployments	y
F3.3	M	Namespace users shall be able to read and write cluster resources only in their own namespace	The solution applies authorization rules so that namespace users can read and write cluster resources only in their own namespace	y
F3.4	M	The restrictions and constraints for namespaces shall be configurable	The Helm 3 chart allows the customization of values for the restrictions and constraints in namespaces	y
F3.5	M	The solution shall create access credentials for a service account related to the created namespace	The solution creates a service account in the namespace and retrieves its access credentials	y
F4	M	The solution shall expose the created template project's service at a predictable address	The solution uses the namespace's name (which equals the project name's slug) as a subdomain of the cluster domain to expose the service	y
F4.1	S	When exposing a service, the solution shall use TLS termination	The solution creates Ingress objects with TLS certificates and relies on cert-manager for certificate issuing	y
F5	M	The solution shall allow the project creation only if the name is available	The solution checks the project name and namespace name availability when creating a new project	y
F6	S	The solution shall associate a created namespace with a user/person for administration purposes	The solution creates a namespace with labels associated with the user (e.g. email, GitLab ID)	y
F7	C	The solution shall point out the terms and conditions and require the user to accept them	The solution requires the user to accept the terms and conditions for the project creation	y
F8	C	The solution shall automatically delete inactive namespaces	The solution deletes namespaces after a certain time period of inactivity	n

Table 9 continued from previous page

ID	P	Description	Fit Criterion	F
F9	C	The solution shall offer users easy resetting and deletion of a namespace	The created GitLab project's CI pipeline contains two manual stages for resetting and deleting of the namespace	y
F10	W	The solution shall support internationalization	The solution's user interface and readme supports multiple languages	n
F11	W	The solution shall restrict communication between namespaces	The solution applies network policies for the namespace's resources	n

Table 9: Functional Requirements

Non-Functional Requirements

ID	P	Description	Fit Criterion	F
NF1	M	The solution shall be distributed as a Helm 3 chart	A Helm 3 chart is available for the solution	y
NF2	M	The solution shall be installable in any K8s Cluster via a Helm 3 chart	The Helm 3 chart is customizable to fit individual (cluster) requirements	y[3]
NF3	M	The solution's backend shall be written in Python	The solution's backend is written in Python	y
NF4	M	The solution shall be stateless to be simple, horizontally scalable, and resilient	The solution does not rely on persistence like storage and does not maintain a state over the namespaces, users, and user projects	y[4]
NF5	S	The solution shall work correctly in case of a connection loss during the creation process	The solution continues with the creation process during a connection loss and writes the cluster information in the repository's Readme[5]	y

[3] Customization possibilities described in section 5.1.3

[4] It does maintain a state over the user sessions. But since this is inevitable, the requirement is still considered to be fulfilled.

[5] A more reliable and less confusing solution could be to automatically send the log and cluster information via e-mail.

Table 10 continued from previous page

ID	P	Description	Fit Criterion	F
NF6	S	The solution shall not be granted administrator rights for the GitLab server	The solution only relies on the rights of authorized GitLab users	y
NF7	S	The solution's authorization shall be secure	The solution uses the server-side OAuth 2.0 authorization code flow without storing plain credentials on client devices	y
NF8	S	A created project shall work "out of the box"	The solution creates a project containing a demonstration application that works immediately after the creation	y
NF9	S	The solution shall not rely on K8s' Custom Resource Definitions (CRDs)	The solution does not create own CRDs	y
NF10	C	The creation process of the project and namespace shall be transparent to the user	The solution's user interface prints each step of the creation process	n
NF11	C	The solution shall be well tested with automated tests	The solution's service endpoints have full test coverage on unit tests	n
NF12	C	The solution shall be white-labeled	The solution offers the customization of logos, labels and terms and conditions	p[6]
NF13	C	The solution's user interface shall be usable on different screen sizes	The solution's user interface is responsive to both mobile and desktop devices	y
NF14	C	The solution shall create new projects only if the cluster has enough capacity available	The solution allows the project creation only if the deployment of the demo application is possible	y

Table 10: Non-Functional Requirements

[6] The SPA's environment variables can not be changed after building the source code. Thus customization is possible only if rebuilding the SPA.

Event Storming for the System Architecture

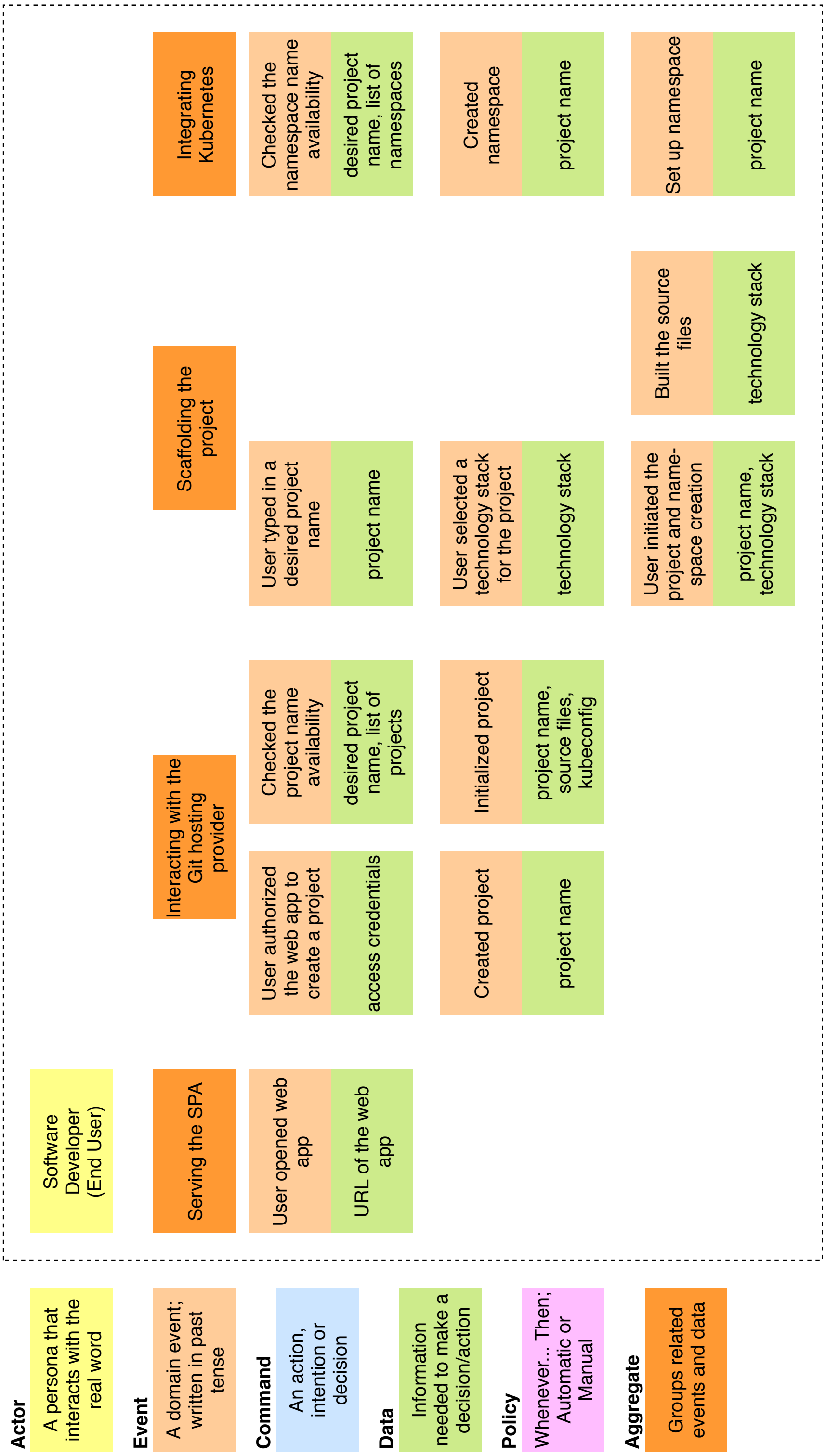

Software Developer (End User)
Serving the SPA
User opened web app
URL of the web app
Interacting with the Git hosting provider
User authorized the web app to create a project
access credentials
Created project
project name
Checked the project name availability
desired project name, list of projects
Initialized project
project name, source files, kubeconfig
Scaffolding the project
User typed in a desired project name
project name
User selected a technology stack for the project
technology stack
User initiated the project and namespace creation
project name, technology stack
Built the source files
technology stack
Integrating Kubernetes
Checked the namespace name availability
desired project name, list of namespaces
Created namespace
project name
Set up namespace
project name
Actor
A persona that interacts with the real word
Event
A domain event; written in past tense
Command
An action, intention or decision
Data
Information needed to make a decision/action
Policy
Whenever... Then; Automatic or Manual
Aggregate
Groups related events and data

Figure 9: Event storming to identify all activities and domains of a possible solution [76]. (Commands and policies omitted for space reasons.)

List of Figures

List of Tables

Listings

Bibliography

[1] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study," *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017.

[2] myLab, *myLab Homepage*, Technische Hochschule Lübeck. [Online]. Available: https://mylab.th-luebeck.de/ (visited on 10/03/2020).

[3] GitLab, *Requirements for Auto DevOps for Kubernetes*, GitLab Inc. [Online]. Available: https://docs.gitlab.com/ee/topics/autodevops/requirements.html (visited on 10/02/2020).

[4] R. Stanton, "Do technical/professional writing (tpw) programs offer what students need for their start in the workplace? a comparison of requirements in program curricula and job ads in industry," *Technical Communication*, vol. 64, no. 3, pp. 223–236, 2017.

[5] G. H. L. Pinto, F. Figueira Filho, I. Steinmacher, and M. A. Gerosa, "Training software engineers using open-source software: The professors' perspective," in *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)*, IEEE, 2017, pp. 117–121.

[6] L. Körbes. (Sep. 4, 2020). "Toolchains Behind Successful Kubernetes Development Workflows - L Körbes, Tilt," CNCF, [Online]. Available: https://www.youtube.com/watch?v=4YanEWCAPlk (visited on 10/03/2020).

[7] *Namespaces*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/ (visited on 10/03/2020).

[8] I. Dvoretskyi and C. Aniszczyk. (Dec. 10, 2018). "Cloud Native Computing Foundation ("CNCF") Charter," Cloud Native Computing Foundation, [Online]. Available: https://github.com/cncf/foundation/blob/master/charter.md (visited on 10/03/2020).

[9] OCI, *About the Open Container Initiative*, Open Container Initiative. [Online]. Available: https://opencontainers.org/about/overview/ (visited on 10/03/2020).

[10] Stack Overflow. (Feb. 28, 2020). “Stack Overflow Developer Survey 2020,” Stack Exchange Inc., [Online]. Available: https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-platforms-wanted5 (visited on 10/03/2020).

[11] VMware, *Cloud native applications: Ship faster, reduce risk, and grow your business*, VMware, Inc. [Online]. Available: https://tanzu.vmware.com/cloud-native (visited on 10/04/2020).

[12] H. El-Rewini and M. Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing*, ser. Wiley Series on Parallel and Distributed Computing. Wiley, 2005, ISBN: 9780471478393. [Online]. Available: https://books.google.de/books?id=7JB-u6D5Q7kC.

[13] A. Bondi, “Characteristics of scalability and their impact on performance,” Jan. 2000, pp. 195–203. DOI: 10.1145/350391.350432.

[14] N. Herbst, S. Kounev, and R. Reussner, “Elasticity in cloud computing: What it is, and what it is not,” *International Conference on Autonomic Computing*, pp. 23–27, Jan. 2013.

[15] P. Mitropoulou, E. Filiopoulou, C. Michalakelis, and M. Nikolaidou, “Pricing cloud iaas services based on a hedonic price index,” *Computing*, vol. 98, no. 11, pp. 1075–1089, 2016.

[16] Microsoft, *Cloud-native resiliency*, Microsoft. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/resiliency (visited on 10/04/2020).

[17] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: Yesterday, today, and tomorrow,” *CoRR*, vol. abs/1606.04036, 2016. arXiv: 1606.04036. [Online]. Available: http://arxiv.org/abs/1606.04036.

[18] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[19] JetBrains, *The State of Developer Ecosystem 2020*, JetBrains s.r.o., 2020. [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2020/ (visited on 10/05/2020).

[20] IBM Cloud Team, *Cloud-native resiliency*, IBM, Sep. 2, 2020. [Online]. Available: https://www.ibm.com/cloud/blog/soa-vs-microservices (visited on 10/05/2020).

[21] M. Fowler and J. Lewis, *Microservices: a definition of this new architectural term*, Mar. 25, 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html (visited on 10/05/2020).

[22] D. Gannon, R. Barga, and N. Sundaresan, “Cloud-native applications,” *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, 2017.

[23] A. Wiggins, *The Twelve-Factor App: VI. Processes: Execute the apps as one or more stateless processes*, 2017. [Online]. Available: https://12factor.net/processes (visited on 10/06/2020).

[24] C. Pautasso and E. Wilde, “Why is the web loosely coupled? a multi-faceted metric for service design,” in *Proceedings of the 18th international conference on World wide web*, 2009, pp. 911–920.

[25] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina, “Microservices: How To Make Your Application Scale,” *CoRR*, vol. abs/1702.07149, 2017. arXiv: 1702.07149. [Online]. Available: http://arxiv.org/abs/1702.07149.

[26] Web Services Architecture Working Group, *Web Services Architecture*, W3C, Feb. 11, 2004. [Online]. Available: https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest (visited on 10/05/2020).

[27] R. Bauer, *What is the Diff: VMs vs Containers*, Jun. 28, 2018. [Online]. Available: https://www.backblaze.com/blog/vm-vs-containers/ (visited on 10/05/2020).

[28] Docker Community, *Docker Overview*, Docker Inc. [Online]. Available: https://docs.docker.com/get-started/overview/ (visited on 10/07/2020).

[29] OCI, *OCI Runtime Specification*, Open Container Initiative. [Online]. Available: https://github.com/opencontainers/runtime-spec (visited on 10/05/2020).

[30] ——, *OCI Image Specification*, Open Container Initiative. [Online]. Available: https://github.com/opencontainers/image-spec (visited on 10/05/2020).

[31] Containerd, *containerd - An industry-standard container runtime with an emphasis on simplicity, robustness and portability.* [Online]. Available: https://containerd.io/ (visited on 10/05/2020).

[32] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud computing patterns: fundamentals to design, build, and manage cloud applications.* Springer, 2014.

[33] A. Khan, “Key characteristics of a container orchestration platform to enable a modern application,” *IEEE Cloud Computing*, vol. 4, pp. 42–48, Sep. 2017. DOI: 10.1109/MCC.2017.4250933.

[34] *Connecting Applications with Services: The Kubernetes model for connecting containers*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/connect-applications-service/#the-kubernetes-model-for-connecting-containers (visited on 10/06/2020).

[35] Etienne Tremel, *Kubernetes deployment strategies*, Container Solutions, Sep. 15, 2017. [Online]. Available: https://blog.container-solutions.com/kubernetes-deployment-strategies (visited on 10/06/2020).

[36] Alexis Richardson, *GitOps - Operations by Pull Request*, Weaveworks Inc., Aug. 7, 2017. [Online]. Available: https://www.weave.works/blog/gitops-operations-by-pull-request (visited on 10/06/2020).

[37] OpenJS Foundation, *Node.js*, OpenJS Foundation. [Online]. Available: https://nodejs.org/en/ (visited on 10/08/2020).

[38] ——, *Express - Node.js web application framework*, OpenJS Foundation. [Online]. Available: https://expressjs.com/ (visited on 10/08/2020).

[39] Docker Community, *About storage drivers*, Docker Inc. [Online]. Available: https://docs.docker.com/storage/storagedriver/ (visited on 10/09/2020).

[40] ——, *Best practices for writing Dockerfiles*, Docker Inc. [Online]. Available: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (visited on 10/09/2020).

[41] *Kubernetes Components*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/ (visited on 10/12/2020).

[42] *Managing Resources*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/ (visited on 10/13/2020).

[43] *The Official YAML Web Site*. [Online]. Available: https://yaml.org/ (visited on 10/13/2020).

[44] *Understanding Kubernetes Objects*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/ (visited on 10/12/2020).

[45] *Authorization Overview*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/authorization/ (visited on 10/14/2020).

[46] *Resource Quotas*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/concepts/policy/resource-quotas/ (visited on 10/14/2020).

[47] *Building large clusters*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/setup/best-practices/cluster-large/ (visited on 10/14/2020).

[48] *Limit Ranges*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/concepts/policy/limit-range/ (visited on 10/14/2020).

[49] *Pods*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/ (visited on 10/13/2020).

[50] *Init Containers*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/init-containers/ (visited on 10/13/2020).

[51] *ReplicaSet*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ (visited on 10/13/2020).

[52] *Rancher Docs: Kubernetes Workloads and Pods*, Rancher Labs. [Online]. Available: https://rancher.com/docs/rancher/v2.x/en/k8s-in-rancher/workloads/ (visited on 10/13/2020).

[53] *StatefulSet*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/ (visited on 10/14/2020).

[54] *Service*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 10/14/2020).

[55] *Ingress*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/ingress/ (visited on 10/14/2020).

[56] S. Pittet, *Continuous integration vs. continuous delivery vs. continuous deployment*, Atlassian, 2020. [Online]. Available: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment (visited on 10/16/2020).

[57] *Shells supported by GitLab Runner*, GitLab Inc., 2020. [Online]. Available: https://docs.gitlab.com/runner/shells/ (visited on 10/16/2020).

[58] *Introduction to CI/CD with GitLab*, GitLab Inc., 2020. [Online]. Available: https://docs.gitlab.com/ee/ci/introduction/ (visited on 10/16/2020).

[59] *DNS for Services and Pods*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/ (visited on 10/19/2020).

[60] B. Adams, *Another very late response: you're confusing Googlebot (the crawler) with Caffeine (Google's indexing system)*, Jul. 18, 2017. [Online]. Available: https://medium.com/@badams/another-very-late-response-youre-confusing-googlebot-the-crawler-with-caffeine-google-s-b9ef24d81524 (visited on 10/20/2020).

[61] *CNCF Survey 2019: Deployments are getting larger as cloud native adoption becomes mainstream*, Cloud Native Computing Foundation. [Online]. Available: https://www.cncf.io/wp-content/uploads/2020/08/CNCF_Survey_Report.pdf (visited on 10/22/2020).

[62] *Ingress*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/ (visited on 10/22/2020).

[63] *Securing Ingress Resources - How it works*, cert-manager. [Online]. Available: https://cert-manager.io/docs/usage/ingress/#how-it-works (visited on 10/22/2020).

[64] *ACME Configuration*, cert-manager. [Online]. Available: https://cert-manager.io/docs/configuration/acme/ (visited on 10/22/2020).

[65] *Rate Limits - Let's Encrypt - Free SSL/TLS Certificates*, Internet Security Research Group (ISRG), Mar. 5, 2020. [Online]. Available: https://letsencrypt.org/docs/rate-limits/ (visited on 10/22/2020).

[66] *Features restricted to secure contexts*, Mozilla Foundation, Jun. 3, 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Secure_Contexts/features_restricted_to_secure_contexts (visited on 10/22/2020).

[67] *Using Admission Controllers*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/ (visited on 10/22/2020).

[68] *kube-apiserver*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/ (visited on 10/22/2020).

[69] *Using RBAC Authorization*, Kubernetes - The Linux Foundation. [Online]. Available: https://kubernetes.io/docs/reference/access-authn-authz/rbac/ (visited on 10/22/2020).

[70] *Projects API*, GitLab Inc., 2020. [Online]. Available: https://docs.gitlab.com/ee/api/projects.html (visited on 10/22/2020).

[71] *GitLab Runner Docs*, GitLab Inc., 2020. [Online]. Available: https://docs.gitlab.com/runner/ (visited on 10/22/2020).

[72] *GitLab Container Registry administration*, GitLab Inc., 2020. [Online]. Available: https://docs.gitlab.com/ee/administration/packages/container_registry.html (visited on 10/22/2020).

[73] *Installation GitLab*, GitLab Inc., 2020. [Online]. Available: https://docs.gitlab.com/ee/install/README.html (visited on 10/22/2020).

[74] *GitLab as OAuth2 authentication service provider*, GitLab Inc., 2020. [Online]. Available: https://docs.gitlab.com/ee/integration/oauth_provider.html (visited on 10/22/2020).

[75] J. Riggins, *Miniservices: A Realistic Alternative to Microservices*, Jul. 11, 2014. [Online]. Available: https://thenewstack.io/miniservices-a-realistic-alternative-to-microservices/ (visited on 10/24/2020).

[76] K. G. Brown, *What's the right size for a Microservice?* Apr. 14, 2014. [Online]. Available: https://medium.com/@kylegenebrown/whats-the-right-size-for-a-microservice-bf1740370d47 (visited on 10/24/2020).

[77] M. Fowler, *FirstLaw*. [Online]. Available: https://martinfowler.com/bliki/FirstLaw.html (visited on 10/24/2020).

[78] ——, *MonolithFirst*, Jun. 3, 2015. [Online]. Available: https://martinfowler.com/bliki/MonolithFirst.html (visited on 10/25/2020).

[79] S. Newman, *Microservices For Greenfield?* [Online]. Available: https://samnewman.io/blog/2015/04/07/microservices-for-greenfield/ (visited on 10/25/2020).

[80] M. Fowler, J. Lewis, S. Newman, T. Palanisamy, and E. Bottcher, *MicroservicePremium*, May 13, 2015. [Online]. Available: https://martinfowler.com/bliki/MicroservicePremium.html (visited on 10/25/2020).

[81] S. Tilkov, *Don't start with a monolith ... when your goal is a microservices architecture*, Jun. 9, 2015. [Online]. Available: https://martinfowler.com/articles/dont-start-monolith.html (visited on 10/25/2020).

[82] *GitLab as an OAuth2 provider*, GitLab Inc., 2020. [Online]. Available: https://docs.gitlab.com/ee/api/oauth2.html (visited on 10/27/2020).

[83] B. Rhodes, *The Singleton Pattern - A Creational Pattern from the Gang of Four book*. [Online]. Available: https://python-patterns.guide/gang-of-four/singleton/ (visited on 12/22/2020).

[84] Auth0, *Prevent Attacks and Redirect Users with OAuth 2.0 State Parameters*. [Online]. Available: https://auth0.com/docs/protocols/state-parameters (visited on 12/18/2020).

[85] *Using Helm - Three Big Concepts*. [Online]. Available: https://helm.sh/docs/intro/using_helm/ (visited on 12/30/2020).