DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Designing Robust Interaction Frontend for Decentralized Edge Infrastructures

Daniel Mair

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor´s Thesis in Informatics

Designing Robust Interaction Frontend for Decentralized Edge Infrastructures

Entwurf eines robusten Interaktions-Frontends für dezentralisierte Edge-Infrastrukturen

Author:	Daniel Mair
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	Dr. Nitinder Mohan
Submission Date:	March 15th, 2022

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, March 15th, 2022 Daniel Mair

Acknowledgments

First of all, I would like to thank Prof. Dr-Ing. Jörg Ott, the chairholder of the Connected Mobility Chair at the Technical University of Munich, for making this work possible. Many thanks also to my supervisor Dr. Nitinder Mohan who always took time for me and could always help me with my questions. I would also like to thank all the survey participants and all people who supported me in this work. A special thanks to Giovanni Bartolomeo from the EdgeIO working group who was always able to help me with my questions. Last but not least, many thanks to my family and friends who have always supported me in my life.

Munich, March 2022

Abstract

In recent years, edge computing has become increasingly popular. It replaces central cloud computing and brings computing power closer to the end-user. Due to its popularity, many edge computing frameworks have emerged to automate the deployment of new services or control the infrastructure. Since the whole topic is relatively new, the focus in the development of these frameworks has always been on automation. As a result, the control of this framework has often been neglected.

This thesis is about the design and development of such a control. Both a graphical and a command-line-based control were created to increase the usability of such systems. The user interface was created specifically for the EdgeIO framework in this work. Multi-user operation was enabled without ignoring the security aspects of the system. In addition, a survey was created to evaluate the developed system with domain experts, to subsequently analyze the weaknesses and improve it in future iterations. The survey concluded that the developed system provides a good overview of the framework, but there is still room for improvement. The system could support the user more, and graphical inconsistencies could be corrected.

This work has laid the foundation for the user interface, which can be extended by more functions and additional views in the future.

Zusammenfassung

In den letzten Jahren wurde Edge Computing immer populärer. Es löst das zentrale Cloud Computing ab und bringt die Rechenleistung näher an den Endverbraucher. Durch die große Beliebtheit entwickelten sich viele Edge Computing Frameworks, welche das Ziel haben, das Erstellen von neuen Applikationen oder das Steuern der Infrastruktur zu automatisieren. Da die gesamte Thematik relativ neu ist, wurde der Fokus bei der Entwicklung dieser Frameworks immer auf die Automatisierung gelegt, wodurch die Steuerung dieser Frameworks häufig vernachlässigt wurde.

In dieser Arbeit geht es um das Design und die Entwicklung einer solchen Steuerung. Sowohl eine Grafische als auch eine Kommandozeilen-basierte Steuerung wurde erstellt und soll die Benutzerfreundlichkeit solcher Systeme erhöhen. In dieser Arbeit wurde die Benutzeroberfläche speziell für das EdgeIO Framework erstellt. Dabei wurde der Mehrbenutzerbetrieb, ohne die Sicherheitsaspekte des Systems zu vernachlässigen, ermöglicht. Zusätzlich wurde eine Umfrage erstellt, die das entwickelte System mit erfahrenen Probanden evaluiert, um anschließend Schwächen des Systems zu analysieren, die in zukünftigen Iterationen verbessert werden können. Die Umfrage ergab, dass das entwickelte System einen guten Überblick über das Framework liefert, es jedoch noch Verbesserungspotenzial in der Unterstützung der Benutzer gibt und grafische Inkonsistenzen verbessert werden sollten.

Mit dieser Arbeit wurde die Basis für die Benutzeroberfläche implementiert, die in Zukunft mit mehr Funktionen und weiteren Ansichten erweitert werden kann.

Contents

1 Introduction

Edge computing refers to the enabling technologies that allow computation to be performed at the network edge so that computing happens near data sources. This can reduce the amount of network bandwidth required by the sensor or drastically reduce the latency between the application on the edge device and the end-user. An edge device can be any computing or networking device between the data sources and the cloud. For example, an edge device could be a Raspberry PI, between different sensors and the cloud. The Raspberry PI could now perform various tasks, such as processing, storing, caching, or load balancing, yielding many benefits.

Due to the popularity of edge computing, more and more edge orchestration frameworks are emerging. The goal of these frameworks is to take advantage of edge computing and provide users with the best possible performance. However, these frameworks focus mainly on the architecture and rarely on the user interface and a developer-friendly environment. As the number of different edge orchestration frameworks grows, the number of users of such systems will also increase. Consequently, the large number of users with different roles and requirements will bring challenges in terms of the usability of edge systems. No matter how powerful the edge orchestration framework is, it would be difficult to capture the interest of users if a customer does not receive a good user experience.

The key point is how the interface between the framework and the end-user should be designed to make service delivery easier and more convenient. In most cases, a command-line interface (CLI) or application programming interface (API) is provided to control a system of this type. However, a graphical interface can make control even more accessible, especially for inexperienced users, as it can be more intuitive in many cases.

From the end-user perspective, user experience already enjoys full attention in cloud computing [1]. This work attempts to bring this attention to edge orchestration frameworks as well. In particular, this work creates a user interface for the EdgeIO framework, an orchestration framework developed by the Technical University of Munich.

1.1 Research Questions

In this thesis we want to investigate and answer the following research questions.

RQ1: **How can a large number of edge devices be controlled with a simple UI in a distributed deployment infrastructure?** Edge orchestration frameworks can manage a large number of different devices and services. Despite the large number, it should be easy for the user to keep track of and manage the resources. This work wants to determine how such systems should be designed to allow easy control.

RQ2: **How should UI be designed in edge frameworks to be developer-friendly?** The usability of a system is crucial for its success. In order to guarantee good usability, certain requirements must be met. Therefore, this thesis analyzes how other edge orchestration systems ensure usability. Additionally, we define our own functional and non-functional requirements.

RQ3: **What must be considered that despite multi-user use the framework remains secure?** By introducing the use of multiple users with different user roles, it must be ensured that each user is only allowed to execute the functions for which he has the permissions. Also, a user is only allowed to view and modify their own configurations. We are trying to figure out how to enable this separation and restrict the users with their functions.

This thesis considers these research questions in the design and implementation of a user-friendly user interface to control the EdgeIO framework.

1.2 Contribution

We have created two different control options for the EdgeIO framework. First, the command-line tool *edgeiocli*. This tool allows the control of the system in the terminal. And second, we designed and developed a graphical user interface to control the system. Both systems are described in detail in chapter 4. In addition, we added multi-user support to the existing EdgeIO framework and various security aspects like access tokens. Furthermore, we created a survey with experienced people in this field to evaluate the GUI.

1.3 Overview of Content

In the beginning, this thesis introduces in chapter 2 the EdgeIO framework on which this work is based. Existing cloud-edge orchestration platforms from the industry and research domain, like AWS Greengrass, Kubernetes, EdgeX, and other edge frameworks are presented and analyzed. The requirements of our system are described in chapter 3. First, the already existing system is analyzed and how it can be extended. We also define the general design, which functions are offered to the user, and briefly present the final user interface. Chapter 4 contains the system design and implementation of the proposed EdgeIO user interface prototype. The individual components, used technologies, and security aspects are described here. In chapter 5 the system is evaluated, beginning with the evaluation of the requirements we set and how they were met. In addition, the user survey is presented in this chapter. Finally, chapter 6 concludes this thesis, and future work is proposed.

2 Background

This chapter provides information on the theoretical background of this thesis. The architecture and components of EdgeIO are described, as well as the Service Level Agreements (SLA) to define a new service. Furthermore, a brief overview of how the current version works, using API requests and what features it offers, is provided. The remaining part presents different solutions from the industry like AWS Greengrass or Kubernetes but also prototypes from the research community and how they control their framework.

2.1 EdgeIO

EdgeIO [2] is an edge orchestration platform with several components. They can be deployed independently, making the framework very flexible and scalable. A large number of computing nodes, even without a public IP address, can be connected to the framework and assigned tasks. The allocation goes through several phases and can take into account the capacity and technological capabilities of the computing nodes.

EdgeIO was developed at the Connected Mobility chair at the Technical University of Munich and has been under continuous development ever since.

In the next sections, we will take a closer look at the design of EdgeIO, its key components, and how the current version works. However, we will mainly focus on the Root Orchestrator since it is the central unit of EdgeIO and is aware of all participating clusters. Also, our later developed user interface will only communicate with the Root Orchestrator, which will take care of the communication with the rest of the framework components.

2.1.1 System Design

The EdgeIO framework was designed from the beginning to enable edge applications. It has a horizontal structure that allows the framework to scale to any size without significant investments in deployment. Figure 2.1 shows the high-level architecture of EdgeIO.

The Root Orchestrator is one of the components of the framework. It is the central control plane. Another component is the Cluster Orchestrator. Any number of clusters can join a Root Orchestrator and then receive tasks from it. A Cluster Orchestrator can then, in turn, take on any number of resources (workers) and manage them. The individual components are described in more detail in the following sections.

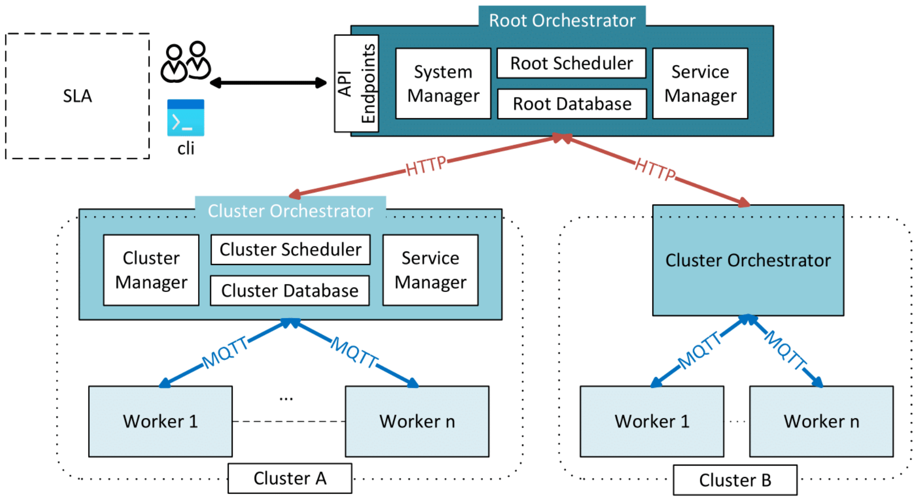


Figure 2.1: EdgeIO architecture adapted from [3].

Root Orchestrator

The Root Orchestrator consists of several components. One of them is the System Manager, which has several API endpoints to allow users to perform operations in the framework, such as deploying a new service. Our frontend will later provide exactly this interface between the users and the System Manager and should improve the usability. Other components are the database to store all relevant information about the participating cluster and workers within the system, a scheduler that calculates a placement for a particular service, and the service manager, which is responsible for monitoring the state of all operational services.

All components within the Root Orchestrator can be installed on different devices, which increases the reliability and makes the whole system much more robust. It also allows the system to scale much better. To enable this distribution, the individual components communicate via APIs. Users can deploy a service in the EdgeIO framework by specifying a high-level SLA and sending it to the System Manager, who then takes further steps to find a suitable cluster for the service. For the Root Orchestrator, the participating clusters are like worker nodes that receive scheduling decisions and deploy tasks. Once a suitable cluster has been found, the service can be forwarded to the cluster. The Root Orchestrator receives monitoring information from the cluster at certain intervals. If the user wants to change something, he can manage or delete a service with the help of API requests to the System Manager.

Cluster Orchestrator

The Cluster Orchestrator is very similar to the Root Orchestrator. However, the System Manager has been replaced with a Cluster Manager, whose responsibility is to manage a group of workers. The scheduler calculates task placements, and a database is used to store information about the participating worker nodes. In addition, the Cluster Orchestrator has a message broker that enables communication with the worker via MQTT. The Cluster Orchestrator either publishes commands for the worker or reads the data (CPU, memory) published by the worker nodes. This data is then summarized by the Cluster Orchestrator and sent to the Root Orchestrator at certain time intervals to include the utilization of the cluster in his scheduling calculations.

Worker

A worker can be any computing device with an operating system and the running Node Engine software. New workers can be added to a cluster using the Cluster Manager. First, there is an initialization phase where the cluster and worker exchange various information. Once this phase is complete, the worker can also be considered in the scheduling process of the cluster. In addition, as already mentioned, the worker publishes its monitoring data at regular intervals.

2.1.2 SLA

The current implementation of EdgeIO consists of two different scheduling steps. First, the scheduler of the Root Orchestrator finds a suitable cluster for the application. In the second step, the scheduler of the cluster orchestrator selects a suitable worker. The schedulers make their decision based on the application specified by the developer. An application consists of one or more microservices. A microservice, in turn, has certain requirements that must be met. These requirements and other information about the microservices to be deployed can be specified by the developer in the Service Level Agreements (SLA) [4]. Developers who want to deploy an application in EdgeIO send their code with the SLA in JSON format to the System Manager. The SLA, shown in listing 2.1, contains various requirements such as the number of CPU cores, memory capacity, bandwidth, etc.

```
1  {
2    "api_version" : "v0.3.0",
3    "customerID" : 10000000001,
4    "applications" : [
5      {
6        "applicationID" : 1000010001,
7        "application_name" : "Example Application",
8        "application_desc" : "No description here",
9        "microservices" : [
10         {
11           "microserviceID": 4,
12           "microservice_name": "Micro A (4)",
13           "virtualization": "container",
14           "memory": 4096,
15           "vcpus": 2,
16           "vgpus": 0,
17           "vtpus": 0,
18           "bandwidth_in": 1000000,
19           "bandwith_out": 8000,
20           "storage": 1000,
21           "code": "https://example.com/repositories/example/code.py",
22           "state": "https://path.to.root.orchestrator.customer.application/state_1.json",
23           "port": 80,
24           "added_files": [
25             "https://example.com/repositories/example/asset_1.jpg",
26           ],
27           "constraints": [
28             {
29               "type": "latency",
30               "area": "munich−1",
31               "threshold": 100,
32               "rigidness": 0.99,
33               "convergence_time": 300
34             }
35           ],
36           "connectivity": []
37         }]
38     }
39   ],
40   "args" : []
41 }
```

Listing 2.1: EdgeIO JSON file to configure a service.

The SLA in the EdgeIO framework can be roughly divided into five different parts. First, in line 3, the customer is specified. After that, an application is defined. First, in lines 6 to 8, there is some information about the application. After that, the microservices of the application are defined. Everything up to line 26 defines the worker's requirements and configuration information. After that, in lines 27 to 35, the constraints are specified.

They describe the requirements that must be offered to the user at least. Furthermore, in line 36, connection constraints can be defined. They are very similar to the previously mentioned constraints, but these requirements must be respected between two services.

If the worker violates these requirements several times, the Cluster Orchestrator or the Root Orchestrator tries to find a new worker for this service. This ensures optimum performance for the user most of the time.

2.2 Related Orchestration Systems for Edge

This section will look at some existing orchestration frameworks and their user interface. We look at the design, what they are capable of, and the main difference between the systems. The goal is to understand better how these state-of-the-art orchestration frameworks work. They are used by thousands of people and can give us some ideas for designing our user interface. It should be mentioned that the systems presented are much more complex than our system due to human resources. However, we can learn a lot from their designs for this very reason.

2.2.1 AWS Greengrass

AWS Greengrass [5] is a service of Amazon Web Services that allows the user to create a local AWS IoT server to which different IoT devices can connect and perform computation such as AWS Lambda functions. A user can deploy his own Lambda functions on the Greengrass core, which is usually implemented on a dedicated device. To enable communication between an IoT Device and AWS Greengrass, they must be configured in the same group. A general overview of AWS Greengrass is depicted in the following fig. 2.2. Each IoT device can connect to AWS Greengrass to perform computation without connecting to the AWS server. IoT devices and AWS Greengrass can only communicate within the Greengrass Group:

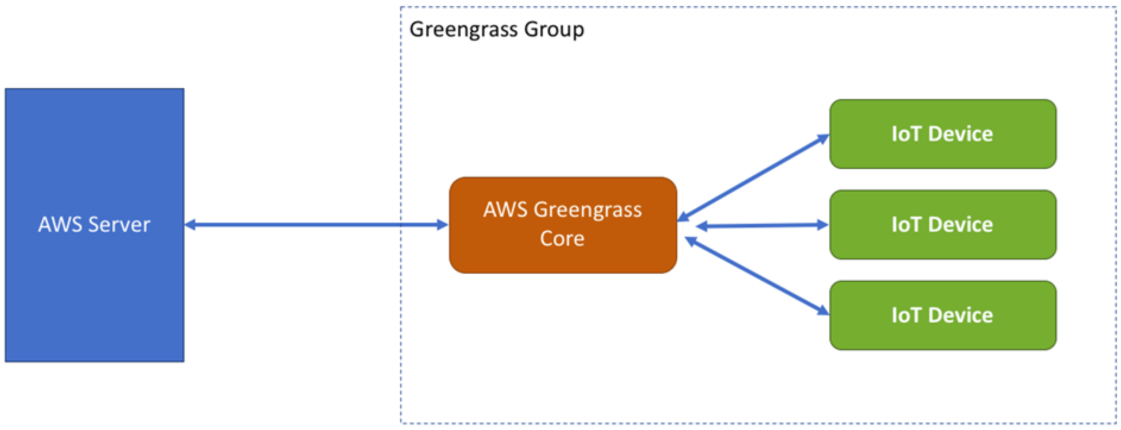
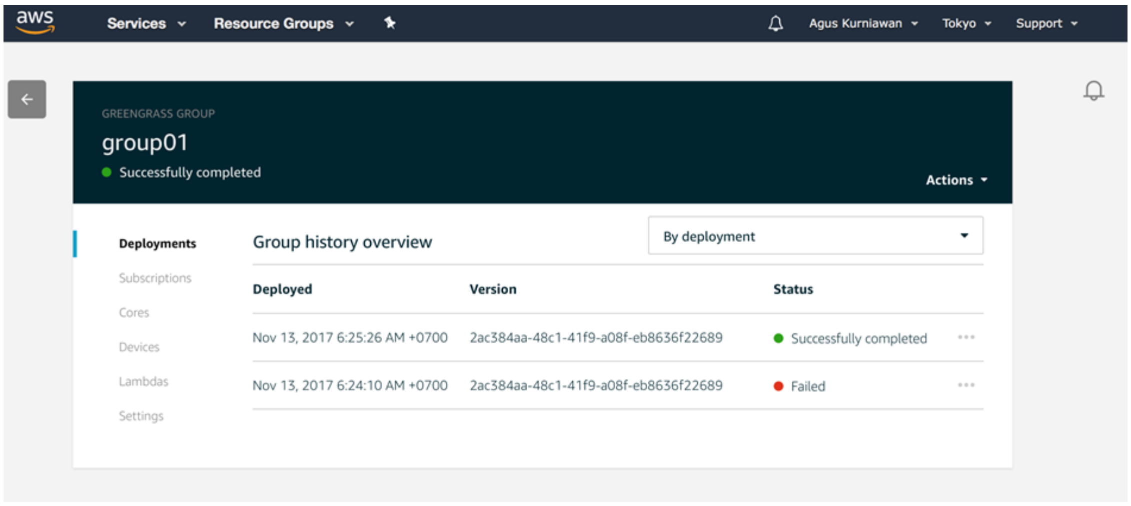


Figure 2.2: AWS Greengrass connection overview. [6]

All AWS Greengrass and IoT devices are managed over the AWS console, the central dashboard for all AWS services. To add a new Lambda function to Greengrass, the user has to upload his function to AWS, configure the Lambda and deploy it to the Greengrass core. Once a service is deployed, it is possible to manage the service and view the status like in fig. 2.3.



Deployed	Version	Status
Nov 13, 2017 6:25:26 AM +0700	2ac384aa-48c1-41f9-a08f-eb8636f22689	● Successfully completed
Nov 13, 2017 6:24:10 AM +0700	2ac384aa-48c1-41f9-a08f-eb8636f22689	● Failed

Figure 2.3: AWS Console sample of a successful deployment. [6]

2.2.2 Kubernetes

Kubernetes [7] is an open-source system for automating deployment, scaling, and management of containerized applications. It was originally developed by Google and is now a CNCF project. Kubernetes is nowadays the de-facto standard in cloud computing technologies and consists usually of a master node and several worker nodes. The master node is the central unit of Kubernetes. It consists of several components, such as an API server, a scheduler, and a controller manager. [8]

The API server can be used to create, update and delete resources. The code listing 2.2 shows the specification in JSON format to create a pod. A pod in this context is a group of one or more containers that are highly interdependent and share resources. The resources of a pod always reside on the same (virtual) server, are scheduled together, and run in a common context.

```
{
    "kind": "Pod",
    "apiVersion": "v1",
    "metadata":{
        "name": "nginx",
        "namespace": "default",
        "labels": { "name": "nginx" }
    },
    "spec": {
        "containers": [{
            "name": "nginx",
            "image": "nginx",
            "ports": [{
                "containerPort": 80
            }]
        }]
    }
}
```

Listing 2.2: Kubernetes JSON file to deploy a pod. [8]

After the pod is deployed, API endpoints can be used to monitor, delete and debug the group of containers. All the functions can be executed using the command-line tool, but Kubernetes also provides its built-in dashboard, which is deployed as a service in a separate pod.

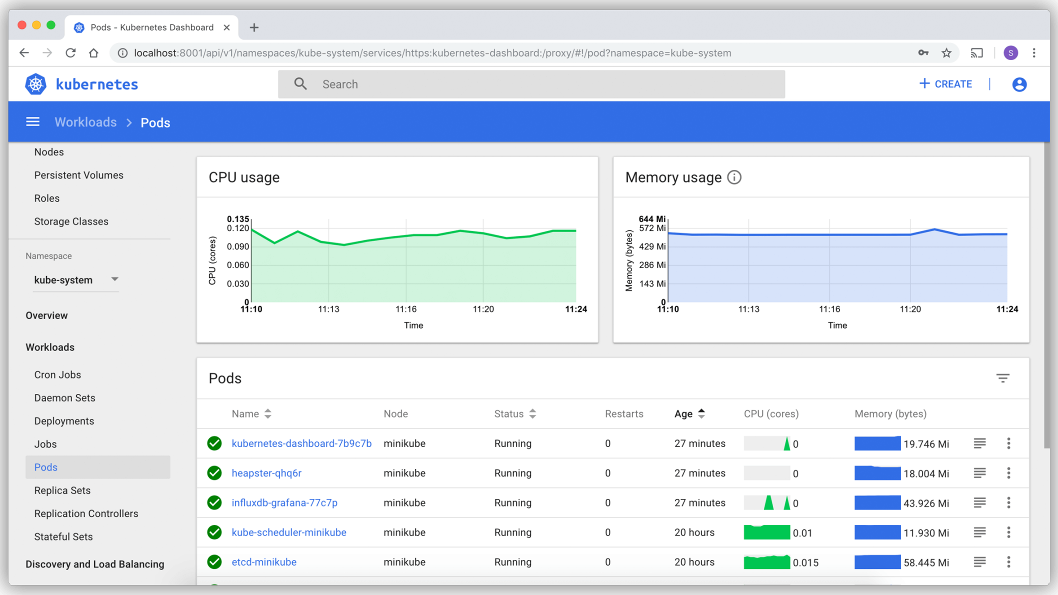
Kubernetes Dashboard

The Kubernetes dashboard [9] is an easy-to-use interface and covers everything from the cluster to the individual container. To deploy a new application using the dashboard you can upload the proper YAML or JSON file as shown in listing 2.2.

The monitoring also works via the dashboard, where all information about the nodes is either summarized as in fig. 2.4 or displayed individually. The Dashboard offers several top-level categories:

- Admin
- Workloads
- Services and Discovery
- Storage
- Config

In these categories, you get more detailed information on each area that could be relevant. You can also filter everything by a specific namespace.



Name	Node	Status	Restarts	Age	CPU (cores)	Memory (bytes)
kubernetes-dashboard-7b9c7b	minikube	Running	0	27 minutes	0	19.746 Mi
heapster-qhq6r	minikube	Running	0	27 minutes	0	18.004 Mi
influxdb-grafana-77c7p	minikube	Running	0	27 minutes	0	43.926 Mi
kube-scheduler-minikube	minikube	Running	0	20 hours	0.01	11.930 Mi
etcd-minikube	minikube	Running	0	20 hours	0.015	58.445 Mi

Figure 2.4: Kubernetes Dashboard with CPU and memory history of a pod. [8]

2.2.3 EdgeX Foundry

EdgeX Foundry [10] is an open-source, interoperable framework for IoT edge computing. It interacts with various physical devices such as sensors, actuators, or other IoT devices. The devices can be managed, and data can be collected to process the data locally or send it to the cloud. It is designed to be agnostic to hardware, CPU, operating system, and application environment and can run natively or in docker containers. EdgeX consists of a collection of microservices, which allows services to scale up and down based on device capability. These microservices can be grouped into four service layers and two underlying augmenting system services, as depicted in fig. 2.5. Each layer consists of different components, and each component offers its own Restful API.

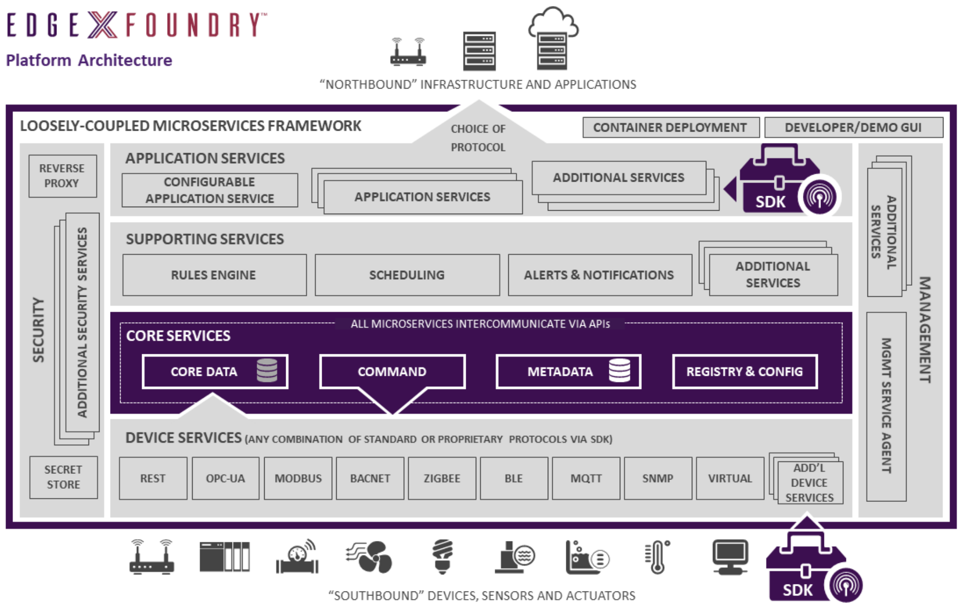


Figure 2.5: EdgeX architecture. [11]

The augmenting system services are the most interesting for our work, as they include service system management and security. The security service protects the data of IoT devices connected to the EdgeX Foundry, and the system management service provides managements operations such as installation, upgrade, starting, stopping, and monitoring. To enable developers to interact with the framework, EdgeX provides a command-line interface and a graphical user interface, which is called *EdgeX UI*. It has many similarities to the Kubernetes dashboard, such as the same structure, but this interface is much simpler and has fewer features. The Frontend was written in Angular and can run in a docker. It allows the user to manage EdgeX objects, start, stop or monitor the EdgeX services and investigate the configuration of an EdgeX service. However, the GUI is limited currently to be used as a developer tool or as a demo and not in production because e.g. the GUI does not offer authentication or cannot reach the services outside the docker network. Nevertheless, the EdgeX community is improving its system constantly and will offer a GUI for the production environment in the future.

2.2.4 More Edge Computing Projects

In addition to the projects described earlier, there are many more projects in different domains trying to manage edge, fog, and cloud infrastructures. Some of them are briefly analyzed in this section.

K3s [12] is a lightweight version of Kubernetes. The goal of the project is to significantly reduce the memory footprint and provide a single binary package that can be installed on small edge nodes. The complete framework is controlled by the "k3s server".

And since it is built on top of Kubernetes, it can also be operated with the Kubernetes dashboard, described in section 2.2.2. All functions that are available in Kubernetes are also available in this dashboard.

Another system that can be used with the Kubernetes dashboard is KubeEdge [13]. It is a plugin for Kubernetes that allows computation at the edge. The cloud component is Kubernetes, which communicates with the edge component via a tunnel. The edge component is a lightweight agent that runs in each edge node and provides all the deployment and management operations for the pods.

The IoFog [14] project from the Eclipse Foundation aims to enable users to deploy services in edge infrastructures in the same way as in the cloud. To achieve this, each node runs its own IoFog agent, and several control services can be used to manage the deployed services. The CLI tool *iofogctl* can be used to install, configure and manage the system.

The fog05 [15] is another system for managing and deploying applications in edge and fog environments. Also, this project is developed by the Eclipse Foundation and makes it possible with its decentralized architecture to manage different nodes end-to-end. Fog05 can also be used with a command line tool. In this case, it is called *fosctl*.

2.2.5 Overview of all considered frameworks

In the table 2.1 one can see all the important information about the considered frameworks.

Framework	Operational area	GUI	CLI
AWS Greengrass	Edge/IoT	AWS console	AWS CLI
Kubernetes	Cloud	Kubernetes Dashboard	kubectl
EdgeX Foundry	Edge/IoT	EdgeX UI	EdgeX-CLI
K3s	Edge	Kubernetes Dashboard	kubectl
KubeEdge	Edge	Kubernetes Dashboard	kubectl
IoFog	Edge	-	iofogctl
Fog05	Edge/Fog	-	fosctl

Table 2.1: Overview of all frameworks considered.

All frameworks provide a command-line tool that allows controlling the system. In addition, most systems provide API endpoints to perform operations, and some also provide a graphical user interface that can access the API endpoints. The frontend can then monitor the running services of the systems and perform various operations. All considered frontends were very clearly designed and increased the usability of the system. In all systems, however, the users can freely choose whether the graphical user interface is used or not. With Kubernetes and EdgeX, it is necessary to install the dashboard separately. They can be installed on any device, preferably in the cloud. In AWS Greengrass, installing the graphical user interface is not necessary since the AWS console is already available in the cloud. Nevertheless, all systems can also be used without an internet connection, but then the UI should be installed on a reachable device in the network.

In many cases, where no graphical user interface is offered for data visualization, the open-source tool Grafana [16] can be used. It allows the graphical representation of data from various data sources, such as the data of the different services that can be read from the database.

3 Requirement Analysis

This chapter deals with the requirements of our frontend. First, we will take a look at the current system and analyze how it works. Afterward, we analyze what is already good about the system, where we can make improvements, and how we can best build our UI on top of it. Then we define our own requirements for our system and present an example interaction between the user and the system. Finally, we introduce our developed user interface.

3.1 Overview

We want to develop a frontend for the EdgeIO framework in which it should be possible for different types of users to operate the framework. The user should be able to define, deploy or delete new services. Most important is the demarcation of the different users. Depending on the type of user, he has different rights and should only see his configurations. These requirements allow some interpretation possibilities, and there are different ways to implement them. Therefore, in the next step, we will analyze and define the requirements in more detail.

3.2 Existing System

In order for a user to deploy a service in the current frontend, he must create an SLA, as discussed in section 2.1, and send this via an API request to the System Manager. This was mostly done via a CURL command like in listing 3.1, which is the only way to deploy a service.

```
curl -F file=@'deploy.yaml' http://ip_of_system_manager:10000/api/deploy
```

Listing 3.1: CURL command to deploy a service.

In addition, there is no division of different users. Anyone who has access to the System Manager can make API requests and thus create new applications or view and possibly modify already created services.

As we have seen in the analysis of the other systems presented in section 2.2, many of these frameworks have a command-line tool to perform certain operations. Other systems like Kubernetes have a web interface to perform these operations. Since EdgeIO already provides API endpoints, it makes sense to use them and implement a web interface. However, users should still be able to use the system in the terminal, so it also makes sense to create a similar command-line tool as the other systems provide. Thus, the framework offers two different control options for the user. All functions are described in more detail in section 3.3.1.

3.3 Proposed System

So far, we have gathered some information on how other solutions designed their UI, and we have looked at how the current version of EdgeIO works. To summarize ideas, we will do the following:

- Implement a website with user authorization.
- Differentiate between different users.
- Provide an extensive form for service configuration.
- Various user and service management functions.
- Provide command-line version.

We now specify the required functionalities of the UI. Therefore, we will define and formalize individual requirements for our system as functional requirements (FR) and non-functional requirements (NFR).

3.3.1 Functional Requirements

Functional requirements capture the intended behavior of the system. This behavior may be expressed as services, tasks, or functions the system is required to perform [17]. In our case, the functional requirements define how the system should behave when the user interacts with it.

FR1 Different roles: The system differentiates between the following three different user roles. Each role has a corresponding user interface.

Application Provider: Users are allowed to create applications and services, which can be deployed, edited, and monitored.

Infrastructure Provider: Users can add new workers or an entire cluster to the Root Orchestrator and manage these devices.

Admin: This user performs both roles mentioned – Application Provider and Infrastructure Provider – in addition he can add and manage new users.

FR2 Authentication: Only registered users are able to use the system.

FR3 Form configuration: Users can enter the SLAs of a service using a form and deploy the service without creating external files.

FR4 Define dependencies between microservices: Dependencies between services within an application can be defined and displayed in a graph.

FR5 Headless version: The whole system can also be used without a website in the terminal with a command-line-tool.

FR6 Service management: It is possible to manage the created services. A service can be modified, deployed, or deleted.

FR7 User management: If a user holds the admin role, he can manage all users. The admin can delete, change the roles of existing ones, or create new user.

3.3.2 Non-Functional Requirements

Non-functional requirements can play a critical role during system development. Performance, maintainability, portability, and robustness are non-functional requirements, just to name a few. It is crucial to define them before programming as they can have a decisive influence on the system design. [18]

However, they specify not the functionality of the system. The user and the developer define Non-functional requirements to ensure the system's quality. Since there can be so many non-functional requirements, we focus in our system on the most common, namely the URPS model. That stands for usability, reliability, performance, and supportability.

Usability

NFR1 Easy to use: The user interface is easy to use and follows certain design guidelines. New users do not have to get used to the system first.

NFR2 Complete: No matter whether the user uses the CLI or the web client, the system's current state is always reflected.

Reliability

NFR3 Robustness: All configurations created with the form are in the correct format and can therefore not create problems.

Performance

NFR4 Response time: All pages on the website are displayed with no or barely noticeable delay.

Supportability

NFR5 Responsive: All common screen sizes are supported.

NFR6 Extensibility: The system is implemented in modules, so it is easy to extend in the future.

3.4 System Models

The term "system model" is used in many different domains, in various forms with different meanings [19]. But in our case, we use the system model to represent the previously defined requirements in a model.

First, we generate a use-case-diagram (fig. 3.1) to show how different users might interact with the system and how the system must behave as a result. Afterward, we use a flow chart diagram (fig. 3.2) to describe a certain user scenario. These diagrams help to understand the entire system better.

3.4.1 Use Case

As described in FR1, our frontend should support multiple user roles. Therefore, it depends on the role of the user, which tasks can be performed. However, in this model, we limit ourselves to the admin and the application provider. For each user, it is necessary to log in to the dashboard (FR2). All functions can be executed by both users. Only the user management functions (FR7) are reserved for the admin. A user can create, edit, delete, deploy a service, and of course, view the status of the service (FR3, FR6). It does not matter if the user uses the graphical user interface or the CLI. All these functions are available in both variants (FR5). Dependencies between services within an application can be defined. If the user uses the graphical user interface, he can also view these dependencies of the individual services in a graph (FR4).

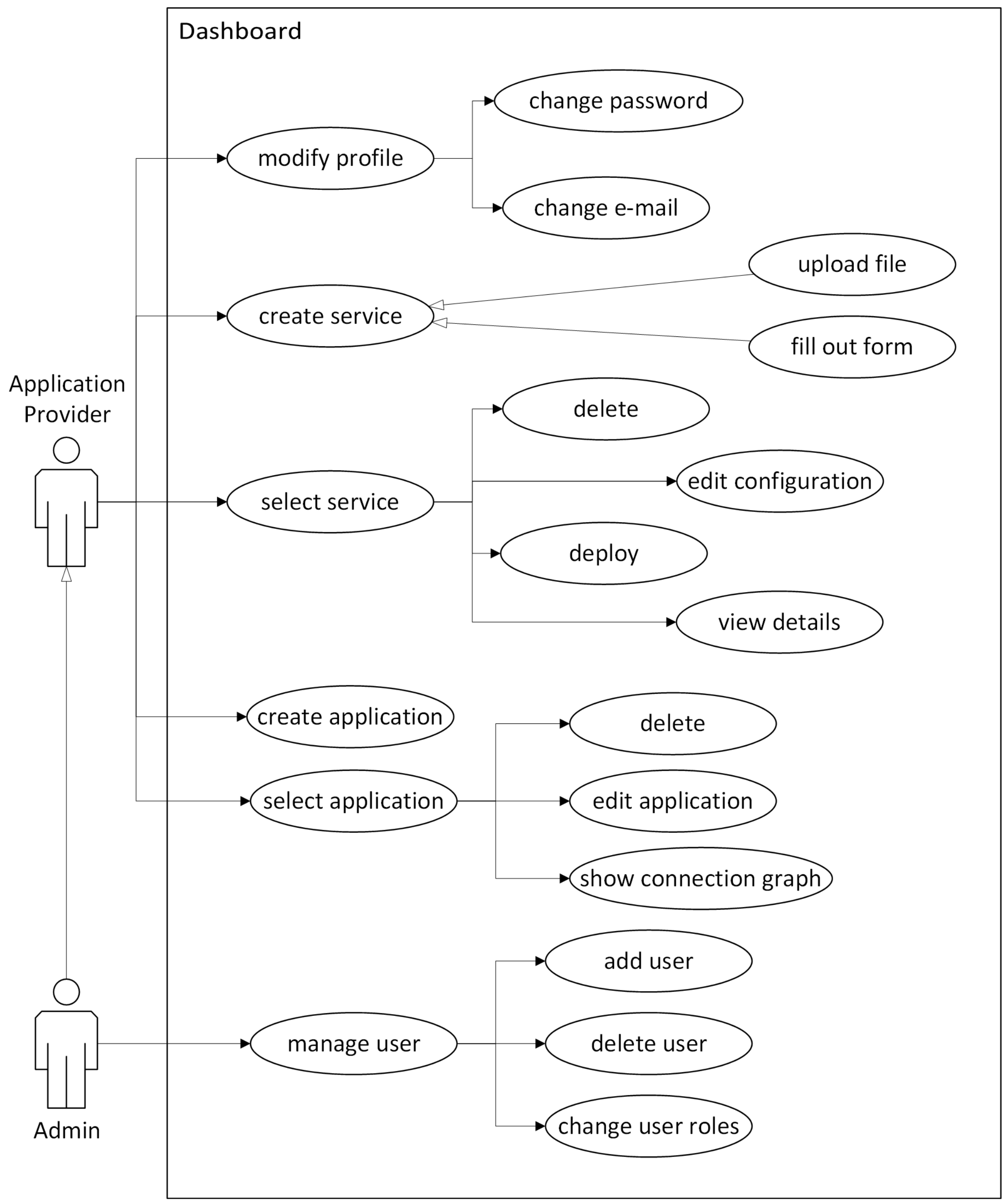


Figure 3.1: Use case model of the admin and the application-provider.

3.4.2 Flow chart of the login and deployment process

The Flow Chart Diagram in fig. 3.2 is used to present the workflows of a particular process. In this case, we present a user's workflow who logs in to the system, creates a new service, and then deploys it. First, the user must log in to access the dashboard. If the authentication is successful, the user is able to view the dashboard. Here, a corresponding application can be selected in which the new service should be created. Now the user can create a new service, either by using the form input or by uploading an already existing configuration. If the configuration is correct, the service will be created in the system and can then be deployed afterward.

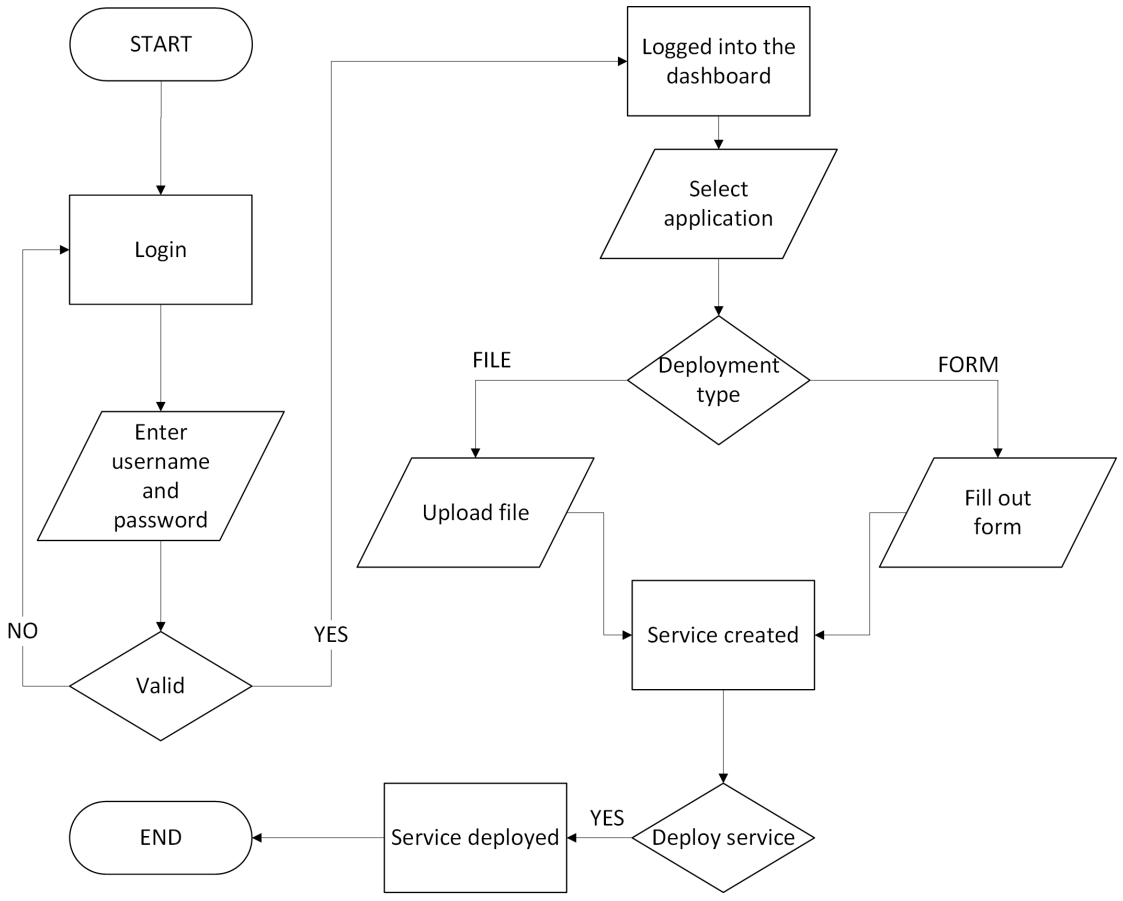


Figure 3.2: Flow chart diagram of the login and deployment process.

3.5 User Interface

This section will introduce the user interface and the different components it contains. First, we will describe the general structure and then present the two most important views: the application view and the configuration view. Due to similarities to the presented views, we do not have a closer look at the other views.

The EdgeIO frontend was designed using the Angular [20] framework with Angular Material [21]. Angular is an open-source TypeScript-based frontend web application framework, and Angular Material is a comprehensive collection of UI components for Angular. This collection of UI components allows designing a consistent and familiar user interface with just a few lines of code. Since Angular applications consist of several components and modules, the frontend can be implemented in a very modular way. Google developed Angular and Angular Material. Because of that, it is also used in many Google applications. The Kubernetes framework, described in section 2.2.2 also uses Angular and Angular Material. This means that users, who have already used such applications, do not need a long time to get used to our frontend, as everything looks very familiar.

For the arrangement of the individual buttons, we have followed the common conventions. The frontend can be divided into three structurally different areas: the header, the side-nav, and the content area.

I The header (fig. 3.3 (1)) contains the EdgeIO logo on the left side. With this, it is always possible to come back to the main dashboard. The links to all user-related pages are on the right side of the header. Here, the profile can be viewed, the admin can manage other users, and the user can log out from the dashboard.

II On the left side of the dashboard, the side-nav is located (fig. 3.3 (2)), where the corresponding application can be selected, edited, or a new application can be created. It also contains links to the help page and the survey.

III The area in the middle is the content area (fig. 3.3 (3)). Here the user can see the content of the corresponding page. So, if the view is changed, only this area changes, and the header and the navbar remain unchanged. The two most important pages are the application view (fig. 3.3) and the configuration view (fig. 3.4).

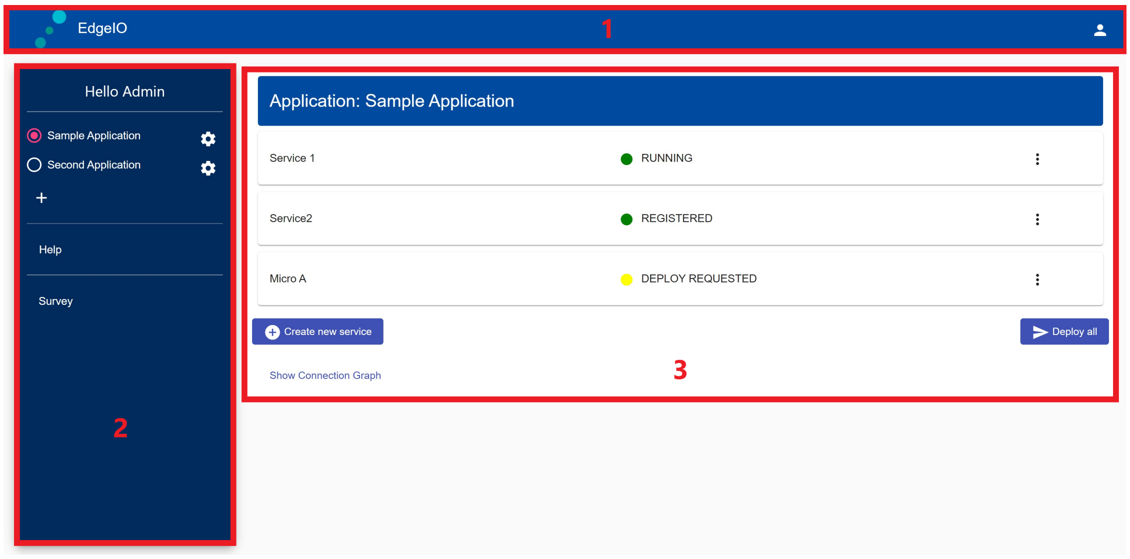


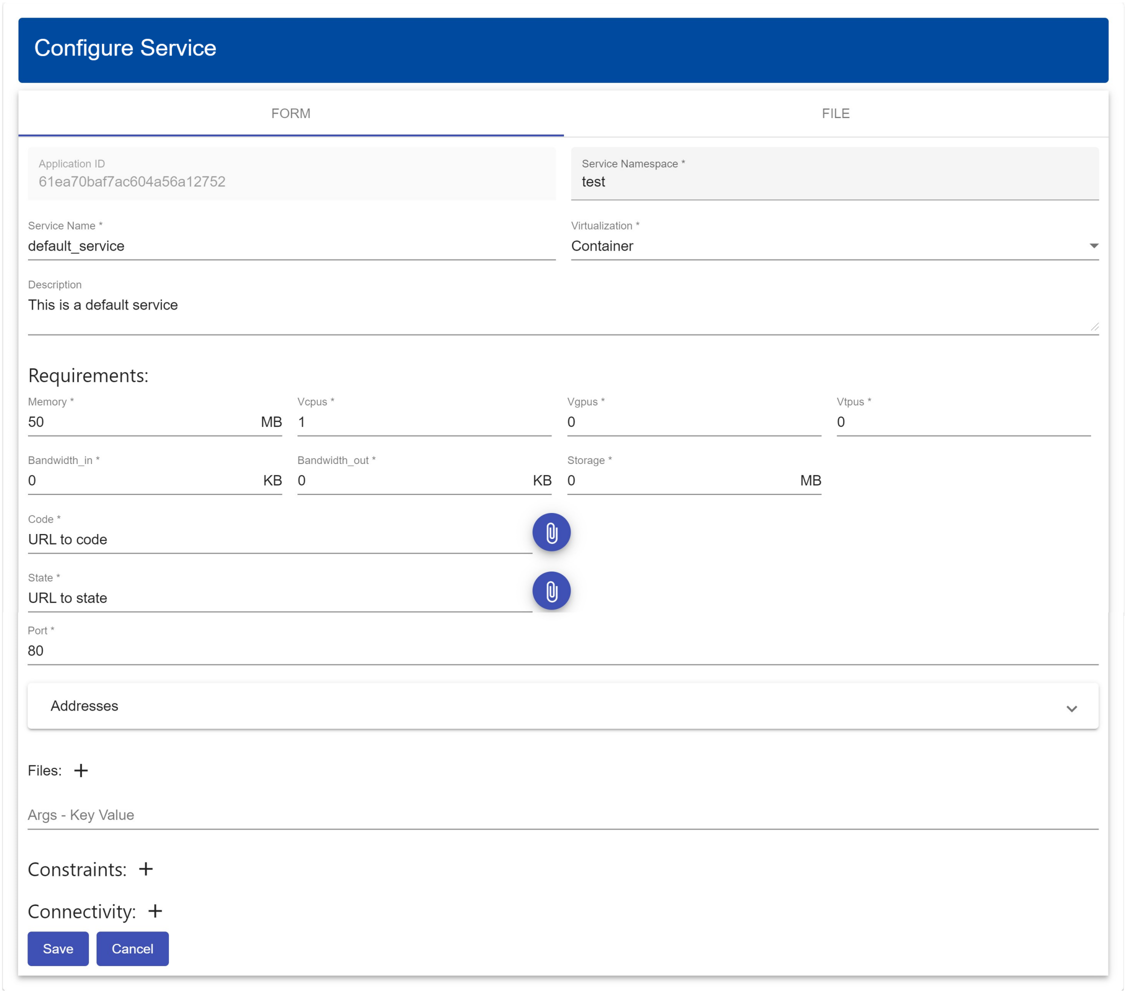
Figure 3.3: Dashboard main page with selected application.

In the application view, all services of the currently selected application are displayed. A service is displayed as a small component and contains the name, the status, and further links to manage the service. In addition, the application view contains the connection graph, where all connections between the individual services are displayed and can be managed.

A service can have one of the following status:

- **REGISTERED:** The service has been successfully created in the frontend, but has not been deployed yet.
- **DEPLOY REQUESTED:** After the user contacted the system to deploy a service.
- **SCHEDULED:** When the scheduler took a decision.
- **RUNNING:** When the service is running.
- **FAILED:** If an error came up.

In the configuration view, a large input form is displayed. This form allows the user to configure a service and define all the requirements. The input data is used to create an SLA as described in section 2.1.2. Only correctly defined services can be sent from the web client to the System Manager to increase robustness. The configuration of an already created service can be downloaded. For simple replication, developers can upload this or self-written configurations in the dashboard again to create a new service.



Configure Service

FORM | FILE

Application ID: 61ea70baf7ac604a56a12752	Service Namespace *: test
Service Name *: default_service	Virtualization *: Container

Description: This is a default service

Requirements:

Memory *: 50 MB	Vcpus *: 1	Vgpus *: 0	Vtpus *: 0
Bandwidth_in *: 0 KB	Bandwidth_out *: 0 KB	Storage *: 0 MB	

Code *: URL to code

State *: URL to state

Port *: 80

Addresses

Files: +

Args - Key Value

Constraints: +

Connectivity: +

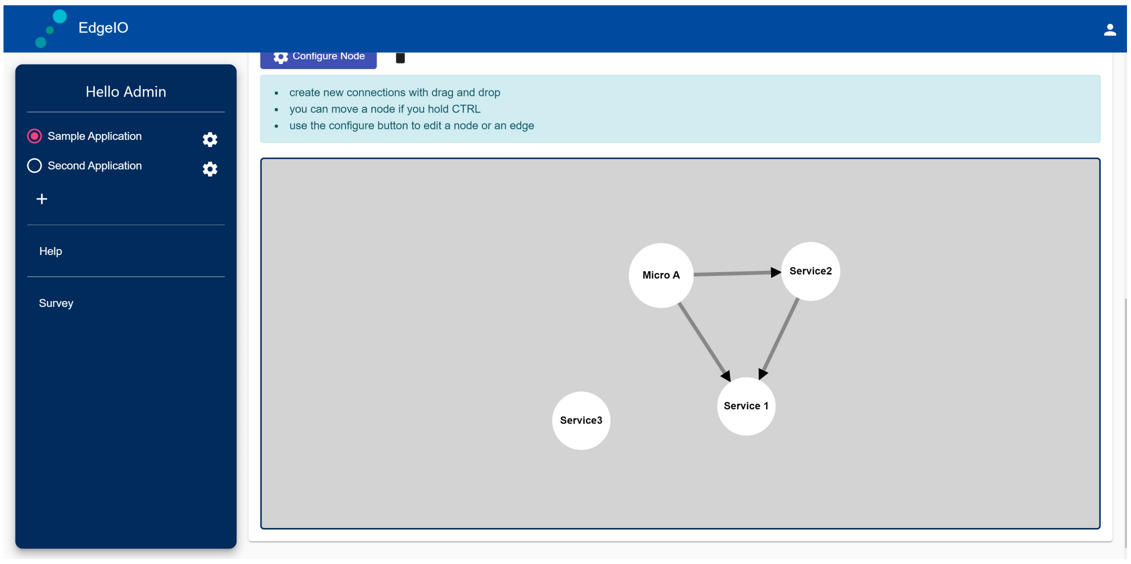
Save Cancel

Figure 3.4: Dashboard form for creating new service.

As specified in the requirements, it should be possible to display the connections between the services graphically. The connection graph shown in fig. 3.5 is used for this purpose. It represents a service as a circle and the connections as links between the services. The arrow symbolizes the target service of a connection. For a connection, the target must comply with all the requirements defined by the start service. A new connection can be created using drag and drop. A dialog opens, where the connection can be configured in more detail.

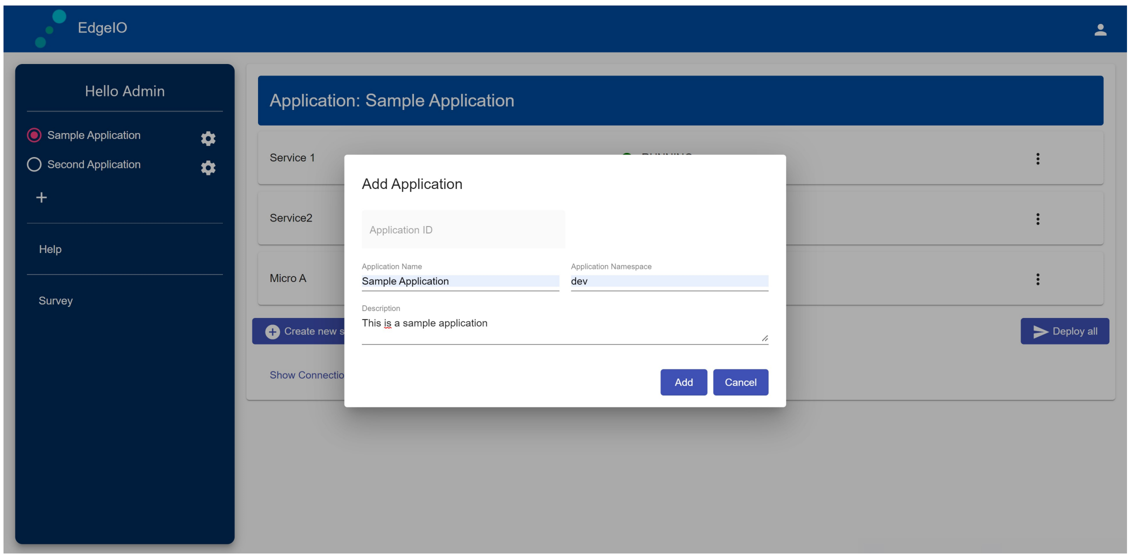
In all situations, where it is possible, only the needed components are displayed, to ensure a clean, user-friendly overview.

For example, the connection graph is only displayed when the user wants to use it. Also, only the most relevant inputs are shown in the configuration form. If more is needed, the form can be expanded. In many situations, dialogs like in fig. 3.6 are used to either visualize data or get user input, so a developer does not have to create a new page, and the current page remains very clean.



- create new connections with drag and drop
- you can move a node if you hold CTRL
- use the configure button to edit a node or an edge

Figure 3.5: Connection graph with different services.



Application: Sample Application

Add Application

Application ID

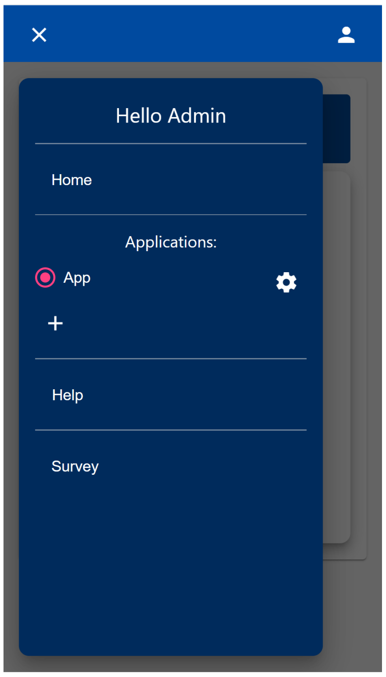
Application Name: Sample Application

Application Namespace: dev

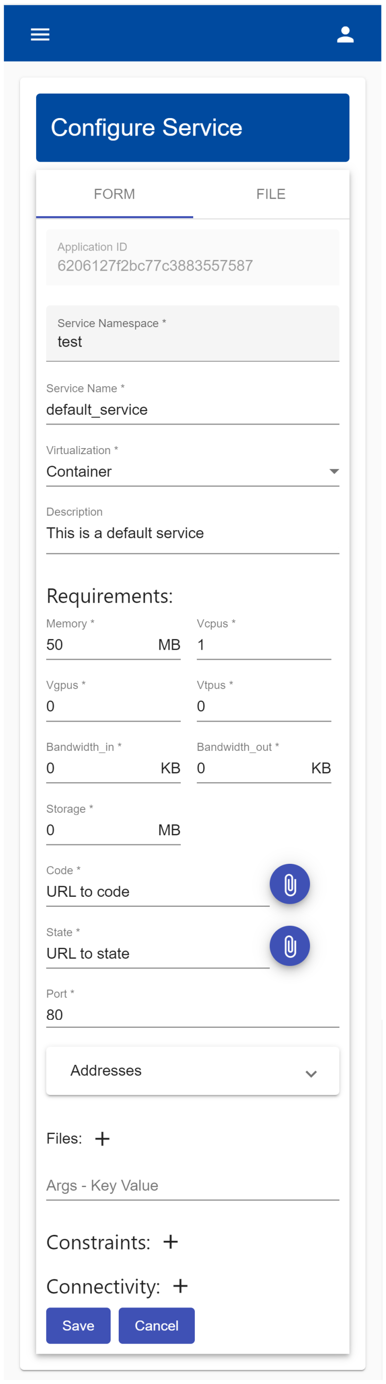
Description: This is a sample application

Figure 3.6: Dialog to add a new application.

Special attention has been paid to support all different screen sizes to ensure that the user always has the best user experience. All screenshots shown here, were made on a laptop, but smaller screens like tablets or smartphone screens are also supported. Suppose the dashboard is used on a smaller screen, only the position of the individual components changes. For example, the side-nav is hidden and displayed only when needed. The side-nav in the mobile version of the dashboard can be seen in fig. 3.7 (a). In fig. 3.7 (b), a responsive view of the service configuration form is shown.



(a) Responsive sidenav.



Application ID
6206127f2bc77c3883557587

Service Namespace *
test

Service Name *
default_service

Virtualization *
Container

Description
This is a default service

Requirements:

Memory *
50 MB

Vcpus *
1

Vgpus *
0

Vtpus *
0

Bandwidth_in *
0 KB

Bandwidth_out *
0 KB

Storage *
0 MB

Code *
URL to code

State *
URL to state

Port *
80

Files: +

Args - Key Value

Constraints: +

Connectivity: +

(b) Responsive configuration view.

Figure 3.7: Responsive views of the dashboard.

4 EdgeIO Frontend

4.1 System Design

In this section, we introduce the EdgeIO frontend design. The different components that belong to it, what each component does and how they communicate with each other. The frontend is based on the already existing EdgeIO framework. Since the framework was not developed with a user interface from the beginning and had no multi-user support, some components had to be modified. We will take a closer look at the modified components. But the parts, where nothing has been changed, we consider here as a black box and therefore do not go into more detail.

4.1.1 Architecture

We now describe the implementation of the EdgeIO user interface. There were several choices: One option would be implementing the GUI directly on the Root Orchestrator, which would have the advantage that the commands can be executed directly without having to be forwarded. However, with this option, the Root Orchestrator must be accessible to all users, which can bring certain security risks, so we decided to create the client separately. This creates a small communication overhead, but the advantages outweigh it. Only the webserver, where the frontend is hosted, has to be accessible for the users. The webserver can communicate with the Root Orchestrator, which is located in a private network. So, the user never has direct access to the orchestrator.

The following figure 4.1 shows the system design of the EdgeIO frontend. It consists of the web client, the CLI client, and the Root Orchestrator. The other components that belong to the EdgeIO framework are not required for the frontend and are therefore omitted for clarity. All components are described in more detail below.

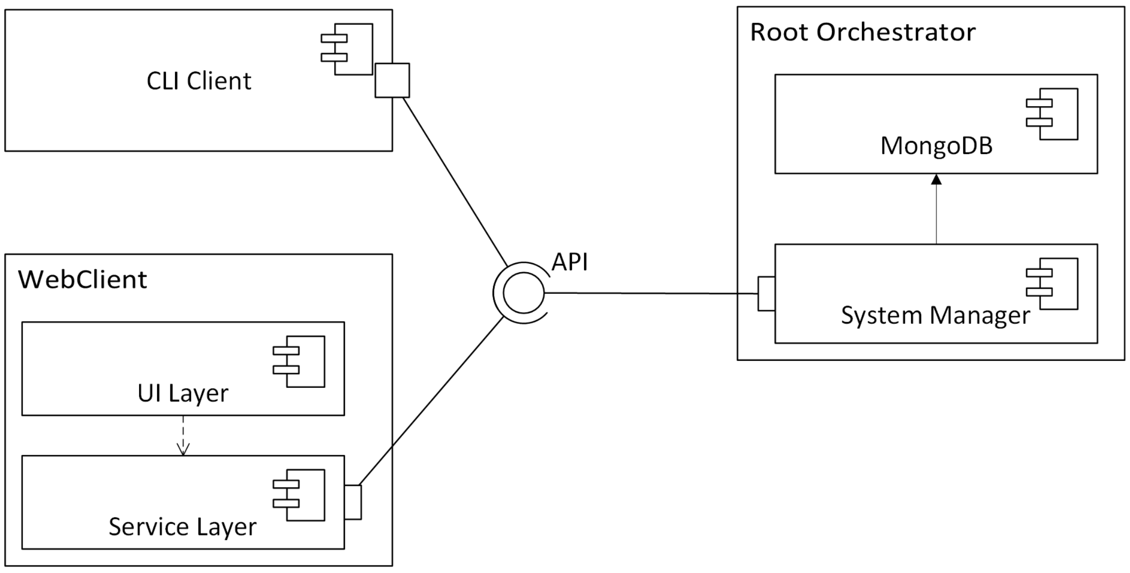


Figure 4.1: System model of the user interfaces.

4.1.2 Web Client

The goal for the web client was to develop an easy-to-use, user-friendly interface. In addition, the frontend should be modular, so it can be easily extended in the future. These requirements can all be fulfilled with the Angular framework used for the frontend as already described in section 3.5.

Now that we have chosen a framework for the development, we can look at the structure of the web client. Figure 4.1 shows the web client. It is divided into two layers: The UI and the service layers. Each layer has a specific task. The UI layer handles the display of the user interface and controls, which component should be displayed. Like any Angular app, our web client has a component-based architecture. This means there is a parent component, and within it are several nested child components. The UI layer and its logic then decide, which component is displayed.

On the other hand, the service layer consists of several services and mainly handles communication between the web client and the System Manager. This includes, for example, the API service and the Auth service. The API service is responsible for sending all API requests correctly with the appropriate headers. The Auth service is responsible for checking if a user has the appropriate role to access the dashboard or specific pages in it.

4.1.3 JSON Web Tokens

Security plays a significant role in the implementation of the user interface. To ensure this, the concept of token-based authentication was introduced. For this purpose, JWT (JSON Web Token) [22] is used in the implementation. JWT is a JSON-based access token and is typically used to identify a user. In addition, JWT is particularly suitable for implementing stateless sessions since all authentication-relevant information can be transferred in the token, and the server does not have to store the sessions.

By default, JWT tokens can be divided into three parts: the header, the payload, and the signature. The header transfers general information about the token. In the payload of the token, data can be entered. A decoded token from the user interface is shown in listing 4.1. This token contains the header in lines 2-5, which transfers general information about the token. The payload is shown in lines 7-16. Among other things, the user name for which this token was issued is specified here. In the end, lines 18-22 contain the signature of the token, which ensures that the token has not been modified. This signature contains a secret key that only the server knows ideally.

```
1  // HEADER:ALGORITHM & TOKEN TYPE
2  {
3    "typ": "JWT", // Type of token
4    "alg": "HS256" // Signature or encryption algorithm
5  }
6  // PAYLOAD: DATA
7  {
8    "fresh": false, // can be used to protect a flask endpoint
9    "iat": 1646415424, // issued at [seconds since UNIX epoch]
10   "jti": "31de4516-3ac5-4855-8b2d-3b5e9c59c57a", -> JWT ID
11   "type": "access", // Type of the token
12   "sub": "Daniel", // subject
13   "nbf": 1646415424, // not valid before [seconds since UNIX epoch]
14   "exp": 1646416024, // expiration time
15   // ... additional data can be added here by the developer
16 }
17 // SIGNATURE
18 HMACSHA256(
19   base64UrlEncode(header) + "." +
20   base64UrlEncode(payload),
21   256-bit-secret)
22 )
```

Listing 4.1: Decoded JWT access token.

There are three different tokens in the EdgeIO framework: The access token, the refresh token, and the reset password token. The access and refresh tokens are very similar, and only the type of the token defined in line 11 and the expiration time in line

14 differ. The access token has a lifetime of 10 minutes and authenticates a user. The refresh token can be used to ensure that the user does not have to reauthenticate every 10 minutes. If the lifetime is expired, it is possible to get a new access token with the refresh token. The refresh token has a lifetime of 7 days. If a user forgets his password, he can call the "forgot password function" and receive a mail with the reset token. This is not a JWT token but cryptographically strong random numbers. This token has a lifetime of 3 hours and can be used to reset the password and reaccess the system.

All these tokens allow us to provide secure authentication for the EdgeIO framework. A user can only see his services, and the services of other users remain hidden. Furthermore, the token always identifies the user who made the request. Based on this, it can be decided if this user has the permissions to execute this request.

4.1.4 CLI Client

All systems considered in section 2.2 offer a command-line control of their system. Some developers prefer this type of control. Thus, a headless version of the user interface was developed to allow users to choose which control they want to use. The tool is called *edgeiocli*. It is a Python-based tool created with the help of the Poetry [23] package. This tool allows the user to use the framework with a traditional command-line tool. *Edgeiocli* is provided via a wheel file and can be installed on every device using python and pip. A user needs an account that the admin can create to use the tool. If a user has an account, he has to login in order to be able to control the framework. For example, a user can log in with the command in listing 4.2 (1). The username, password and IP address of the System Manager must be specified in this command. As soon as the login is successful, the user receives his JWT tokens in response, which are usually stored in the browser's local memory. In the case of *edgeiocli*, however, these tokens are saved in a hidden file and deleted again at logout.

As soon as the user is logged in to the tool, various API requests can be sent to the Root Orchestrator. The access token is added to the header on every request, so it does not matter for the Root Orchestrator if the request was sent from the web client or the CLI tool. A service can be deployed with the command in listing 4.2 (2).

```
(1) edgeiocli login Admin --system-manager-ip 127.0.0.1:10000
    --password Admin

(2) edgeiocli job deploy file.json
```

Listing 4.2: Command with the *edgeiocli* tool.

The difference to the web client is that with this command the service is deployed directly. Likewise, the users must be versed because a service can only be successfully deployed if the JSON file is specified correctly.

Another difference is that currently, only the access token is used in the CLI client. This results in a login session being only valid as long as the validity period of the access token, which in our case is 10 minutes. Therefore, the user has to re-authenticate every 10 minutes. We deliberately decided to do this because otherwise, we would have to extend the validity period of the JWT token, which would make the entire system less secure, or alternatively split up the authentication process between console client and web client and assign separate JWT tokens to each of them. Thus, this way was the most sensible one in our eyes.

The web interface can provide a better user experience, but to still support all the functions in the console, the following commands have been implemented in *edgeiocli*.

- **login** [USERNAME] Logs the user into the system and creates an access token.
- **logout** Logs out the user and deletes the token.
- **change-password** Changes the password of the user.
- **application** [COMMAND]
  - **create** Creates a new application.
  - **delete** [APPLICATION-ID] Deletes an application.
  - **list-jobs** [APPLICATION-ID] Displays all jobs in the application.
  - **list** Displays all applications of the user.
- **job** [COMMAND]
  - **delete** [JOB-ID] Deletes the job.
  - **deploy** [PATH] Tries to deploy the file in EdgeIO.
- **user** [COMMAND]
  - **create** –role [Admin | Application-Provider | Infrastructure-Provider] Creates a new user.
  - **delete** [USERNAME] Deletes the user.
  - **list** Displays all user of the system.
  - **set-roles** [USERNAME] –role [Admin | Application-Provider | Infrastructure-Provider]

After each command, it is possible to use the `--`**help** parameter to call the help page, listing all available commands and any arguments or parameters.

4.1.5 Root Orchestrator

The Root Orchestrator was already part of the existing system of the EdgeIO framework and was described in the background part in section 2.1.1. However, to support the new user interface, the Root Orchestrator was modified in some places. Especially in the System Manager and MongoDB. Only these components are shown in the component diagram in fig. 2.1. However, the Root Orchestrator still consists of all the components described in section 2.1. In the System Manager, the REST API has been significantly expanded, and it now provides all API endpoints needed for the user interface. Also, JWT is now supported, as described earlier. On every startup of the System Manager, a random secret key for the signature of the tokens is created to increase the security. This means if the System Manager is restarted, all users have to log in again because no token is valid anymore.

The database has also been significantly expanded. As there was no multi-user support before, only the services had to be stored in MongoDB, but now the registered users and their attributes have to be stored as well. In addition, it must be possible to assign services to applications and applications to users.

4.2 User Authentication

Since the system now supports multiple users, it must be ensured that only authenticated users have access to the REST API of the System Manager and that only certain users with the corresponding rights can make changes.

The JSON web tokens, explained earlier in section 4.1.3, can be used for this purpose. A JWT is generated during the login process on the System Manager. If the login was successful, the client receives the access and refresh tokens as a response and stores them in the browser's local storage or a hidden file if the *edgeiocli* was used. Before the client sends a new request to the System Manager, he checks if the access token is valid. If that is the case, the bearer token can be added to the request's header. If the token is expired, the refresh token is used to get a new access token. The System Manager also checks if there is a valid token in the header. If there is one, the System Manager knows that the user has already been authenticated and continues to process the request. If not, it prohibits access to the API endpoint. This makes the whole system much more secure as now only users with the appropriate tokens can execute the API endpoints.

4.2.1 User Roles

As already mentioned, there are 3 different user roles:

- Admin
- Application-Provider
- Infrastructure-Provider

Not all users are allowed to make the same API requests to the System Manager. Therefore, the System Manager must know to which role the user belongs. To figure this out, the System Manager looks at the transmitted JWT token in the request, which contains the username, and then checks in the MongoDB if the user has the appropriate role to perform this API request.

4.3 Service Deployment and Installation

The system can be installed in many ways, but the way presented in this work is recommended to take full advantage of the system design. For the web client, a globally accessible webserver is needed. The frontend can be deployed on it using a docker file. In the docker file, the IP address of the Root Orchestrator must be defined so the client can connect to the System Manager. The exact commands that are used to start the user interface are shown in appendix A.3.

We assume that the Root Orchestrator and Cluster Orchestrator have already been installed correctly. In the EdgeIO versions without the user interface, the Root Orchestrator needed to be globally accessible as well, this is no longer necessary in the current version. Here it is sufficient if the frontend and the EdgeIO internal components reach the root; thus the EdgeIO framework could be installed in a non-globally reachable network, and then a VPN connection could be established between the hosted frontend and root. This would further encapsulate the components of EdgeIO and thus increase security.

The *edgeiocli* tool, which allows accessing the framework with the terminal, can be installed simply with a "pip install" and the appropriate wheel file. The host, on which the tool is executed must reach the Root Orchestrator directly. In this case, also a VPN could be used.

5 Evaluation

In this chapter, we evaluate our user interface regarding different aspects. First, we look again at the functional and non-functional requirements from chapter 3, analyze the current status of the implementation, which requirements are already met, and what still needs to be worked on. Afterward, we present the survey, which was made to evaluate different aspects of the dashboard, and in the end, we look at the results of this survey.

5.1 Requirements

To facilitate the evaluation, we introduce two new symbols that indicate which requirements have been met and which still need to be worked on.

- ● **Fulfilled:** This symbol indicates that a certain requirement has been fulfilled.
- ◑ **Not Fulfilled:** This symbol indicates that a particular requirement has not yet been fulfilled and still needs to be worked on.

In the following, the Functional Requirements in table 5.1 and the Non-Functional Requirements in table 5.2 are shown with the corresponding symbols. Now we analyze the different features of the user interface and find out if all requirements have been realized.

FR1 Different roles	◑
FR2 Authentication	●
FR3 Form configuration	●
FR4 Define dependencies between microservices	●
FR5 Headless version	●
FR6 Service management	●
FR7 User management	●

Table 5.1: Status of the functional requirements.

Users must be logged in to the system in the graphical interface and the CLI version to use it (FR2). A distinction is made between the users. There can be application provider, infrastructure provider, and admin users (FR1), although the frontend presented here currently focuses only on the application provider view. Logged-in users can configure a new service with a form (F3) and display already created services and their connections using the connection graph (FR4). Services can be edited, deleted, and monitored. If a service is running and has the appropriate data, CPU and memory usage can be viewed, and a rough location of where this service is deployed (FR6). More detailed data can be viewed in Grafana. If the logged-in user is an admin, he can create new users and edit or delete existing users (FR7). All these functions, except for the visualization of the connections, are also provided with the *edgeiocli* tool (FR5).

NFR1 Easy to use	●
NFR2 Complete	●
NFR3 Robustness	●
NFR4 Response time	◑
NFR5 Responsive	●
NFR6 Extensibility	●

Table 5.2: Status of the non-functional requirements.

Now we take a look at the non-functional requirements. The following survey analyzes the usability of the system (NFR1). Since the created applications and services are stored on the Root Orchestrator, and both the web client and the *edgeiocli* tool only access the root, the current status of the system is always displayed (NFR2). Before a new service can be stored in the root, it is always checked whether the specified configuration is valid. Thus only correct configurations can be deployed, which makes the system much more robust (NFR3). The user interface of the web client has no noticeable delay. Suppose the connection to the server on which the interface is hosted is poor. Some components may be loaded with a minimal delay, but this does not affect the usability in any way (NFR4). However, many scripts are loaded the first time the website is called. Here it would be possible to optimize the performance and load only those that are really used. The graphical user interface constantly adapts to the different screen sizes (NFR5). The modular structure of the client with its components makes the system extensible for new views (NFR6).

5.2 User Survey

This section presents the central part of the evaluation. The goal of the work was to create a robust and user-friendly front-end for the EdgeIO framework. We created a survey, in which 13 people participated. This survey helped us to evaluate the usability and the satisfaction of the user with the developed system and to collect feedback from the users. The feedback could then be used to identify weaknesses of the system, find bugs, and find potential features for the future.

During the survey, the participants interacted with the developed system. It was hosted on a Google Compute Engine with a public IP address so that everyone could access the front end. However, only a webserver for the web client and the Root Orchestrator with which the client interacted were hosted. All other components of the framework were either not needed or simulated.

To provide the same user experience for all participants, they were asked to access the website with a laptop or a desktop PC. All should use the same admin account. In addition, the dashboard was reset to a certain point each time a user logged in. The reset has cleared all configurations from previous participants and ensures that everyone has the same starting situation. If a person logged in to the system, the account was locked for all other participants for 30 minutes to prevent multiple participants from accessing the system simultaneously. Thus, the entire survey could be conducted during this time without being disturbed. However, most have only needed 10 to 15 minutes for the survey.

5.2.1 Structure

Since none of the participants had already used the developed dashboard, at the beginning of the survey, they had to solve five different tasks using the frontend to get familiar with the system and to be able to evaluate it. In addition, the participants had to take a dashboard tour before they started with the tasks. In this tour, the dashboard explains the essential functions to the users, so the participants already have a small overview. We gave the subjects the following tasks. For clarity, the properties that should be used have always been omitted, the exact tasks with all the details are shown in appendix A.1.

Task 1: Create a new application with the following attributes.

Task 2: Have a closer look at the status of the service named "Micro A" in Application "Sample Application". At which location was the service deployed?

Task 3: You are the admin of this system and another user keeps causing problems, so delete the user "Evil-user".

Task 4: Create two new services in your application "MyApplication" with the following properties, you can leave the other attributes unchanged.

Task 5: Display the connection graph in the application "MyApplication". Then create a connection between the two services with the following properties.

After the subjects had completed the tasks, they were asked to fill out the questionnaire. The questionnaire was divided into five different sections. First, the subjects were asked how many tasks they could solve. After that, some background questions were asked about the test person to determine the subjects' level of knowledge and how their feedback should be assessed. For example, *How familiar are you with IT and computer science? (Q3)* or *Have you ever worked with the EdgeIO framework? (Q7)* was asked. Afterward, they were asked, how they liked the different functions of the dashboard, how easy it was to use the system, and finally, the participants were asked how satisfied they were with the graphical user interface and what could be improved.

There were always different answer options for the questions. In most cases, a 5-point Likert scale [24] was used, where 5 points means that the participants completely agree with the statement and 1 that they do not agree at all. In some cases, the participants also had to answer with yes or no or write their answer in a text field. The complete questionnaire with all possible answers is shown in appendix A.2.

5.2.2 Results

As already mentioned, the questionnaire was divided into different parts. We will now take a closer look at all questions and their results.

To find out how many tasks they got right, participants could complete the survey in the dashboard and the number of completed tasks was given to them. Ten participants solved all five tasks correctly, two solved four tasks correctly, and one participant could solve two tasks correctly.

Familiarity Questions

To find out how familiar the participants are already with such frameworks and how the answers should be assessed, the survey participants were asked to fill in the following general questions about themselves:

- **Q2:** What is the highest degree or level of school you have completed?
- **Q3:** How familiar are you with IT and computer science?
- **Q4:** How familiar are you with cloud/edge computing?
- **Q5:** Have you already used systems from AWS, Kubernetes, or something similar?
- **Q6:** If you have the choice between GUI and CLI, which interface do you tend to use?
- **Q7:** Have you ever worked with the EdgeIO framework?

All participants stated that they are very familiar with IT and computer science. One of them was a bachelor's student, two were master's students, nine had already completed the master's degree, and one participant had a doctorate title, all in computer science.

12 of the 13 participants stated that they were very knowledgeable about IT and computer science. They rated the question Q3 with 5 Likert points, the highest possible option. The other participant rated the question with a 4 out of 5 Likert points. Which means that he is also familiar with IT and computer science.

The question *How familiar are you with cloud/edge computing? (Q4)* was answered with an average score of 3.46 points. This means that most participants are already somewhat familiar with the field. Eight participants have also already used systems such as Kubernetes or services from AWS.

When asked whether a participant prefers to use a GUI or the CLI, the answers are relatively balanced but tend minimally towards CLI use.

Nine participants have also already worked with the EdgeIO framework, but they have never used the user interface developed here.

In summary, it can be said that all participants are familiar with computer-science, and most of them have already dealt with the area of edge/cloud computing and can therefore assess the developed system well.

Usability Questions

Now we look at the usability part. Here we try to find out from the questions, how easy it was for the subjects to complete the tasks and get a good overview of their service. If everything is very simple and clear according to the respondents, we can conclude that the usability of the system is high. The following questions were asked:

- **Q8:** How helpful was the tour to understand the frontend?
- **Q9:** Creating new services is straightforward.
- **Q10:** Through the dashboard, I get a good overview of my services.
- **Q11:** I was able to understand the difference between application and service.
- **Q12:** The connection graph illustrates the connections between services very clear.
- **Q13:** The dashboard allows me to express my application requirements accurately.
- **Q14:** What are the strengths of the dashboard, according to you?
- **Q15:** What are the limitations of the dashboard, according to you?

As it can be seen from the graph in fig. 5.1, the usability of the system was rated well. This part received an average score of 4.19. This means that according to the participants the system is useful. The question *Creating new services is straightforward. (Q9)* scored best with an average score of 4.38. The lowest score in this section, with an average score of 3.42, was the question *How helpful was the tour to understand the frontend? (Q8)*. However, it should be noted that this question was added later, and only 7 participants completed it.

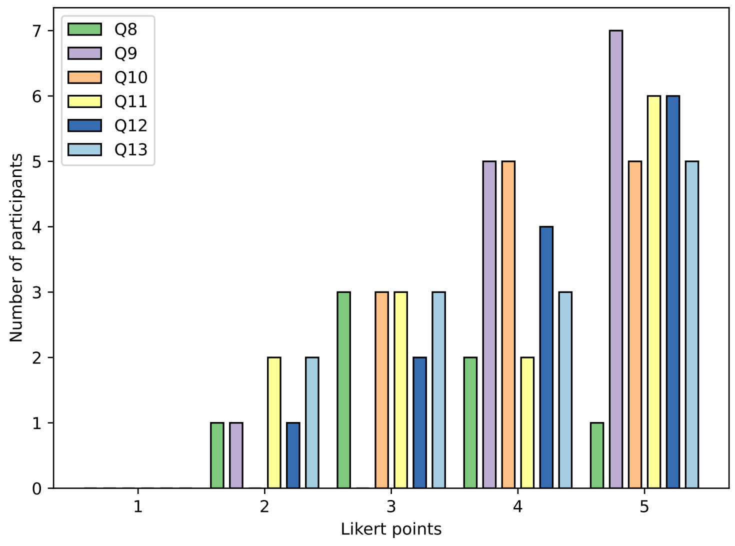


Figure 5.1: Results of the questions Q8 - Q13.

After the usability questions, participants were asked to describe the strengths and weaknesses of the dashboard in open-ended questions. According to the respondents, the dashboard provides a simple and good overview of the services even if the system has been used for the first time. In addition, the graphical view of the connections between the individual services was praised. This feature can simplify complex scenarios.

Now we come to the weaknesses of the system. The most frequently mentioned criticism is that the dashboard should support the user even more. For example, all entries should be provided with descriptions. Some attributes are understandable for users who use the framework more often but can be unclear for new users and need further explanations. Another criticism was that configuring connections across the graph could become problematic as the number of services in an application grows.

Ease of Use Questions

The next section of the questionnaire, the "Ease of Use" part, is presented now. This part tried to determine how easy the dashboard is to use. The participants were asked to answer the following questions:

- **Q16:** It is easy to use.
- **Q17:** It is user-friendly.
- **Q18:** It requires the fewest steps possible to accomplish what I want to do with it.
- **Q19:** I can use it without written instructions.
- **Q20:** I don't notice any inconsistencies as I use it.

- **Q21:** The presented dashboard is better/worse than known cloud platform UI.
- **Q22:** I could differentiate between edge and cloud resources in the platform.

The survey participants also evaluate this question block quite positively. Figure 5.2 shows the results of questions Q16 to Q20. For these questions, there was an average rating of 3.71. The statement *It is easy to use. (Q16)* was rated the highest with an average rating of 4.30. In contrast, the statement *I don't notice any inconsistencies as I use it. (Q20)* was rated the lowest, with an average score of 3.5. This score can certainly be improved in the future. The question Q21, whether the presented dashboard is better or worse than the known UI of other cloud platforms, was rated four times as "better" and nine times as "I have no idea." In addition, question Q22 asked the respondents if they could distinguish between edge and cloud services on the platform, and here, four voted for yes and nine for no.

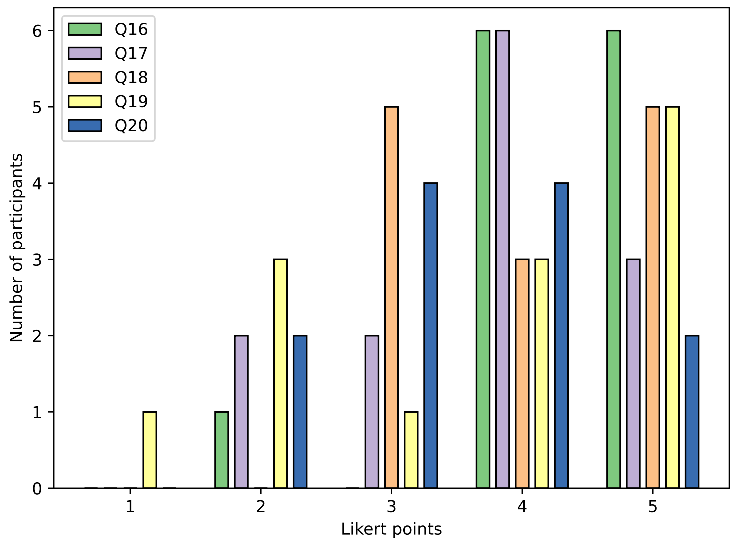


Figure 5.2: Results of the questions Q16 - Q20.

Satisfaction questions

In the last part of the survey, the users were asked, how satisfied they were with the frontend. This question received an average score of 4.07, which means that the participants are satisfied with the system. In addition, the question of whether the dashboard works very similarly to dashboards from other cloud and edge operators was given an average score of 3.38. This can be interpreted as a neutral result. Most participants do not know exactly if the systems are similar or are undecided.

At the end of the survey, two more open questions were asked. First, we wanted to know from the participants how we could improve the dashboard in the future. The following points were mentioned:

- Users should receive more information about their services. For example, how they are currently configured and the service monitoring should be extended.
- Likewise, one respondent suggested creating different templates for services, which would make creating a service even easier for the user.
- The most frequently mentioned point is that the dashboard is already clear, but some things could be made even more evident.
- The last noticeable point that came up in these responses was that the dashboard does not provide the same user experience in all browsers. In particular, the Safari browser on Apple devices sometimes causes problems. On the other hand, the Chrome browser did not cause any problems and delivered the expected experience.

In the second question, the test persons were asked for additional comments on the dashboard. Here we were mainly shown minor graphical errors that could be improved. For example, a button is not displayed correctly, or not all input elements are consistent.

In addition, the feedback suggested that the configuration of a service could be made a little easier. In some cases, the respondents did not know how to enter the data, why they should enter it, and in which data format the inputs should be specified. This brings us back to a previously mentioned point, the additional hints at the different input fields. The last point highlighted from these answers is the suggestions for future features, like a view for the Infrastructure Manager and the Cluster Manager and an improvement of the Service Monitoring.

6 Conclusion and Future Work

The initial goal was to design and build a robust and user-friendly user interface for the EdgeIO orchestration framework. The system should also support multi-user operations and consider various security aspects. We designed and built the user interface for the EdgeIO framework. This user interface was then hosted on a Google Compute Engine and evaluated by several experienced developers in this area through a survey.

First, let's consider the research questions presented at the beginning. Later, we will summarize the survey and look at some specific elements that we can improve in the future.

6.1 Conclusion

RQ1: How can a large number of edge devices be controlled with a simple UI in a distributed deployment infrastructure? First, we looked at other orchestration systems in section 2.2 and how they allow a user to control his services. All considered systems offer a command-line tool and API endpoints for control. In addition, some systems offer a graphical user interface. A GUI and a CLI have been created in this work that allows controlling such systems. The architecture and implementation of the developed systems are explained in more detail in chapter 4.

RQ2: How should UI be designed in edge frameworks to be developer-friendly? A user interface needs a very simple and clear design to be developer friendly. The design of the developed web client was presented in section 3.5. It was tried to stick to the most common design standards to get the best possible overview. In addition, a survey was conducted in section 5.2 to evaluate user-friendliness. The results of the survey showed that the developed system offers good usability, but it can be improved in the future.

RQ3: What must be considered that despite multi-user use the framework remains secure? For multi-user access, we have introduced several roles with different permissions. Section 4.2 describes how we ensured secure authentication. The JSON web tokens described in detail in section 4.1.3 were also used for this purpose. The Root Orchestrator was also modified as described in section 4.1.5, so only authorized users have access to certain resources.

In summary, we have laid the foundation of the user interface for the EdgeIO framework with this work. However, the project is certainly not finished yet. Soon, the user interface will be integrated into the framework. This provides the users with a much better user experience. Since our implementation uses Angular, it is modular and easily extensible, allowing future projects to improve the system constantly.

The web client could be stabilized in the future, and additional views for other roles like the infrastructure provider could be implemented. In addition, the *edgeiocli* tool should be further improved, and security aspects could still be enhanced.

6.2 Survey

The survey was very useful and gave us a lot of insights. We could see that the dashboard is on the right track and provides a good overview of the services and applications, making working with the EdgeIO framework a lot easier. The survey was very well rated, but some problems and suggestions for improvement were pointed out. Through the participants' feedback, it became clear that there are still some graphical inconsistencies in the current version of the web client. Some have already been fixed, but the dashboard should be constantly developed further to meet modern standards and provide a good user experience.

In addition, the monitoring of services could be improved by providing more data or integrating elements from Grafana for visualization. An important point is that the dashboard should support the user even more by adding additional descriptions to all inputs.

6.3 Admin and Infrastructure Provider View

As already mentioned, due to time constraints, we focused on the application provider view in this work. However, to provide a helpful user interface for the other users of the framework, the admin view should be extended. Moreover, the infrastructure provider view should be added in the future. The admin should be able to see all services of each user and manage them if necessary.

The infrastructure provider should be able to connect several workers or even an entire cluster to the EdgeIO framework using the graphical user interface. It should also be possible to manage all integrated devices and define specific SLAs for them. For example, certain devices should respect a load limit.

The development of these views has the same challenges as the application provider view. The new views should provide a good overview, facilitate the system's usability, and take security aspects into account.

6.4 CLI Client

The CLI client already offers some functions, but it could also be further extended. First and foremost, it should be avoided that the user has to log in every 10 minutes again to use the system. This can be realized as already described in section 4.1.4 by using separate tokens with a high expiration time for the client or by increasing the total token expiration time, negatively affecting the web client. Both solutions are not satisfactory, so in the future, a way should be found to find out if the session is still needed and, if so, to refresh the access token with the refresh token.

Another point that should be further developed is the monitoring of various services. Currently, only the CPU and memory data are displayed if available.

6.5 Security

JSON Web Tokens are used in the developed user interface, as mentioned several times. These secure the individual API endpoints and ensure that only authorized users can execute the corresponding functions. The use of such authentications is beneficial, but at the moment, it cannot exploit its advantages, since all data is currently transferred via HTTP. Therefore, any potential attacker could intercept the transmitted token and make API requests themselves. Therefore, this should be changed to an HTTPS connection soon. Also, the connection between the webserver hosting the frontend and the Root Orchestration should be changed to an HTTPS connection. Furthermore, the internal connections of the EdgeIO framework could be secured via HTTPS connections or a TLS layer.

The Root Orchestrator and the webserver can be installed on the same host. However, if they are installed on separate hosts, a VPN connection can be established between them so only the webserver needs to be publicly accessible. The Root Orchestrator is then protected from external requests.

Looking at the other systems, it has become increasingly apparent that many modern user interfaces do not use a password for authentication but PATs (Personal Access Tokens). These are long non-human readable strings and have advantages over a regular password. They offer better performance because nothing has to be encrypted/unencrypted. Furthermore, they are unique, revocable, and more secure because brute force attacks cannot harm them. Another advantage is that multiple personal access tokens can be generated per user, but passwords and usernames can be created only once. The various advantages make it worthwhile to look at this authentication possibility and possibly include it in future versions of the EdgeIO user interface.

A Appendix

This chapter contains the tasks that were given to the survey participants, as well as the detailed questionnaire and the different answer options. At the end the folder structure is presented and which commands are needed for the start for the start of the system.

A.1 Tasks

1. Create a new application with the following attributes:
   - Name: MyApplication
   - Namespace: survey
   - Description: survey task
2. Have a closer look at the status of the service named "Micro A" in Application "Sample Application". At which location was the service deployed?
   - The possibility to select from a dropdown menu one of the following cities: Rom, London, Munich, New York
3. You are the admin of this system and another user keeps causing problems, so delete the user "Evil-user".
4. Create two new services in your application "MyApplication" with the following properties, you can leave the other attributes unchanged.
   - Service 1
     - Name: Service 1
     - Namespace: dev
     - Memory: 100
   - Service 2
     - Name: Service 2
     - Vcpu: 2

5. Display the connection graph in the application "MyApplication". Then create a connection between the two services with the following properties:
   - Type: latency
   - Threshold: 300 ms
   - Rigidness: 50
   - Convergence Time: 250 sec

A.2 Questionnaire

General

Q1: How many tasks did you solve correctly?

Background Questions

Q2: What is the highest degree or level of school you have completed?

Q3: How familiar are you with IT and computer science?

Q4: How familiar are you with cloud/edge computing?

Q5: Have you already used systems from AWS, Kubernetes, or something similar?

Q6: If you have the choice between GUI and CLI, which interface do you tend to use?

Q7: Have you ever worked with the EdgeIO framework?

Usability

Q8: How helpful was the tour to understand the frontend?

Q9: Creating new services is straightforward.

Q10: Through the dashboard, I get a good overview of my services.

Q11: I was able to understand the difference between application and service.

Q12: The connection graph illustrates the connections between services very clear.

Q13: The dashboard allows me to express my application requirements accurately.

Q14: What are the strengths of the dashboard, according to you?

Q15: What are the limitations of the dashboard, according to you?

Ease of Use

Q16: It is easy to use.

Q17: It is user-friendly.

Q18: It requires the fewest steps possible to accomplish what I want to do with it.

Q19: I can use it without written instructions.

Q20: I don't notice any inconsistencies as I use it.

Q21: The presented dashboard is better/worse than known cloud platform UI.

Q22: I could differentiate between edge and cloud resources in the platform.

Satisfaction

Q23: I am satisfied with it.

Q24: It works very similar to comparable cloud user interfaces.

Q25: How do you think we could improve the user-interface?

Q26: Do you have any additional comments or feedback for us?

Questionnaire answer options

This section now lists the various response options of the questionnaire.

- In the question Q1 you could choose a number between 1 and 5.
- For question Q2 you could choose the following options.
  - High school
  - Bachelor´s degree
  - Master´s degree
  - Doctorate degree
  - other
- Questions Q5, Q7 and Q22 were yes or no questions.
- The question Q21 you could choose between better, worse, or I have no idea.
- For questions Q14, Q15, Q25 and Q26, participants should enter the answer in a text field.
- For all other questions the participants had to answer with a 5 point Likert scale where 1 was disagreement and 5 agreement.

A.3 Directory Structure

This section presents the directory structure of the user interfaces and describes which steps are needed to start up the entire systems. It describes how to start the web client and the command line tool, but assumes that the EdgeIO framework has already been started.

```
EdgeIO-Frontend/
├── dist
├── docker
│   ├── angular-environment
│   │   └── main.js
│   ├── entrypoint.sh
│   └── nginx.conf
├── docker-compose.yml
├── Dockerfile
└── src
```

Figure A.1: Directory structure of GUI software components.

The tree structure in fig. A.1 shows the different files needed for the web client. The src folder contains all the source code. However, this is not illustrated here. If Angular is installed on the system, the *ng serve* command can be used to create a local development server. The client is by default available on http://localhost:4200. However, this should only be used for development.

For the production the dist folder is needed, the *ng build* command can be used to build the web client and create the dist folder. The src folder is not used in the production environment.

The docker-compose.yml file starts the frontend. In this file, the environment variable API-ADDRESS can be set. The IP address of the System Manager should be specified here.

The docker-compose.yml file executes the Docker file. This, in turn, uses the files in the docker folder. The nginx.conf file is used to configure the server, and the entrypoint.sh and the main.js script are needed to use the previously set environment variables in the web client. The following command can be used to start the graphical user interface.

- docker-compose.yml up –build -d

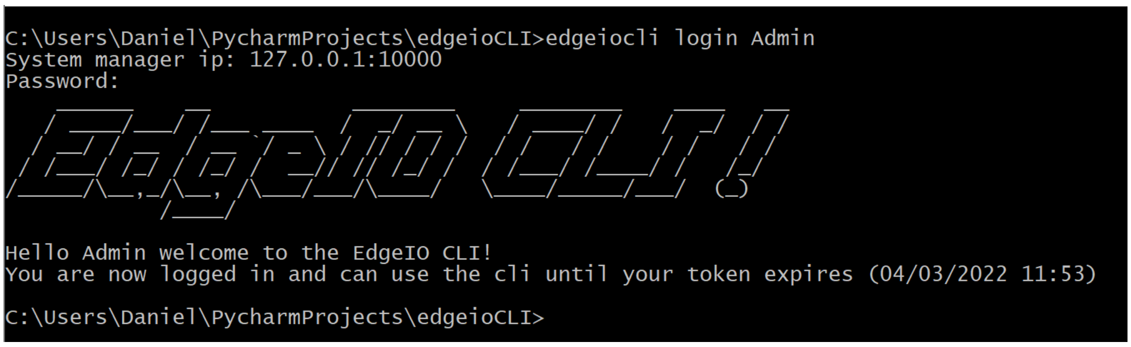
```
CLI/
├── dist
│   ├── edgeiocli.tar.gz
│   └── edgeiocli.whl
└── edgeiocli
```

Figure A.2: Directory structure of CLI software components.

The CLI version consists of fewer components. The exact structure is shown in fig. A.2. The source code is located in the edgeiocli folder. With the command *poetry build*, the dist folder can be created. This folder contains the edgeiocli.tar.gz and the edgeiocli.whl file. The tool can then be installed with the following command:

- pip install edgeiocli.whl

Now the edgeiocli command should be available, and a user can log in to the system as shown in fig. A.3 and use all commands.



```
C:\Users\Daniel\PycharmProjects\edgeioCLI>edgeiocli login Admin
System manager ip: 127.0.0.1:10000
Password:

Hello Admin welcome to the EdgeIO CLI!
You are now logged in and can use the cli until your token expires (04/03/2022 11:53)

C:\Users\Daniel\PycharmProjects\edgeioCLI>
```

Figure A.3: Login with the CLI in the terminal.

List of Listings

List of Figures

List of Tables

Bibliography

[1] C. Gong, J. Liu, Q. Zhang, H. Chen, and Z. Gong. “The Characteristics of Cloud Computing.” In: *2010 39th International Conference on Parallel Processing workshops (ICPPW 2010)*. Ed. by W.-C. Lee. Piscataway, NJ: IEEE, 2010, pp. 275–279. ISBN: 978-1-4244-7918-4. DOI: 10.1109/ICPPW.2010.45.

[2] GitHub. *EdgeIO*. 27.02.2022. URL: https://github.com/edgeIO.

[3] Giovanni Bartolomeo. “Enabling Microservice Interactions within Heterogeneous Edge Infrastructures.” Master's Thesis. TUM, 15.09.2021.

[4] S. A. Baset. “Cloud SLAs.” In: *ACM SIGOPS Operating Systems Review* 46.2 (2012), pp. 57–66. ISSN: 0163-5980. DOI: 10.1145/2331576.2331586.

[5] Amazon Web Services, Inc. *Intelligenz bei IoT Edge — AWS IoT Greengrass — Amazon Web Services*. 5.02.2022. URL: https://aws.amazon.com/de/greengrass/.

[6] A. Kurniawan. *Learning AWS IoT: Effectively manage connected devices on the AWS cloud using services such as AWS Greengrass, AWS button, predictive analytics and machine learning*. 1st ed. Birmingham: Packt Publishing Limited, 2018. ISBN: 9781788394666.

[7] *Kubernetes*. 17.02.2022. URL: https://kubernetes.io/.

[8] G. Sayfan. *Mastering Kubernetes: Automating container deployment and management*. Birmingham, UK: Packt Publishing, 2017. ISBN: 9781786469854.

[9] Kubernetes. *Deploy and Access the Kubernetes Dashboard*. 15.02.2022. URL: https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/.

[10] *Edge X Foundry*. 17.02.2022. URL: https://www.edgexfoundry.org/.

[11] M. Johanson. *Introduction - EdgeX Foundry Documentation*. 14.02.2022. URL: https://docs.edgexfoundry.org/2.1/.

[12] *K3s: Lightweight Kubernetes*. 12.10.2021. URL: https://k3s.io/.

[13] KubeEdge. *KubeEdge*. 15.10.2017. URL: https://kubeedge.io/en/.

[14] *Eclipse ioFog*. 22.02.2021. URL: https://iofog.org/docs/2/getting-started/whats-new.html.

[15] *Eclipse fog05 - The End-to-End Compute, Storage and Networking Virtualisation solution*. 11.05.2021. URL: https://fog05.io/.

[16] Grafana Labs. *Grafana: The open observability platform*. 26.02.2022. URL: https://grafana.com/.

[17] Malan, Ruth and Bredemeyer, Dana and others. *Functional requirements and use cases*. Bredemeyer Consulting, 2001. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.436.4773&rep=rep1&type=pdf`.

[18] L. Chung. *Non-Functional Requirements in Software Engineering*. 1st ed. Boston: Springer, 2000. ISBN: 9781461552697.

[19] H. Hick, M. Bajzek, and C. Faustmann. "Definition of a system model for model-based development." In: *SN Applied Sciences* 1.9 (2019). ISSN: 2523-3963. DOI: `10.1007/s42452-019-1069-0`.

[20] *Angular*. 11.02.2022. URL: `https://angular.io/`.

[21] A. C. Team. *Angular Material*. 17.02.2022. URL: `https://material.angular.io/`.

[22] Auth0.com. *JWT.IO*. 21.02.2022. URL: `https://jwt.io/`.

[23] *Poetry - Python dependency management and packaging made easy*. 21.02.2022. URL: `https://python-poetry.org/`.

[24] R. Likert. "A technique for the measurement of attitudes." In: *Archives of psychology* (1932).