



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# **Designing Interaction Framework for Multi-Admin Edge Infrastructures**

**Maria Nieves Viñals**





DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# **Designing Interaction Framework for Multi-Admin Edge Infrastructures**

## **Entwurf eines Interaktionsrahmens für Multi-Admin-Edge-Infrastrukturen**

Author:	Maria Nievas Viñals
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	Dr. Nitinder Mohan, Giovanni Bartolomeo
Submission Date:	September 15th, 2022



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

Munich, September 15th, 2022

Maria Nieves Viñals

## Acknowledgments

I would like to thank Prof. Dr-Ing. Jörg Ott for giving me the opportunity to work on this project. Also, I would like to thank Dr. Nitinder Mohan and Giovanni Bartolomeo for all the guidance and learning that they provided me over the past few months. I am very appreciative of the assistance and support you gave me throughout the process. Finally, I want to thank my friends and family who inspire me every day and support me in every endeavor I pursue and especially Silvia, who has stood by me throughout this project and has given me unending support.

# Abstract

As advanced technologies emerge, improved performance and latency are in demand. Edge computing, a branch of cloud technologies, brings data processing closer to the end-user, rather than sending data to be processed in central data servers, away from the device network. Recently, the need to develop straightforward, easy-to-go platforms arises, in order to have a better experience deploying services and controlling edge infrastructures. This is not an easy task, since such environments introduce several constraints, and open technological challenges are currently being faced.

Oakestra is a lightweight framework with a unique architectural implementation that works on the edge. Our goal is to provide a recipe for developers to manage their resources in this complex edge environment. We designed the interactions as part of a three-step pairing process for setting up and monitoring clusters. The cluster handshake procedure consists of (i) registering basic information of the cluster and getting a security key as a result, (ii) attaching the cluster providing the previously received key, and (iii) monitoring the clusters attached within the framework. With this last step, the cluster is given a new identifier to be used within the orchestration framework for access to a single management interface and automatic service reconfiguration. In addition, the design is implemented in the Oakestra framework, improving the graphical user interface to facilitate the entire process. Future work will integrate more cluster management features into the system, assisted by an improved user interface.

# Kurzfassung

Mit dem Aufkommen fortschrittlicher Technologien sind verbesserte Leistung und Latenzzeiten gefragt. Edge Computing, ein Zweig der Cloud-Technologien, bringt die Datenverarbeitung näher an den Endnutzer heran, anstatt die Daten zur Verarbeitung an zentrale Datenserver zu senden, die vom Gerätnetz entfernt sind. In jüngster Zeit hat sich die Notwendigkeit ergeben, unkomplizierte, leicht zu bedienende Plattformen zu entwickeln, um die Bereitstellung von Diensten und die Kontrolle von Edge-Infrastrukturen zu erleichtern. Dies ist keine leichte Aufgabe, da solche Umgebungen mehrere Einschränkungen mit sich bringen und technologische Herausforderungen zu bewältigen sind.

Oakestra ist ein leichtgewichtiges Framework mit einer einzigartigen architektonischen Implementierung, das auf einer Edge-Umgebung läuft. Unser Ziel ist es, den Entwicklern ein Rezept für die Verwaltung ihrer Ressourcen in dieser komplexen Edge-Umgebung an die Hand zu geben. Wir haben die Interaktionen als Teil eines dreistufigen Kopplungsprozesses für die Einrichtung und Überwachung von Clustern konzipiert. Das Cluster Handshake-Verfahren besteht aus (i) der Registrierung grundlegender Informationen des Clusters und dem Erhalt eines Sicherheitsschlüssels als Ergebnis, (ii) dem Anschließen des Clusters unter Verwendung des zuvor erhaltenen Schlüssels und (iii) der Überwachung der angeschlossenen Cluster innerhalb des Frameworks. Mit diesem letzten Schritt erhält der Cluster eine neue Kennung, die innerhalb des Orchestrierungsrahmens für den Zugang zu einer einzigen Verwaltungsschnittstelle und die automatische Neukonfiguration von Diensten verwendet wird. Darüber hinaus wird der Entwurf in das Oakestra-Framework integriert und die grafische Benutzeroberfläche verbessert, um den gesamten Prozess zu erleichtern. Zukünftige Arbeiten werden weitere Funktionen für die Clusterverwaltung in das System integrieren, unterstützt durch eine verbesserte Benutzeroberfläche.

# Contents

<b>Acknowledgments</b>	<b>iii</b>
<b>Abstract</b>	<b>iv</b>
<b>Kurzfassung</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Contribution . . . . .	1
1.2 Overview of Content . . . . .	2
<b>2 Background</b>	<b>3</b>
2.1 Multi-Access Edge Computing . . . . .	4
2.1.1 Main Principles and Challenges . . . . .	4
2.1.2 Strategy and Proposed Solutions . . . . .	5
2.1.3 Orchestration Frameworks and Standards . . . . .	6
2.2 Oakestra Structure Framework . . . . .	9
2.2.1 Root Orchestrator . . . . .	9
2.2.2 Cluster Orchestrator . . . . .	10
2.2.3 Worker nodes . . . . .	10
2.2.4 User interface . . . . .	10
2.2.5 Project Structure and Service Deployment . . . . .	11
<b>3 System Design</b>	<b>13</b>
3.1 Requirements . . . . .	13
3.2 System Modeling and Integration . . . . .	14
3.2.1 Register cluster . . . . .	15
3.2.2 Attach cluster . . . . .	16
3.2.3 Monitor clusters . . . . .	18
<b>4 Implementation and Evaluation</b>	<b>19</b>
4.1 Overview of the technologies used . . . . .	19
4.1.1 REST API . . . . .	19
4.1.2 Flask and Flask-RESTful . . . . .	20
4.1.3 JSON Web Tokens . . . . .	21
4.2 System Implementation . . . . .	23
4.2.1 Working with Flask Blueprints . . . . .	23
4.2.2 Cluster registration in the system . . . . .	25

4.2.3	JSON Web Tokens Implementation . . . . .	25
4.2.4	Communication between Cluster Manager and System Manager . . . .	30
4.2.5	Architecture . . . . .	33
4.2.6	Frontend . . . . .	34
4.3	Evaluation . . . . .	40
4.3.1	Static analysis evaluation . . . . .	40
4.3.2	Security analysis of JSON Web Tokens . . . . .	41
<b>5</b>	<b>Conclusion and Future Work</b>	<b>43</b>
5.1	Future Work . . . . .	43
	<b>List of Figures</b>	<b>45</b>
	<b>Bibliography</b>	<b>46</b>



# 1 Introduction

Edge computing is revolutionizing IoT. Edge computing is a branch of cloud services that is remarkably successful at providing cost-effective deployments and low-latency experiences closer to users. In particular, a combined edge and cloud architecture enables time-sensitive data to be processed efficiently by being processed in advance, with no need to call a more powerful and centralized server. Due to this, high-quality results are available when it comes to applications that require low-latency processing, such as video streaming and augmented reality, whose services strive for real-time accessibility, improving user experience. The rapid development of edge applications leads to a high demand for more development in edge infrastructures and better resource management from infrastructure providers. Thus, edge infrastructure providers play an increasingly important role in today's orchestration frameworks for managing distributed edge clusters at the local network. An edge node, also called a gateway node, can be comprised of multiple edge servers and has specific computing and storage capabilities [1].

Decentralized applications run on ubiquitously connected heterogeneous devices at the edge of the network, creating challenges in their deployment and management. As a result, it is important to have a clear and defined method for connecting and monitoring those devices, taking into account the varying network capabilities and abstracting their difficulties. The purpose of this thesis is to propose a generic mechanism for facilitating device connectivity in edge environments. Particularly, we explore an efficient procedure for infrastructure providers to set up their resources in a multi-cluster infrastructure. Research is conducted on principles and challenges as well as different strategies and current frameworks that try to mitigate the edge constraints. We continue to design our system to manage edge nodes in a multi-cluster federated environment, using a secure approach. As part of the design, we want to register the cluster in the framework, attach it, and monitor it from intuitive user interfaces, including cluster status updates at all times. Additionally, the cluster pairing process and management are secured using reliable methods to guarantee the right authorization is only granted to the appropriate infrastructure providers and that no external parties are allowed to access it.

## 1.1 Contribution

To overcome the challenges mentioned above, we have designed a cluster pairing mechanism in a multi-cluster federated environment for lightweight orchestrations. The procedure is applied to the Oakestra framework [2]. Our approach takes care of setting up clusters and managing them in our infrastructure in a convenient and secure way. New features have also

been added to the Dashboard user interface of Oakestra to support this design.

## **1.2 Overview of Content**

Before diving into the details of how the above-outlined process would be implemented, we introduce the theoretical background of the project in chapter 2. The study starts with some research on the principles and challenges of mobile edge computing. It is also important to examine current solutions and existing frameworks for application development in cloud-edge environments where multiple service providers share resources, and what constraints they face. Oakestra's original architecture is presented, followed by a review of the components, and how the main deployment of the control plane of Oakestra is conducted. In chapter 3 we discuss a generic pairing process of a cluster into a federated multi-cloud framework. Our first step is to examine the requirements, and then we describe the design details, raising a three-step process: registering, attaching, and monitoring clusters. Finally, chapter 4 provides the technical details of the proposed prototype for the Oakestra framework. Moreover, we present a security evaluation of tokens, used for the handshake procedure. We conclude chapter 5 by presenting a summary of what has been achieved and what remains to be completed in the future.

## 2 Background

Multi-cloud is evolving towards the edge. With more data being generated closer to end-users, edge locations represent the next frontier for innovation, and container-based application packaging is the logical approach for organizations to manage and orchestrate those edge applications [3]. Containers can run from various cloud service providers, optimizing infrastructure costs by using resources smartly. This suggests that the growth that multi-cloud architectures have been experiencing for the past decade derives significantly from economic interests.

The multi-cloud architecture is based on both a common public cloud and a cost-effective private cloud, offering users flexibility to choose privacy levels most appropriate to their particular project [4]. In its most basic form, a cluster consists of two or more computers, or nodes, that run in parallel to accomplish a common task. Workloads consisting of many parallelizable tasks can be distributed easily among the cluster nodes. As a result, these tasks can take advantage of the combined memory and processing power of each computer to increase performance. The benefits of cluster computing include its high availability, load balancing, scaling, and performance enhancements. We can define clusters in cloud computing as a group of virtual machines connected within a virtual cloud. In addition, cluster resilience and user latency can be enhanced by deploying nodes across multiple availability regions [5].

Edge infrastructure providers are responsible for providing services and managing resources at the edge of the network in close proximity to end-users. They monitor the network's distributed edge nodes. Clusters can be formed from these nodes. Multi-cluster management refers to a single management plan that enables us to gain visibility and control across all of our clusters (for instance through a GUI). In recent years, it has become clear that managing multiple clusters, while using cloud-native tools, is an essential part of cloud computing [6]. Among the capabilities offered is the ability to login and authenticate centrally and to monitor, defend and scale security across clusters.

Optimal performance between different cloud environments requires the development of an interconnection layer. Cloud Federation (also known as Federated Cloud) is a mechanism for managing applications and services across multiple clouds. It integrates private, community, and public clouds into a scalable computing platform. It needs two or more service providers to be connected to the cloud computing environment to load balance traffic and handle spikes in demand [7]. As we progress, this concept is used as an interesting strategy to address the challenges associated with multi-access edge computing, which provides decentralized computing and application management capabilities. In the following sections, we present an overview of multi-access edge computing. The main principles and challenges are highlighted and some current strategies and state-of-the-art technologies are explored. Furthermore, the

architecture and components of Oakestra are described.

## 2.1 Multi-Access Edge Computing

Mobile edge computing, also known as multi-access edge computing (MEC) is an edge computing use case for service providers. It extends cloud computing evolution using mobility, cloud services, and edge computing to move application hosts away from a centralized data center to the edge of the network. Increasingly, service providers are moving workloads and services to the network's edge, thereby reducing latency and increasing the throughput in their applications [8].

The purpose of this section is first to review edge requirements and challenges, discussing cluster management's suitability for facilitating the deployment of distributed multi-operator applications across edge clouds. We then explore some proposed strategies to address the challenges listed. Lastly, we investigate the contributions of existing frameworks and standards such as Federation v2, ClusterAPI, Kubernetes Web View, EdgeNet, and authorization standards like OAuth and OpenId.

### 2.1.1 Main Principles and Challenges

In a cloud-edge infrastructure, managing multiple clusters with multiple users, organizations, and operators enables them to share infrastructure resources. Operators can also separate workloads at the cluster level (sensitive vs nonsensitive data processing, production vs. nonproduction). There are important principles we must consider in cloud-edge environments:

- Container-based application monitoring: Our devices and applications we want to deploy are packaged using lightweight containers. These containers can be distributed among services and applications to the edge. In addition, users may choose whether to deploy edge capability components and customize their configurations [3]. Every container connected to the network should be monitored and analyzed continuously.
- Clusters connectivity. Edge clusters need to be built in a simple and flexible way, so we can add edge nodes within them. There should be zero learning costs associated with attaching these new clusters, following consistency, and ensuring high availability to operate continually. Load balancing or fail-over issues should be addressed.
- Security dimensions. Security issues in the cloud can refer to safety mechanisms, server monitoring, data confidentiality, or avoiding malicious operations. A multi-cloud infrastructure must ensure that internal operations are not shared with the outside world. The confidentiality of data is another important consideration in a federated setting; no organization should transfer data to the cloud until there is a level of trust established between the cloud providers and the consumers [9] [10].

It is important to consider that edge computing environments can lack resource capacity, especially in the context of large user bases. As well, each edge infrastructure provider manages and uses its resources and maintains the edge infrastructure, which is a challenging task to do. Network capabilities can vary from each computing node, as they can run on different architectures [11]. Moreover, enabling deployments in many geographical locations requires more edge nodes, raising costs.

Lastly, another challenge arises in the multi-operators scenario: the cooperation of infrastructure providers. Since individual IPs only know a limited amount about the whole edge computing environment, it is difficult to optimize for efficient delivery of a variety of services. In addition, the collaboration between providers makes it easier to detect attacks and respond to them [12].

### **2.1.2 Strategy and Proposed Solutions**

In order to look into the strategies, we should become familiar with Kubernetes, as it is the most popular container orchestration framework. Kubernetes [13], also known as K8, is an open-source container orchestration engine that automates the deployment, scaling, and management of containerized applications. There are three basic units in the Kubernetes system: pods, clusters, and the control plane. Pods are groups of containers sharing storage and network resources, distributed across multiple nodes. Clusters consist of worker nodes that run the pods, and control plane nodes that manage workers and deployed pods. The control plane has a global view of the clusters' activities, and it can make high-level decisions about the state of worker nodes and deployment. To communicate with the control plane and orchestrate pods, REST APIs are used. In general, Kubernetes multi-clustering approaches can be divided into two categories, the control plane-centric and the network-centric [14]:

- Kubernetes-centric, monitoring multiple clusters from a centralized management plane for multiple clusters. The resources are then distributed across clusters.
- Network-centric, enabling the communication directly between the applications that reside in clusters. Network configuration is used to implement resource replicating and maintain consistency across clusters.

Although Kubernetes is a great orchestration platform for the cloud, it has some limitations to the edge [11]. Due to its cloud-based nature, Kubernetes does not provide offline support (if the control plane fails, so do the clusters). Additionally, it uses a single clustered approach and requires extensions to be deployed in federated environments.

In order to address the challenges outlined above, it is important to develop compound multi-edge orchestrations that overcome the services' constraints. The best way to compose solutions from multiple clouds is unclear. A common issue is that current solutions up to date assume unique single-cloud frameworks. We now take a deeper look into interesting proposed strategies.

In a scenario where both edge and cloud co-exist, a resource provisioning model is proposed, called Edge federation [1]. Multi-edge infrastructure providers (EIP) and clouds collaborate

in order to schedule and utilize resources efficiently. The process converts large-scale linear programming (LP) problems into a more easy-to-solve form by designing a dynamic algorithm that is based on the user's varying demands for service. Figure 2.1 compares in terms of high-level architecture the edge federation model with one that does not follow the approach.

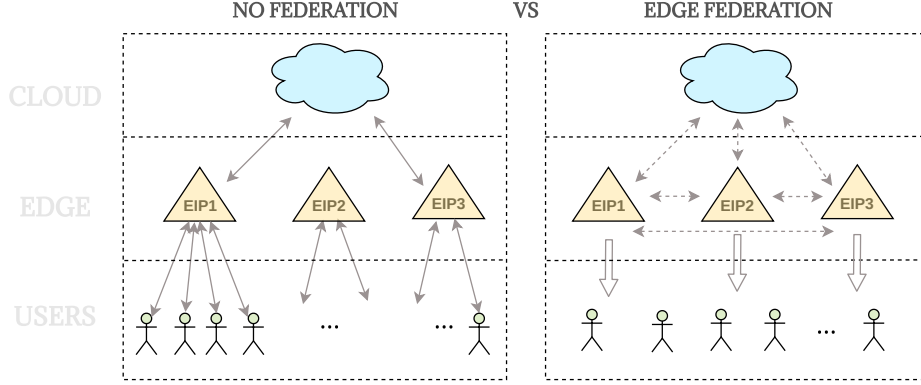


Figure 2.1: Changes in the Edge federation proposal model adapted from [1]

The *Open Infrastructure for Edge* (OIE) [15] is another proposal that aims to mitigate the lack of local computational resources on the edge. In this scenario, interested independent edge providers could share some computational capacity for edge computing needs. Such infrastructure would require, among other things, low-latency demands, low deployment costs, and protection and privacy for its users. The Open Infrastructure must also address challenges such as discovering the resources and managing them.

Edge AI methods have been proposed to enhance security and privacy in the computing continuum [16]. By adding intelligence in a distributed manner, advanced security mechanisms can be deployed at the edge of the network. As a result, attacks could be detected early, limiting their propagation, and improving security overall. Moreover, edge AI-based methods are processed locally instead of in the cloud, reducing the need for raw user data to be transferred over untrusted networks between the user and the cloud. Further advantages of edge AI also include its ability to implement decentralized AI algorithms. This prevents a single point of failure in the system (as with centralized AI algorithms), and attacks are limited to the local area.

### 2.1.3 Orchestration Frameworks and Standards

We explore some existing orchestration frameworks and standards that offer promising solutions to the challenges listed above. We gain a greater understanding of how these state-of-the-art orchestration frameworks work so that we can get inspired to design our use case. In regard to the Kubernetes multi-cluster approaches we explored in the previous subsection, we present a service technology that works for each of the approaches. For the Kubernetes-centric strategy, we see K8s Federation v2 [17], a powerful mechanism that can deploy and manage applications and services in a multi-cluster environment. It

provides a control plane that interacts with multiple clusters and manages the propagation of services and applications using user-defined policies. To have a working federation control plane, a cluster must be registered, and then it can be added to the federation. In addition, Federation v2 introduces higher-level behavior management as a concept, enabling users to make sophisticated decisions such as where resources should be scheduled across clusters. Concerning the network-centric approach, we present Istio [18], a service that extends Kubernetes with programmable, application-aware traffic management, telemetry, and security. It also offers secure service-to-service communication in a cluster with TLS encryption, strong identity-based authentication, and authorization. Load balancing is automated for HTTP, gRPC, WebSocket, and TCP traffic.

ClusterAPI [19] is a prototype tool that provides a declarative set of APIs for cluster creation, configuration, management, and deletion. Figure 2.2 illustrates the two types of clusters based on their role in the system. The management cluster hosts the controller managers for ClusterAPI, and the target Kubernetes clusters are created and managed by the management cluster.

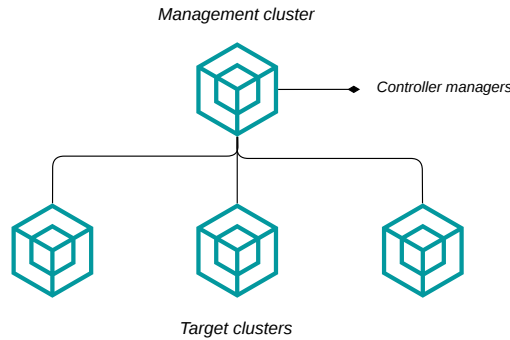


Figure 2.2: ClusterAPI high-level components adapted from [19]

The Kubernetes Web View [20] provides an interactive list of Kubernetes resources, providing a web-based version of kubectl for troubleshooting and incident response scenarios. Some of the main design principles and goals include:

- Downloadable resource lists for processing (spreadsheets, CLI tools) and storing.
- Support label selection.
- Provide a common operational picture (e.g. during incident response).
- Add custom "smart" deep links to other tools for troubleshooting and incident response.
- Keep the Frontend as simple as possible (pure HTML) to avoid accidental problems.

Deployed by independent groups, EdgeNet [21] is a public Kubernetes cluster dedicated to network and distributed systems research. The classic Kubernetes model is extended to the edge, bringing multi-tenancy, geographical deployments, and single-command node installation. The multi-tenancy environment allows tenants to be isolated and limited resources to be

shared fairly so that multiple organizations can simultaneously benefit from the edge cloud. EdgeNet also enables a feature for easy deployment of containers based on the locations of nodes. In order to set up an edge cloud, institutions can use a one-command node installation procedure, reducing their entry barriers.

A vital step to ensure security is to protect the system such that it only allows access to resources (such as APIs) from authorized users. Some of the most used standards to cover the security authorization processes are OAuth and OpenID, both acting as Single Sign-On (SSO) standards.

OAuth (Open Authorization) 2.0 standard [22] is an Authorization Framework standard (RFC 6749) that provides secure delegated access to client applications to use their data or features in another application. This could be a third-party application. After the user authorizes the application to use the corresponding access authority, the authorization server generates a limited lifetime access token, also known as *grant type*, to grant specific access to the protected resource by exchanging this token within the Authorization Server and the Resource Server. Some advantages of using unique keys over users' passwords are that it provides more security and reliability, does not compromise the user's trust and the application does not fail when the password is changed.

The OpenId [23] is an authentication tool that verifies the identity of end-users and provides profile basic information when logging in, employing encryption in a REST-like manner. The process to authenticate clients is done by a key exchange, similarly to OAuth but with an extension; when a user asks for the access token, an additional field called ID Token is returned. This token is a JSON Web Token (JWT) that can store information embedded, such as user ID, username, and ID Token expiration date.



## 2.2 Oakestra Structure Framework

In this section, we introduce Oakestra [24], a lightweight orchestration framework for edge computing. It lets multiple network operators contribute their resources to a shared infrastructure, hence minimizing investment costs to achieve dense computing at the edge. Under the complex context of edge environments, it allows cloud application developers to deploy and manage their services. As shown in figure 2.3, Oakestra is composed of a Root Orchestrator and a Cluster Orchestrator per each cluster. The applications are deployed inside worker nodes, that are located inside a Cluster Orchestrator.

One of the main contributions of Oakestra architecture is that it consolidates resources in logical hierarchies, with a root orchestrator managing clusters of servers. Its federated clustering enables to flexibly scale infrastructure horizontally while providing context separation among resources managed by different operators [2]. In the following sections, we see in more detail the principal components of Oakestra.

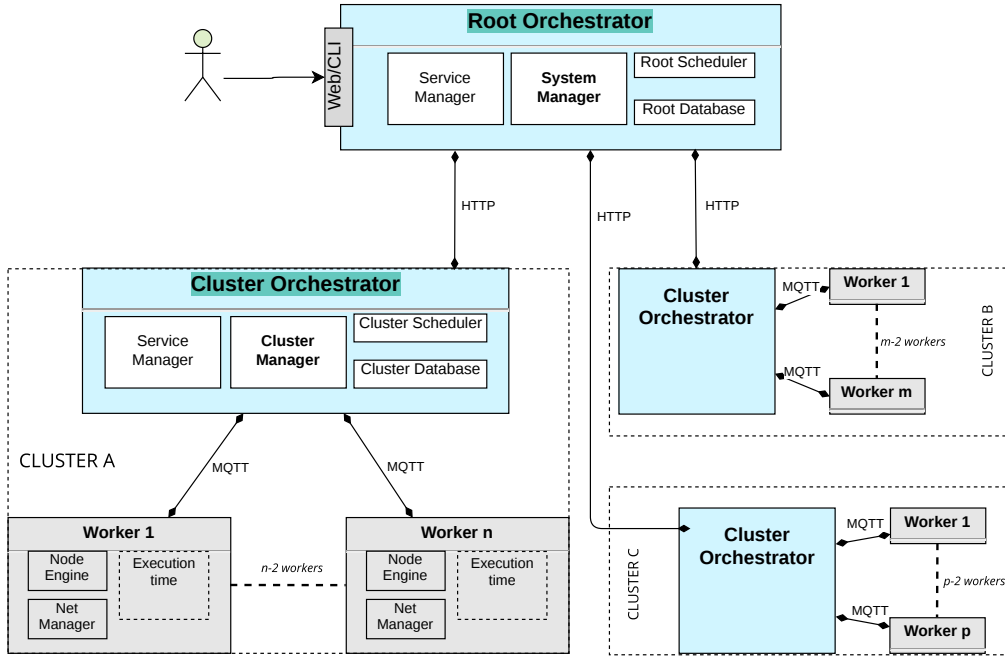


Figure 2.3: Oakestra architecture adapted from [2]

### 2.2.1 Root Orchestrator

The Root Orchestrator is the centralized control plane of the Oakestra framework. It consists principally of the following elements:

- **System Manager.** It is in charge of managing the API endpoints that enable the user to interact with the framework, including the user interface, such as deploying an

application. We examine later on that, to pair and run clusters in the Root Orchestrator, the System Manager is an important element as it is much involved in the process.

- Service Manager. It monitors all operational services that reside in different clusters.
- Root Database. It stores all relevant information about the clusters and workers participating in the system. As the system supports a multi-user approach, not only the services but also the registered users and their attributes are stored.
- Root Scheduler. It calculates a placement for a particular service.

A Root Orchestrator can be installed on a variety of devices, which increases reliability and provides greater stability, as well as makes the system more scalable. He receives all monitoring information from the clusters. To enable communication between the individual components, APIs are used. Users can deploy services and the Root Orchestrator takes care of forwarding a suitable cluster to the specific service [25].

### 2.2.2 Cluster Orchestrator

A Cluster Orchestrator consists of components similar to a Root Orchestrator. The Service Manager placed in the Cluster Orchestrator has instead a local view of the services deployed within the cluster boundaries. The Cluster Manager (who replaces the System Manager) is responsible for managing a group of workers. A message broker via MQTT enables communication of the Cluster Orchestrator with its workers. Moreover, the Cluster Orchestrator can publish commands for the worker and can read the status data (CPU, memory) published by the worker nodes. It is then the Cluster Manager's responsibility to update this status data in real-time to the Root Orchestrator on which the nodes are running. Further on, we explore how we set up the initial communication between the Cluster Manager and the System Manager to run the cluster for the first time.

### 2.2.3 Worker nodes

Worker nodes are computing devices that run Node Engine software and have an operating system. New workers can be added to a cluster using the Cluster Manager. After the cluster and worker exchange initial information, the worker can be considered in the scheduling process of the cluster. Moreover, at regular intervals, the worker publishes its monitoring data.

### 2.2.4 User interface

To interact with the system, API endpoints are defined to perform different operations, such as deploying a new service. To call these the endpoints and access the respective resources, a command-line base (CLI) and a graphical control (GUI) interface are provided. Users can be divided into three categories based on their roles, each of which provides different rights [25].

- **Application Provider:** Users can 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:** Admin user performs as both Application Provider and Infrastructure Provider. He can also add and manage new users in the system.

Up to now, the user interface (in both the GUI and CLI) was designed for application providers and administrators. A user can create, edit, delete, deploy and view the status of a service and the user must be authenticated to interact with the system. In terms of architecture, the web client is divided into two layers: the UI and the service layers [25]. The UI layer displays the user interface and decides what component to display. Communication between the web client and the System Manager is handled by the service layer, including the API requests and the authentication service.

Oakestra dashboard is an easy-to-use interface for deploying applications across the Oakestra infrastructure. It provides application provisioning, including configuration and deployment of applications. The web interface is built on Angular, a platform based on TypeScript and with a component-based framework that provides scalable web applications [26]. Its main advantage is that it includes many well-integrated libraries. The framework includes routing capabilities and basic components like controllers and services. In addition, it includes elements for smooth communication between client and server, modules for unit testing, and mocks for HTTP backend REST calls. The component-based architecture also provides strong encapsulation and isolation, facilitating code reusability. The CLI consists of the command-line tool *oakestra-cli* that allows the control of the system in the terminal. Thus, the system has a headless version; except for the visualization of some connections, the CLI provides the same functionality as the GUI and can be used for the same purpose.

### 2.2.5 Project Structure and Service Deployment

First, let us see how the Oakestra repository is structured and the files we need for the deployment of a Root Orchestrator.

```
1 oakestra/  
2   root_orchestrator/  
3     system-manager-python/  
4       system_manager.py      # contains logic for the System Manager  
5       Dockerfile  
6       docker-compose-amd64.yml  
7   cluster_orchestrator/      # details omitted for clarity
```

The deployment of the Root Orchestrator is done with Docker, thus the developer must have Docker installed on his machine. The following Docker-compose command must be run from the *root\_orchestrator* folder:

```
1 docker-compose -f docker-compose-amd64.yml up --build
```

The above procedure is analog to running the Cluster Orchestrator. The frontend can be also deployed using a docker file. For the client to connect to the System Manager, the IP address of the Root Orchestrator must be defined. Having Angular already installed on the machine, a local development server (only for development purposes) can be created by running the following command:

```
1 ng serve
```

The *oakestra-cli* tool can be installed using a *pip install* and the appropriate wheel file. A VPN can also be used in this case so that the host where the tool is being executed can connect to the Root Orchestrator directly.

## 3 System Design

Multi-cluster distributed topologies are often complex and challenging to implement, even more so when it comes to federated systems across different operators; set-up clusters in a multi-edge infrastructure is not a trivial task. We have the goal of improving the QoS when setting up clusters in a framework. By doing so, we would mitigate the difficulties encountered when running and managing clusters on a multi-edge infrastructure, where multiple clusters from multiple operators could be running at the same time. Users should be able to follow a clearly defined strategy when attaching their clusters. Therefore, the cluster registration process must be simplified, yet effective, by implementing an efficient pairing procedure between a Cluster Orchestrator and a Root Orchestrator. The web interface should also include new features that facilitate cluster registration and resource allocation once a cluster is connected to the Root Orchestrator. Moreover, as security is of utmost importance, we want to ensure that the transmission of information is as secure as possible by using reliable methods to authenticate the parties involved. Two keys are introduced: the *pairing key* and the *secret key* that the next sections explain further. In this chapter, we focus on the handshake procedure to handle the attachment of clusters to an existing control plane that we identify as the Root Orchestrator. The procedure encompasses three macro steps: registering, attaching, and monitoring clusters.

### 3.1 Requirements

The cluster handshake procedure is driven by a number of requirements that gather together the main principles needed for such a context, previously outlined in section 2.1.1.

- R1 **Cluster registration:** Clusters must be registered in order to identify them within the framework.
- R2 **Cluster identification:** Clusters must be identified when they are running within the framework.
- R3 **Infrastructure provider role:** The infrastructure role must be defined as one of the system roles, responsible for managing clusters.
- R4 **Data integrity:** Data cannot be deleted, modified, or fabricated without authorization.
- R5 **Permission management:** Users are allowed to manage for which they have permission.
- R6 **Data privacy:** The information a user owns is hidden from others.

**R7 Cluster data visualization:** Updated cluster data should be periodically displayed to cluster owners.

### 3.2 System Modeling and Integration

An infrastructure provider should be able to attach entire clusters to the Root Orchestrator and monitor them (R3). Our goal is to integrate this role in a system where it coexists with an application provider and an administrator role as well. The use-case diagram (fig. 3.1) generated provides a better understanding of how the user can interact with the system. The overall process has a generic behavior and can be managed via a command-line interface or an implemented dashboard if preferred.

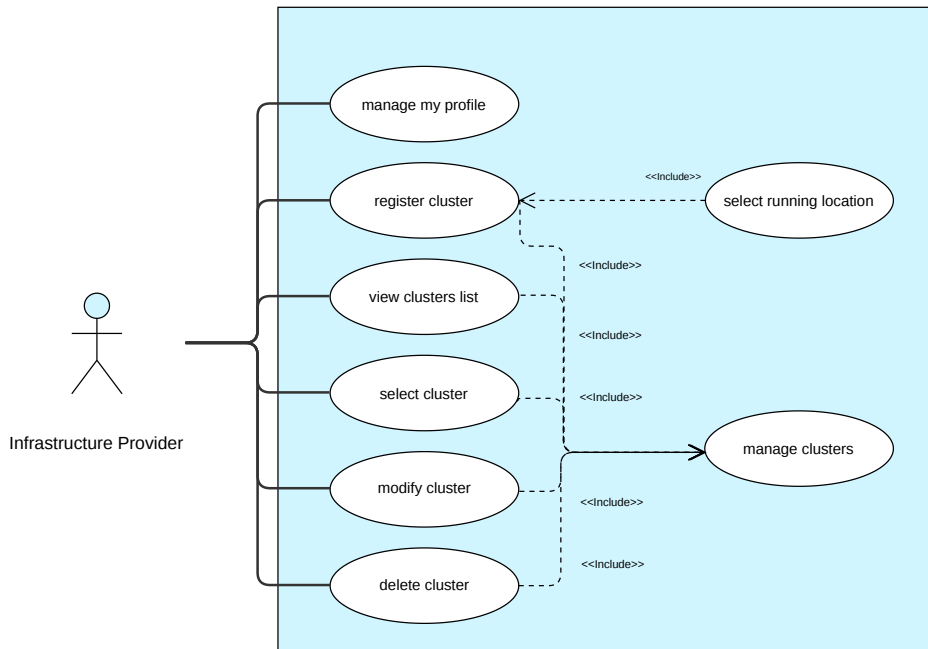


Figure 3.1: Use case model of the infrastructure-provider

An infrastructure provider can register, modify, select, delete, and view the status of his clusters through a user interface. To approach these tasks the user must be registered and logged in to the system. A cluster can be registered to get further paired and run in the system by introducing its name and location. The user should select a registered or active cluster he owns and view the details of its status. In addition, a user should be able to modify and delete his clusters at any time. Moreover, a user should be able to manage his multiple clusters from a list view, and his profile's information details.

To support the model, we have set up a reasonable communication flow between the three main elements involved in the process: the user, the Root Orchestrator, and the Cluster Orchestrator. The sequence diagrams, shown in Figure 3.2, 3.3 and 3.4, provides a high-level

description of the handshake procedure to pair a cluster with the Root Orchestrator. As mentioned earlier, the communication flow is split up into three steps: registering, attaching, and running clusters. We discuss how it works and what information is exchanged at each stage in the following sections.

### 3.2.1 Register cluster

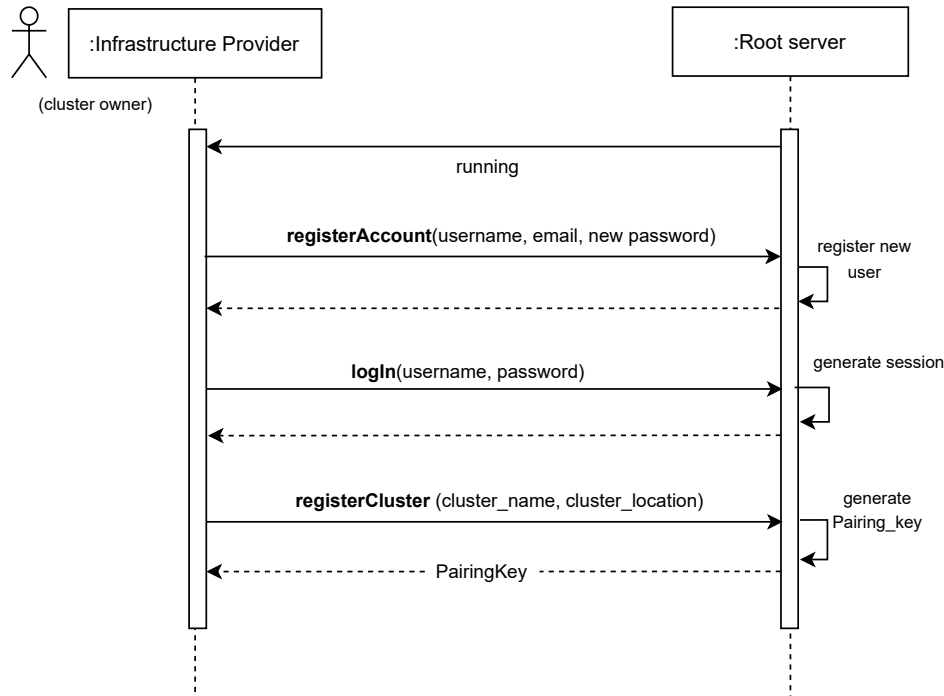


Figure 3.2: Sequence diagram of the process of registering a cluster with the Root Orchestrator

Let us assume we are infrastructure providers looking to run our clusters in the working environment. To get started, the Root Orchestrator(2.2.1) has to be running. To become a user of the system, we have to register our new account, as we can observe in figure 3.2. While the graphical user interface is recommended for better usability, users can also use the command-line tool. Once logged in, the account is registered with a unique username and password, and we can then add our cluster (R1).

As mentioned in the Requirements section, in a multi-edge context one of the principles for an infrastructure provider is to be able to attach new clusters with the desired details (R2). To achieve that, for the registration, we specify a name and a location area for the cluster to be running. Then, the data is captured by the Root Orchestrator along with the user data and it creates an instance of a cluster that gets registered into the system. As part of the secure handshake process, a secure *pairing key* is generated shortly after the cluster information is processed, and it is shown to the user for attachment purposes. Unlike the other parameters given, and for security reasons, this key should not be saved in any database. Additionally, it

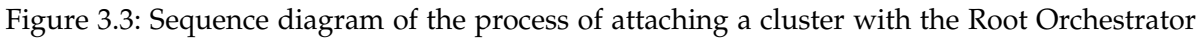
should only be valid for a specific period of time. Next chapter we delve deeper into how this task can be smoothly accomplished.

### 3.2.2 Attach cluster

Cluster Orchestrator takes action now. We set the cluster's fields from the Cluster Manager and connect to the System Manager, the element from the Root Orchestrator that coordinates the attachment of new clusters. Checks are performed at this point to ensure that the cluster that is paired has already been registered in the system. The System Manager also verifies that the pairing process has not yet been completed for that particular cluster. Lastly, the *pairing key* that the user received in the registration step is sent as well in the request to authorize the specific user that owns the cluster to establish the connection. Consequently, the key needs to be validated to ensure security in the handshake procedure, and that no external untrusted individual can perform this action. The sequence diagram from figure 3.3 illustrates the messages' interaction to approach the initial handshake.

To complete the pairing process, and to follow successful validation of the *pairing key*, now we invalidate this key. Thus, we can prevent an attacker from stealing our key, which gives us a security advantage. In this step, the system generates a new key that authorizes the cluster from now on. We call this key a *secret key*, and it is sent back to the cluster manager in response to the communication. The *secret key* is generated to authorize the infrastructure provider to perform some action with the cluster, such as running it, adding a worker, or modifying some data. Like the *pairing key*, the *secret key* also has an expiration date to enhance security. As a consequence, the user must introduce the *secret key* every time he runs his cluster in the Root Orchestrator. Whenever it gets expired, the user must provide his credentials to log in once more, enabling Two-Factor Authentication; the system then provides him with a new *secret key*. Further details on how we have implemented these keys in Oakestra and how they differ from each other are explored in the next chapter. As of now, the user playing the role of the infrastructure provider can monitor its active cluster in the framework (R3), as its cluster has been attached to the Root Orchestrator.





### 3.2.3 Monitor clusters

We want to achieve control and monitoring of our devices within the context of a multi-cluster federated environment, as discussed above. Our framework architecture should then give us the ability to manage clusters at root level while still delegating tasks to nodes. As users of the system, we can edit, modify or delete the clusters we own. However, every time we want to access an endpoint to do such action, we should provide the *secret key* that identifies the respective cluster (R4). Furthermore, the cluster sends its status periodically to the Root Orchestrator, so the orchestrator is up-to-date on the current resource utilization of the cluster. Status updates must be authenticated through the *secret key* as well. Figure 3.4 shows the flow sequence that makes the cluster's status updates possible. As we see, data such as CPU and memory is sent and it is done in a periodic manner, so the Root Orchestrator is updated at all times. Only when the *secret key* is correctly provided, the permission is granted and the cluster data from a root level is updated.

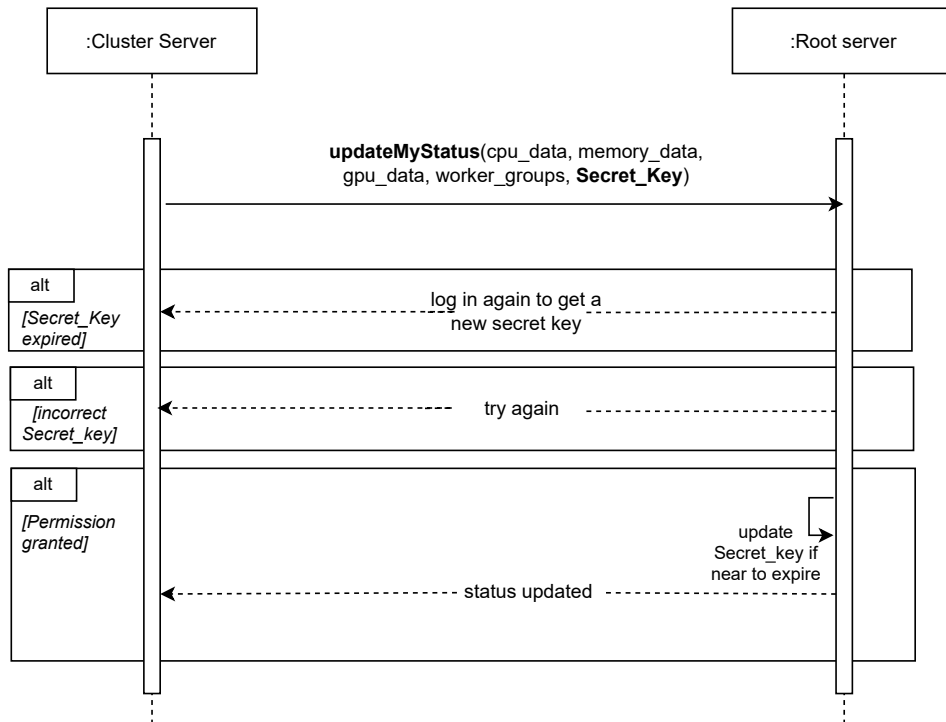


Figure 3.4: Sequence diagram showing the periodic status updates of a cluster

Clusters can be monitored via the API and/or the CLI, which display individual information about both newly added and already paired clusters. As a result, we can manage cluster resources while acknowledging their status at all times (R7), having the right permission (R5). In addition, we have no knowledge of the cluster information other users have in the system (R6).

## 4 Implementation and Evaluation

This chapter examines the logical design implemented to set up clusters in the Oakestra framework using the handshake procedure discussed in the previous chapter. Our first step is to explore the technologies used, and then we describe the implementation in Oakestra of the cluster's pairing process. Finally, we evaluate our system.

### 4.1 Overview of the technologies used

In order to fully comprehend how the implementation took place, we need to be familiar with the well-known technologies we used. In this section, we cover what REST APIs are and why they are important for enabling interactions. We also look at Flask library of Python in detail since it is highly utilized in our system. Lastly, we talk about JSON Web Tokens, which are used to authorize clusters.

#### 4.1.1 REST API

REST API stands for Representational State Transfer Application Programming Interface. A REST API, sometimes referred to as RESTful API, is a mechanism that enables an application or service to access a resource within another application or service [27]. HTTP requests are used to communicate standard database functions such as creating (*POST*), reading (*GET*), updating (*PUT*), and deleting (*DELETE*) records within a resource (as shown in Figure 4.1 below).

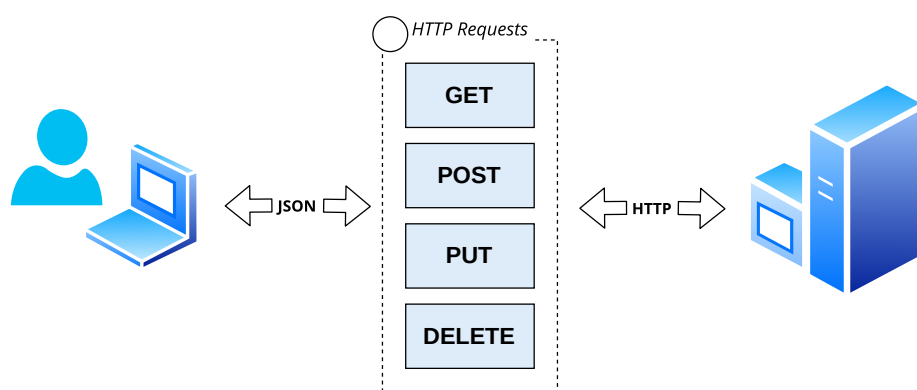


Figure 4.1: REST API schema

REST APIs have many benefits [28], including:

- Scalability and Statelessness. As we can observe in figure 4.1, REST APIs are not tied to the client or the server. REST is stateless, which means the server does not store anything across requests, thus allowing for horizontal scalability.
- Error Monitoring. REST provides API responses, enabling any errors to be reported.
- Caching. More requests can be handled with fewer resources by caching responses and transmitting them instead of re-doing the request.

In order to easily visualize and interact with the API's resources, we use Swagger UI [29] to save the Root Orchestrator's accessible endpoints of Oakestra. It provides easy-to-use visual documentation, automatically generated from an *OpenAPI* Specification. The figure 4.2 below shows a portion of our API Documentation regarding cluster management.

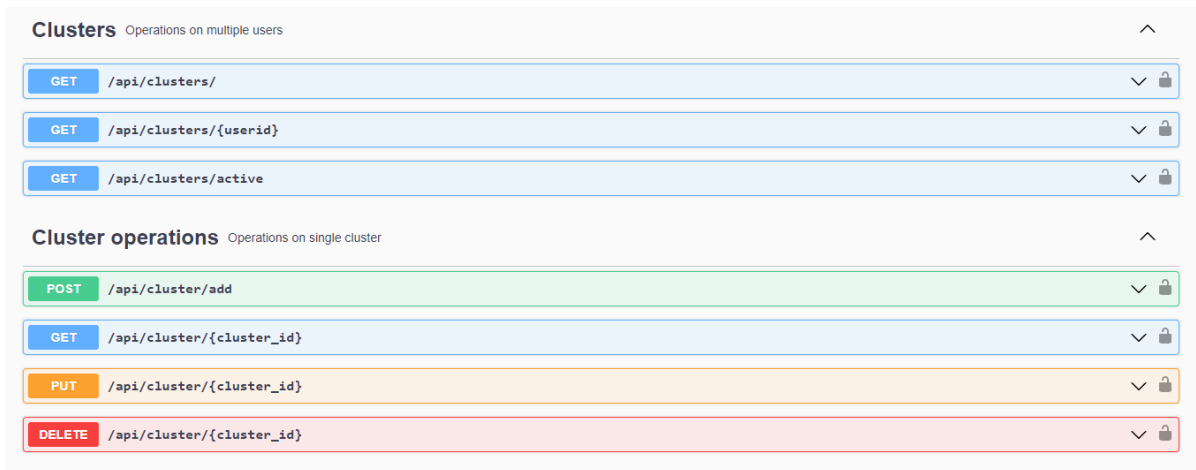


Figure 4.2: API endpoints defining cluster management in Oakestra with Swagger UI

### 4.1.2 Flask and Flask-RESTful

Flask is a framework based on Python that is used to build a REST API. Flask-RESTful is a lightweight extension of Flask that integrates the existing ORM (an objected-oriented technique for querying and manipulating data in a database) and libraries, providing best practices with minimal setup [30]. We create a *Flask* object that represents our application:

```
1 from flask import Flask
2
3 app = Flask(__name__)
```

Now we can associate views functions to routes. In the following code we can see the `@app.route()` Flask decorator assigning the route  `'/'`  to the `hello_world()` function.

```
1 @app.route('/')
2 def hello_world():
3
4 app.logger.info('Hello World Request')
```

The `@app.route()` decorator can specify valid HTTP methods as a second argument. In this way, we build scenario-specific constraints into a route for REST API endpoints. By default, Flask views only accept GET requests. Here is how it works:

```
1 @app.route("users/", methods=['GET', 'POST'])
```

The Flask interface is also very intuitive and offers easy-to-understand features. The following code shows features we have used to set up configuration values:

```
1 app.config["JWT_SECRET_KEY"] = token_hex(32)
2 app.config["JWT_ACCESS_TOKEN_EXPIRES"] = timedelta(minutes=10)
```

### 4.1.3 JSON Web Tokens

To ensure the secure management of the handshake procedure, we need to pay attention to how the authentication is handled. As of yet, we have defined REST-API routes to allow access to a variety of resources. However, before access is allowed, user identity has to be validated. To accomplish this, different methods are considered: password-based, session-based, and token-based authentication.

Password-based authentication consists of providing the username and password of a client to verify his identity, typically required online. This method of authentication can put the application at risk because if the request is leaked, hackers can gain access to the client's credentials.

An improved approach is a session-based authentication, where the server creates a unique session identifier for each login and it gets stored in memory in the database. Sessions tokens are valid only for the specific session but may be used multiple times. Then the server can validate the user by mapping the token to the session. It is however not the best solution, since it introduces some bottlenecks as well, primarily because it is stateful. Data storage on the server for each client can strain the server's performance and overload it significantly, making it a challenge for distributed modern systems with multiple servers to work efficiently [31].

Token-based authentication, contrariwise, has no session, it is stateless, good for balancing server loads and it cannot be modified once created. Rather than storing session identifiers on the server, the session data is encoded and encapsulated directly in the token. As stated in the official documentation, JSON Web Token (JWT) is an open standard (RFC 7519) that defines a secure way to exchange JSON objects between two parties and because they are digitally signed, they can be trusted [32].

The JSON Web Tokens consist of three parts: the header, the payload, and the signature. while the header provides general information about the token and contains the algorithm used to generate the signature, the payload contains the data and identity of the token. In

the signature, there is stored a secret key (generated from the header's algorithm) that only the server should know about. In figure 4.3 we can observe an example of what is saved in an access token created to authenticate the pairing of a specific cluster within the store information about the respective claims used is explained in detail in section 4.2.3.

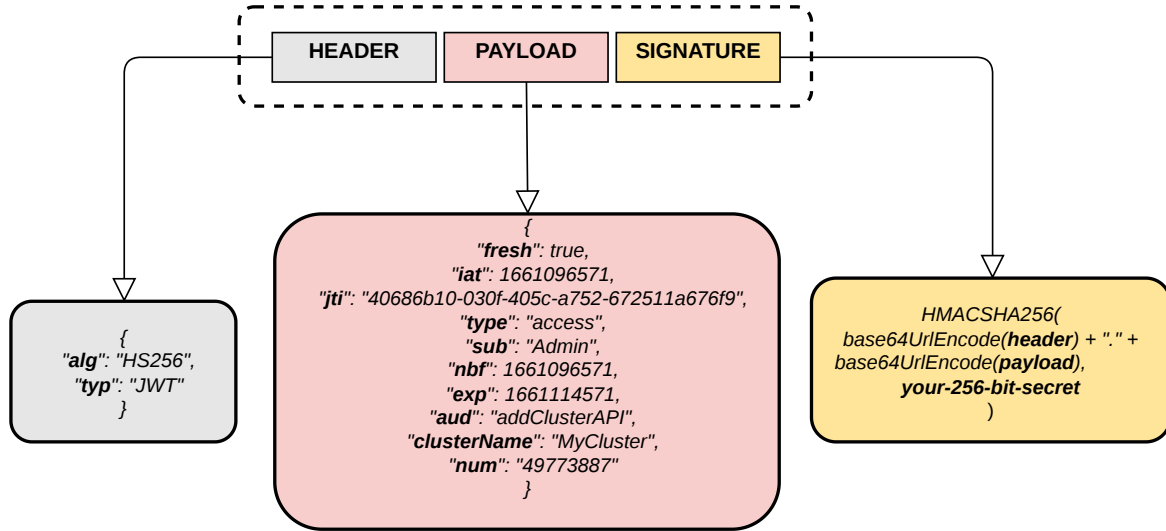


Figure 4.3: Information encapsulated in a JWT Access Token to pair a cluster to the system

Some of the advantages that JWT provides us [33]:

- **Stateless.** The JWT itself, as seen in the payload from the image 4.3, already contains the user and cluster basic information to validate it.
- **Portable.** A single token can be used by different parties.
- **Good Performance.** If we compare it to session tokens, JWT can reduce the load of accessing a database, providing a faster experience.
- **Decentralized.** The token can be generated anywhere.

## 4.2 System Implementation

In this section, we see in more detail how the handshake procedure, described in the System Design chapter, has been implemented in Oakestra. We first explore how we architect our Flask application with Blueprint and then which steps we must follow to register a cluster into the system. We then discuss the implementation of the JSON Web Tokens and the technology used to set up the initial handshake communication between the Cluster Manager and the System Manager. Finally, we review the current system architecture and discuss how we have implemented the cluster management features in Oakestra’s graphical user interface.

### 4.2.1 Working with Flask Blueprints

An application can be customized by applying blueprints that define views, templates, static files, and other elements [34]. When an application is big and has multiple endpoints, Blueprint can help us categorize those endpoints by purpose. We use them for Oakestra as they help to simplify how we register operations, structuring them based on the functionality they provide. For our application, we have saved all the blueprint files in a common folder. Moreover, we keep the logic outside the blueprints, in a *services* folder. The files structure looks like the following:

```
1 root_orchestrator/  
2   system-manager-python/  
3     blueprints/  
4       __init_it__.py  
5       clusters_blueprints.py  
6       application_blueprints.py  
7       # and more ...  
8   services/  
9     cluster_management.py  
10    application_management.py  
11    # and more ...
```

The following code shows a part of how we implement this Flask Blueprint in *clusters\_blueprint.py*. It contains the view at the route */add* that takes care of registering a cluster (line 5), with the function *register\_cluster(data, current\_user)* implemented in the *cluster\_management.py* file (omitted for clarity). In addition, in line 21 we can also see the declaration of another view that also uses the same Blueprint route decorator.

```
1 clusterblp = Blueprint(  
2   'Cluster operations', 'cluster operations', url_prefix='/api/cluster'  
3 )  
4  
5 @clusterblp.route('/add')  
6 class ClusterController(MethodView):  
7     @clusterblp.arguments(schema=cluster_op_schema, location="json",
```

```

8     validate=False, unknown=True)
9     @clusterblp.response(200, content_type="application/json")
10    @jwt_required()
11    def post(self, args, *kwargs):
12        data = request.get_json()
13        current_user = get_jwt_identity()
14        try:
15            resp = register_cluster(data, current_user)
16            return resp
17        except Exception as e:
18            traceback.print_exc()
19            abort(401, {"message": str(e)})
20
21    @clusterblp.route('/<cluster_id>')
22    class ApplicationController(MethodView):
23        @clusterblp.response(200, content_type="application/json")
24        @jwt_required()
25        def get(self, cluster_id, *args, **kwargs):
26            # some code ...

```

Above we can see that we first create the Blueprint object called *clusterblp* and then we use the route decorator to add views linked to them. We have defined this Blueprint object to manage the endpoints for cluster operations in Oakestra. In our case, we define the route to get to the *ClusterController* class by going through the endpoint */api/cluster* (defined in the Blueprint object) plus */add* (indicated with the *@clusterblp.route()* Flask decorator). Similarly, we get to the *ApplicationController* through the endpoint */api/cluster/<cluster\_id>*.

To inject the cluster data as the argument into the view function in the code above, we use the *Blueprint.arguments* decorator. We indicate a Schema, which is later on deserialized, validated, and injected into the view. Here we see how the *cluster\_op\_schema* is declared:

```

1 cluster_op_schema = {
2     "type": "object",
3     "properties": {
4         "cluster_name": {"type": "string"},
5         "cluster_latitude": {"type": "string"},
6         "cluster_longitude": {"type": "string"},
7         "cluster_radius": {"type": "string"},
8         "user_name": {"type": "string"}
9     }
10 }

```

However, before all of that, our application has imported the Blueprints from the *\_\_init\_\_* class of our repository as follows:



```
1 from blueprints.clusters_blueprints import clusterblp
2 # more imports
3
4 blueprints = [
5     clusterblp
6     # more Blueprint objects
7 ]
```

### 4.2.2 Cluster registration in the system

Once we have understood how Blueprint works, we focus on how the cluster data is registered in the Root Orchestrator, achieving the first step of the handshake procedure. As seen in figure 3.2 from the previous chapter, to create and save the cluster in the Root Database, from the dashboard or cli we must pass the following data: the cluster name and location (that includes latitude, longitude, and radius) and the user name and user id. In future sections, we examine how the frontend works, but right now we are focusing on the backend.

The cluster information is sent from the cluster owner using the `/api/cluster/add` route, and the view starting at line 11 from the code above is executed. The called function `register_cluster()` is called from the `cluster_management.py` file contains the logic of the procedure. At this point, a few simple steps are followed before the cluster can be successfully added to the Root Database (thus enabling the next handshake step to be carried out):

1. The information provided is in the right format.
2. Check that no other cluster with the same name and location is saved waiting to be paired.
3. A unique cluster id is generated and the new instance is added to the Root Database.
4. A JWT *pairing key* is created (details in section 4.2.3).
5. Send back the *pairing key* generated of the cluster to the cluster owner as the response.

In the following section, we see in detail how the JWTs work and how we use them for pairing proposes.

### 4.2.3 JSON Web Tokens Implementation

To provide secure authentication of users, there are two different JWT defined in the Oakestra framework: the access token and the refresh token, only differing in the *type* and *exp* additional claims from the payload. The access token has a lifetime of ten minutes and authenticates a user. The refresh token ensures that the user is not required to re-authenticate after every ten minutes. If the lifetime is expired, it is possible to get a new access token with the refresh token, as this has a lifespan of seven days. Thus, our *auth* workflow works as follows:

1. To log in, the client provides his email and password, which are sent to the System Manager.
2. The System Manager then verifies that the email and password are correct and responds with both the access token and the refresh token.
3. The client stores the tokens in the browser's local storage or in a hidden file in case of the command line interface is used to log in.
4. Before the client sends a new request to the API, the System Manager decodes and validates the access token and, if it is valid, the bearer token can be added to the request's header. Otherwise, the refresh token is used to issue a new access token.

In case the user forgets their password, the *forgot password* option can be called in order to receive the reset token by mail. This token is not a JWT but a cryptographically strong random number that has a lifetime of three hours and can be used to reset the password and re-access the system [25].

### Flask-JWT Extended library contribution

For the JSON Web Tokens validation of the Flask REST-API endpoints, the library Flask-JWT-Extended [35] is a very useful tool, as it not only adds support for using protected routes, but it also provides many helpful and optional features built to make working with JSON Web Tokens easier [36]. The following functions are used in our application:

- `@create_access_token()` to generate JWT Access Tokens.
- `@create_refresh_token()` to generate JWT Refresh Tokens.
- `@decode_token()` to decode JWTs.
- `@get_jwt_identity()` to get the identity (username) from the payload of the JWT in a protected route.
- `@get_jwt()` to extract more data from the payload of the JWT.

Flask-JWT-Extended also supports decorators, providing great flexibility to our application. It works by defining an outer function that takes a function as an argument. In Python is called using the form `@expression` (*expression* being the name of the decorator) just before the definition of the outer function. Some already implemented decorators by the extension that we have used are `@jwt_required()` and `@verify_jwt_in_request()`, in order to verify that a valid JWT is present in the request. In addition, we have also created our own decorators that extend their functionality. To accomplish this, we have followed the following basic syntax [37]:

```
1 def my_decorator_func(func):
2
3     def wrapper_func():
4         # Do something before the function.
5         func()
6         # Do something after the function.
7     return wrapper_func
```

There are also other flexible and useful features from Flask-JWT Extended library implemented in Oakestra; we'll examine them in more detail in future sections.

### JWTs Implementation for cluster's pairing proposes

To approach the pairing procedure with JWTs, OAuth is initially considered. This solution, however, is not suitable for our needs. As previously stated, a user can be defined either as Administrator, Application Provider, Infrastructure Provider, or them combined. Admins have access to everything and can deploy everywhere. However, a regular user, that is not an admin, needs the authorization to access certain data and manage the deployments which specifically belong to him, or either resource shared with him. The OAuth protocol has previously been shown to work effectively in authorization scenarios to grant access to users only in the context of the client application layer. In our context, however, the approach must be suitable for both GUI and CLI tools. Moreover, OAuth identifies the user with a unique token and, since a user can have multiple clusters and different permissions attached to them, multiple authorization tokens would be managed from the same user.

As we have seen in chapter 3, we divide the handshake procedure to handle the attachment of clusters to the system into three big steps: registering, attaching, and monitoring clusters. In the sequence diagrams from figures 3.2, 3.3 and 3.4 we saw that a *pairing key* and a *secret key* are generated. They are JSON Web Access Tokens and encapsulate information about the cluster that wants to be active in the system (*pairing key*) or the one already paired (*secret key*). In both cases, we use them to authenticate the cluster within the user with the help of JWT. We take advantage of the stateless characteristic of the token and we encapsulate the following data in the *payload* part:

```
1 {"iat": datetime.now(),           // issued at [current time]
2  "aud": aud,                     // for pairing is "addClusterAPI"
3                                   // and for running is "runningCluster"
4  "sub": cluster['user_name'],
5  "clusterName": cluster['cluster_name'],
6  "num": str(randint(0, 99999999)) //generated random number
7  }
```

For key verification, we begin by decoding the token, obtaining the information encapsulated, and comparing and validating the information obtained. We can understand better the process through the abstract diagram in figure 4.4. The two keys not only differ for the *aud* claim, but also for the *fresh* claim and the expiration time; the *pairing key* is set to five hours,

and the *secret key* is set to thirty days. The reason why the last one is really long is that for the token to be valid it has to pass an implemented decorator as well, which checks the freshness of the token. How this *fresh* claim is managed is explained in more detail in the following section.

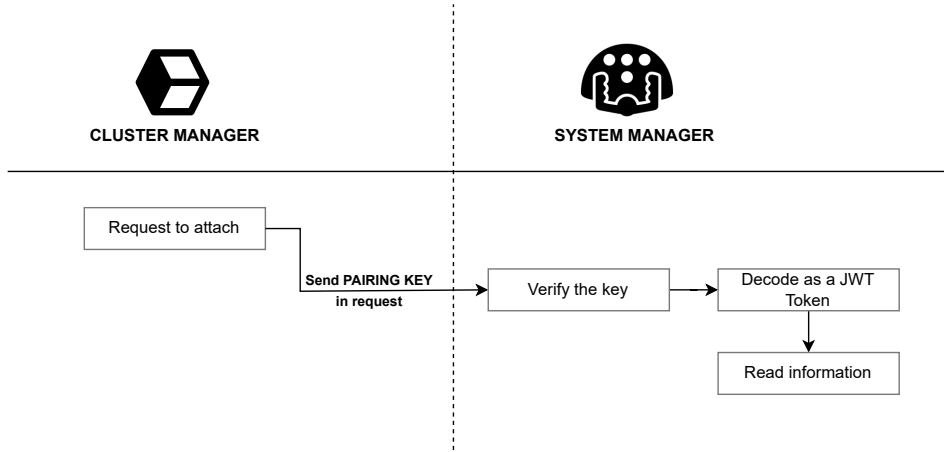


Figure 4.4: Abstract flow diagram of how the pairing key is treated

If a registered cluster is run for the first time providing the pairing key attached to the cluster, but this key has already expired (five hours have passed), the instance is deleted, and the user must re-register the cluster again. Listed below is the directory structure of the Root Orchestrator along with the principal components used to start up the Flask application from the System Manager. As observed, inside the *system-manager-python* folder there is the *system\_manager.py*, where the logic for the System Manager resides. It is then inside the *roles* folder where the *securityUtils.py* file is. The purpose of this file is to communicate with the Flask-JWT Extended library that contains the JWT functions we use for the authorization step, as seen previously.

```

1 root_orchestrator/
2   system-manager-python/
3     roles/
4       securityUtils.py  # communicates with Flask-JWT Extended library
5       system_manager.py # contains logic for the System Manager
6       Dockerfile
7       docker-compose-amd64.yml
  
```

### Token Freshness Implementation

After login into the dashboard, the server responds with a short-living access token and a long-living refresh token. These tokens are used whenever a user wants to access certain resources, by sending the access token with the request. Whenever we try to access a resource but our access token has expired, we receive an *Unauthorized* error, and this is when the

refresh token is required to identify the user, create a new access token and use this new token to get the respective resources. A refresh token is a long-lived JWT that can only be used to create new access tokens [38].

To return to our context, we want the cluster owner to be authorized each time he wants to run his cluster or perform any action in the framework involving the cluster to be authorized. Previously, we have discussed that we generate and return to the cluster owner a secret key, in the format of a JWT, that encapsulates information such as the token's expiration date, among other things. In dealing with the key's expiration date, we considered two approaches.

The first approach would be to generate both an access token and a long-lived refresh token at the same time and return both values as a response to the cluster call. Then, the values must be saved. If the access token expires in a subsequent request, an error message would appear and the user would need to provide the refresh token to generate a new access token and redo the request with the new token.

As a result, it requires multiple kinds of tokens, access to more endpoints, and is more complex and less straightforward. Instead, the second approach, currently being implemented, requires only the access token, and it works with the *fresh* claim to ensure maximum security as well. To contextualize, this claim is by default set to false, and it can be used to protect endpoints. *Flask-jwt-extended* supports the token freshness pattern, which restricts access to certain routes to only tokens with the *fresh* claim to True, otherwise, the user must verify his password again. Even if the token is not fresh, other routes continue to work as normal. For instance, users might not be able to change their email address without a fresh token, but they can still use other endpoints.

Neither methods need to be independent of the other, but for our use case, we are considering only the second approach. When creating the access token, the *fresh* parameter can be defined either as a *Boolean* or as a *datetime.timedelta* object [39]. This second option let us mark the secret key as fresh for a given amount of time after it is created. Here is how we create the secret key as a JWT from the system manager:

```
1 secret_key = create_jwt_secret_key_cluster(cluster_id,  
2 expiration_date=timedelta(days=30), additional_claims,  
3 fresh=timedelta(days=3))
```

And here we define this function that calls the predefined from the Flask-JWT-extended library in order to create an access token, passing all parameters we have previously seen:

```
1 def create_jwt_secret_key_cluster(identity, expiration, claims, fresh):  
2     return create_access_token(identity=identity, expires_delta=expiration,  
3     additional_claims=claims, fresh=fresh)
```

As observed in the code above, the freshness date is short and the expiration date is quite long, making an analogy to the access and refresh token definition. Therefore, as long as the user does not access anything that is critical or more secure within 3 days, they do not need to re-login.

To call the API endpoint to authorize to run clusters, the System Manager first checks if the header of the request has a valid JSON Web Token and that in the payload of this token

the *fresh* claim is set to True. This is implemented using custom decorators that support the Flask-JWT-Extended library. This is done by calling the decorator `@jwt_fresh_required()`. If the token is a valid JWT and the mentioned claim is True, the request is processed, if not, access to the API endpoint is denied.

```
1 def jwt_fresh_required():
2     def wrapper(fn):
3         def decorator(*args, **kwargs):
4             try:
5                 jwt_required(fresh=True)          # it validates the JWT if it
                                                    # is marked with fresh
6             except Exception as e:
7                 print(e)
8                 return {"message": "The token is not valid."}, 401
9             return fn(*args, **kwargs)
10        return decorator
11    return wrapper
```

The decorator implemented indeed only checks the freshness of the token (using the *jwt required* defined function of Flask-JWT Extended), as our situation entails not wanting to re-login every time the *fresh* claim gets old. It is important to understand that, even though the claim has been defined with a *datetime* format, it is still fresh. Nonetheless, there is a predefined decorator that checks if the fresh value is old, the endpoint is inaccessible. We do not use it for now, but it might be considered to access some endpoints in further implementation.

Overall, three possibilities can happen when running our clusters:

1. The token is still fresh and not expired: it runs normally.
2. The token freshness date is expired: the server generates another access token as the secret key, and it is sent back to the user. This time the *fresh* claim is set to False.
3. The token is not fresh anymore: it does not pass the decorator function and it asks the user to log in again, managing the two-factor authentication.

#### 4.2.4 Communication between Cluster Manager and System Manager

In order to set a both secure and fast way to exchange messages to do the handshake between the Cluster Manager and the System Manager, in the beginning, two communication protocols within the IoT (Internet of Things) are considered: MQTT and WebSocket. In both cases, once the connection is established, the client and server can exchange messages in a two-way channel, listening and speaking simultaneously. Nevertheless, it is worth examining the following differences to take a decision:

MQTT uses a publish-and-subscribe model, where messages are broadcast and distributed to receivers, known as *subscribers*, then published to a broker on a topic [40]. The WebSocket

protocol provides low latency, the persistent bidirectional channel between the client and the server, and after the connection is made, they can exchange up to a two-byte frame of data [41]. Both communication protocols perform similarly when it comes to message loss. However, while MQTT is a good option for network usage, WebSocket is best suited for memory usage and CPU utilization [42]. Therefore, WebSocket is the selected approach to do the initial handshake between clusters and the Root Orchestrator. We now see how this WebSocket communication is set up between the two parties and then how the information is exchanged with the help of a sequence motion diagram.

### Set-up

To implement the communication channel in Oakestra, we use the extension *Flask-SocketIO* and the official client library *SocketIO*, a protocol that combines both *WebSocket* and *HTTP*. The Cluster Manager uses *SocketIO* to establish the connection to the System Manager. The following code shows how we import and instantiate the libraries.

```
1 from flask import flash, request
2 from flask_socketio import SocketIO, emit
3
4 app = Flask(__name__)
5 socketio = SocketIO(app, async_mode='eventlet', logger=True,
6 engineio_logger=True, cors_allowed_origins='*')
```

From the Cluster side, the *SocketIO* for the client is imported and instantiated as well.

```
1 import socketio
2 sio = socketio.Client()
```

In addition, in the main we must start the server with the call showed below:

```
1 eventlet.wsgi.server(eventlet.listen(('0.0.0.0', int(MY_PORT))), app, log=
    my_logger)
```

From the asynchronous services that the package relies on, *eventlet* is used, as it supports *WebSocket* and it is the best performant, as stated in the official documentation [41]. In the next section we deep more into how the communication between the two parties is established.

### Sequence Motion

As seen in chapter 3, we want to add, attach and monitor our clusters. In section 4.2.2, we saw how we are approaching the first step. Thus and up to this point, we differentiate two scenarios where the Cluster Manager and the System Manager communicate:

1. To attach a new cluster.
2. To run a paired cluster.

For both scenarios, the number of WebSocket messages is the same, however, the information differs since the keys to authenticate are different for instance the first case requires access to the Root Database. In figure 4.5 we see how the exchange of messages is happening for the scenario of attaching a cluster to the Root Orchestrator. For this diagram, exception cases (negative results) are not considered. The interaction with sockets is based on events. The Socket instance we earlier named as *sio* from the client-side (Cluster Manager), emits three special events:

- **connect()**, event fired upon connection and re-connection (step (1)).
- **connect\_error()**, event fired when the server denies the connection.
- **disconnect()**, event fired upon disconnection (step (4)).

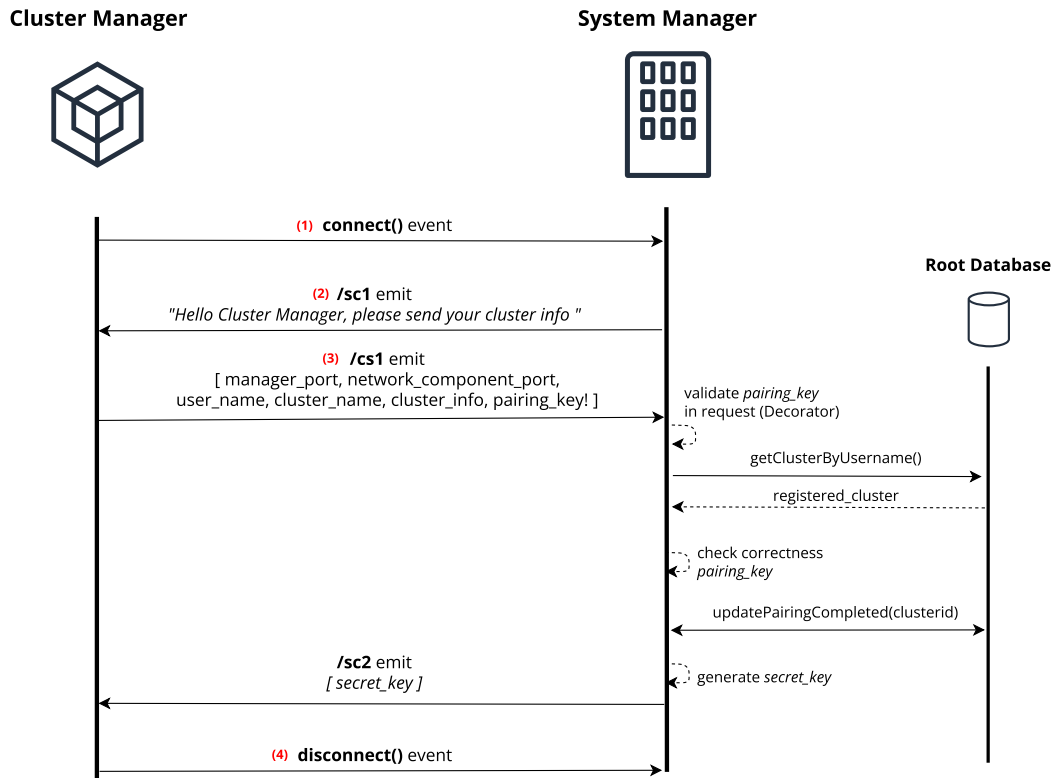


Figure 4.5: SocketIO Sequence Diagram for the Cluster attachment

Once the connection is established and, in order to send messages through the channel, both functions *send* and *emit* can be used for unnamed and named events respectively. For our case, we use named events, as they also appear to be the most flexible, as it avoids the need to include additional metadata for describing the message type [43]. To create an event handler for the handshake procedure we do

```
1 @socketio.on('cs1', namespace='/register')
```



As we can observe, we are also defining a SocketIO *namespace*, as it allows us to be using multiple independent connections for different processes on the same physical socket.

We demonstrate below how to send the messages that correspond to steps (2) and (3) respectively.

– *System Manager side*

```
1 emit('sc1', {'Hello-Cluster_Manager': 'please send your cluster info'},  
2 namespace='/register')
```

– *Cluster Manager side*

```
1 sio.emit('cs1', data, namespace='/register')
```

As we recall, this WebSocket communication occurs for pairing purposes, and Cluster Manager and System Manager exchange API REST messages thereafter with the HTTP protocol.

#### 4.2.5 Architecture

As seen in figure 2.3 of section 2.2, Oakestra framework is organized into distinct tiers with their essential elements. To enable the handshake procedure for attaching clusters to the Root Orchestrator, certain parts of the architecture's elements have been modified. The Cluster Manager and the System Manager have both been enhanced, as have the MongoDB databases. One important feature is that the API endpoints have been extended, so clusters can be managed better.

The diagram 4.6 shows the architecture components in charge of the pairing process. We have the CLI and the web client on one side, where we can differentiate the components as explained in the section 2.2.4 of the Background chapter. The Web Client has been extended to integrate new features to manage clusters from the Dashboard. On the other side, there are the Cluster Orchestrator and the Root Orchestrator, which contain the Cluster Manager and the System Manager respectively. As discussed in section 4.2.4, the two parties initially communicate to attach clusters with the WebSocket protocol. After the cluster has been paired and is running, API requests are exchanged for communication. Both Root Database and Cluster Database have been expanded to store all relevant information about the participating clusters and workers within the system.

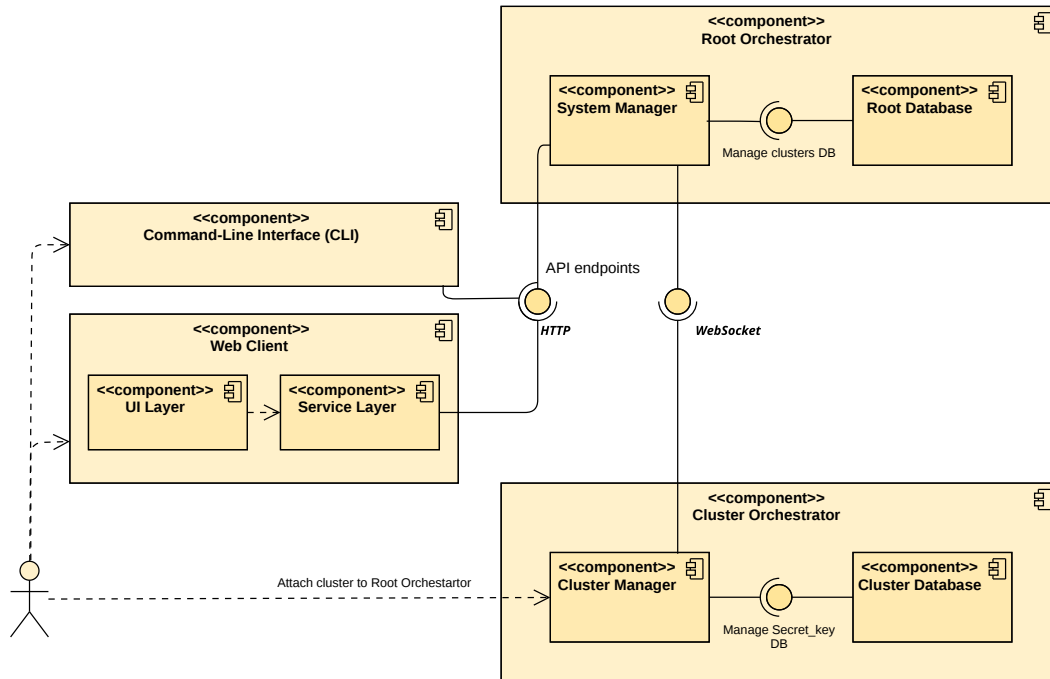


Figure 4.6: Oakestra component diagram of adding a cluster

## 4.2.6 Frontend

This subsection targets the frontend implementation regarding how the Oakestra dashboard has been modified in order to enable the overall pairing process of clusters to the Root Orchestrator to take place.

### User interface

The dashboard, based on the existing Oakestra framework, is already designed for the application provider view, as shown in section 2.2.4. As an infrastructure provider, we should be able to manage clusters from the web interface. By taking advantage of the scalability that Angular apps provide, we could easily integrate our work into the framework.

The main page of the frontend is divided into a header (the Oakestra logo, the user name, and the profile), side navigation, and a content area. Our focus now goes to the left side of the dashboard, where the side navigation is located (fig. 4.7). Here is where we differentiate between the different features of the user from a role's standpoint. From one side there is the deployment of applications and services and from the other side the running and managing of clusters. It also contains links to the help page (which had already been implemented before). Our goal is to make the pairing process of clusters as straightforward as possible.

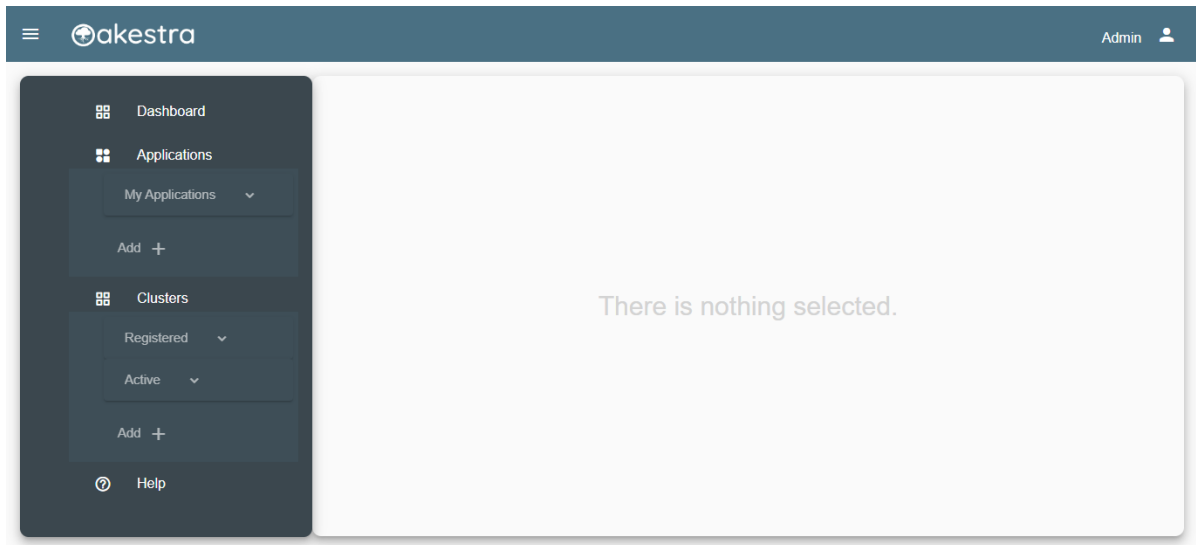


Figure 4.7: Dashboard main page with nothing selected

Using the side navigation, we can access the list of our own clusters by clicking on *Clusters* (as we can observe in 4.11) and adding new clusters using *Add*.

To start with the first step of the handshake procedure, we can go through the *add* cluster's option from the left side of the interface and register a cluster indicating a name and a location (fig. 4.8). If the fields are not filled or are in the wrong format, the permission is denied and a warning screen appears. In case of success, the information is then sent to the backend. The cluster is registered in the Root Orchestrator and a dialog with the corresponding generated *pairing key* appears (fig. 4.9). To now attach the cluster to the Root Orchestrator, we need to run our cluster indicating its registered data. For that we have two options:

1. The fastest way is to click the *download* button of the dialog, and a configuration file is generated, that looks like the following:

```
1 CLUSTER_NAME=my_cluster
2 SYSTEM_MANAGER_URL=192.168.1.189
3 USER_NAME=Admin
4 CLUSTER_PAIRING_KEY='eyJ0eX... imlNGO'
```

And in the terminal, from the *cluster\_orchestrator* folder, we need to run the following copy command:

```
1 cp <address_of_the_file> .env
```

2. The other option is to copy the key to the clipboard and set-up the fields manually from the terminal as environment variables.

In the GUI we can visualize the list of our clusters, enabling the monitoring through the Oakestra federated cluster management (fig. 4.11). Clusters can have two states: registered

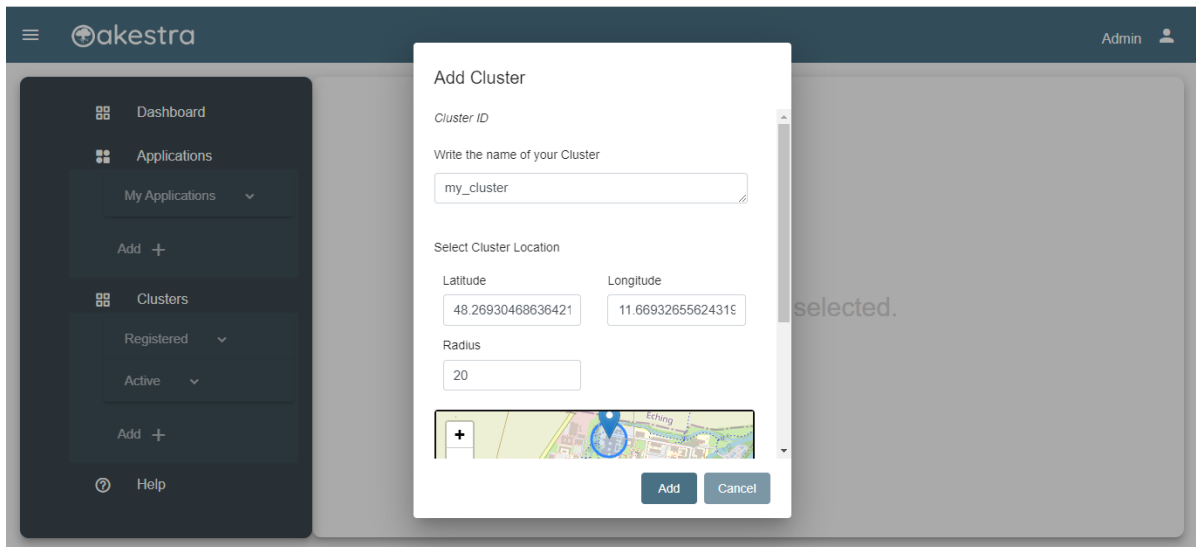


Figure 4.8: Add a cluster to the Dashboard

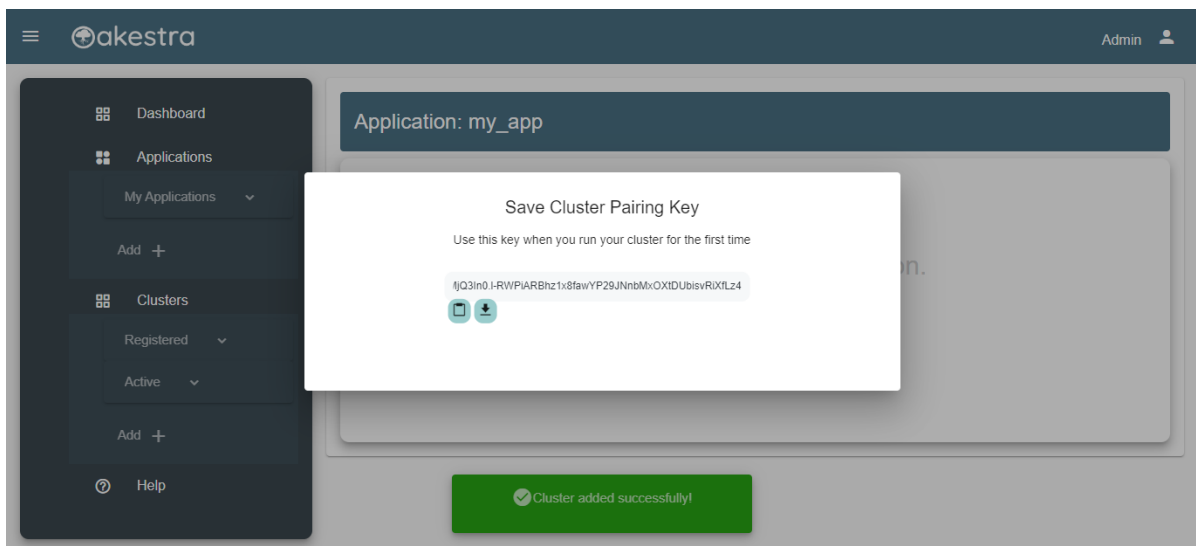


Figure 4.9: Dialog showing the pairing key needed to run the cluster from the Root Orchestrator

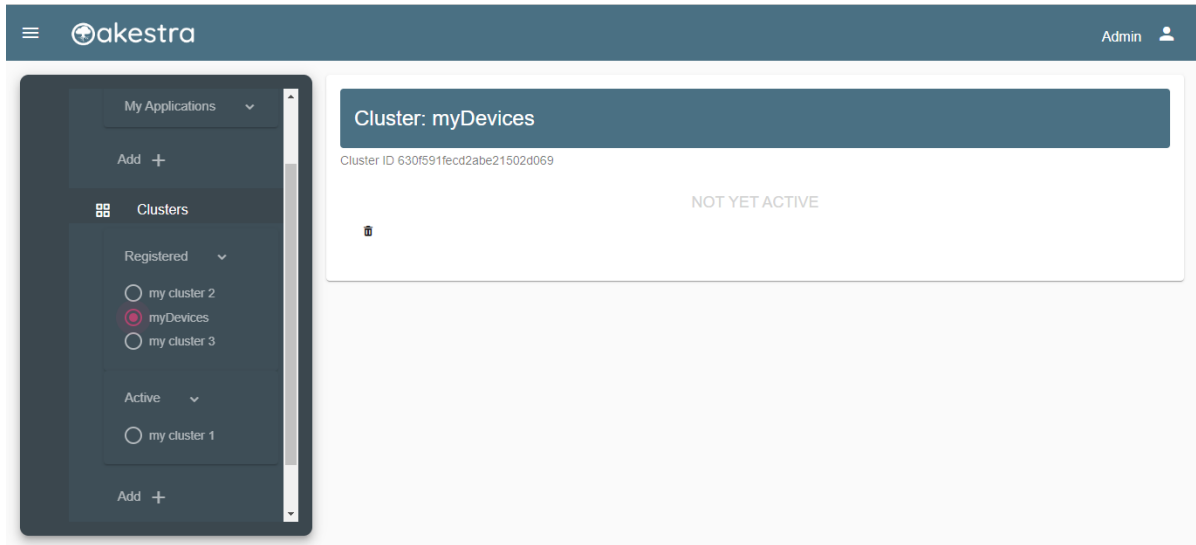


Figure 4.10: Cluster selection view

and paired. From the left side navigation bar but we can select each of them to see further details, divided into two categories (fig. 4.10). Moreover, the web interface also enables the user to navigate comfortably minimizing the screen and having a good QoS as well (fig. 4.12).

### Leaflet map

An interactive map allows users to select the location and the area where they wish their cluster to run. This map has been implemented with the library of *Leaflet*, an open-source JavaScript library, easy to use that provides a well-documented API for mobile-friendly interactive maps [44].

The interactive map shown in the dashboard enables us to mark a location with the respective latitude and longitude, as well as the radius of the zone. To start, we initialize our map on the local variable we have defined with a given center and zoom, and set *setattributionControl* to false, as we do not want to display attribution data on the map.

```

1 this.map = L.map('map', {
2   center: [this.lat, this.lon],
3   attributionControl: false,
4   zoom: 14
5 });

```

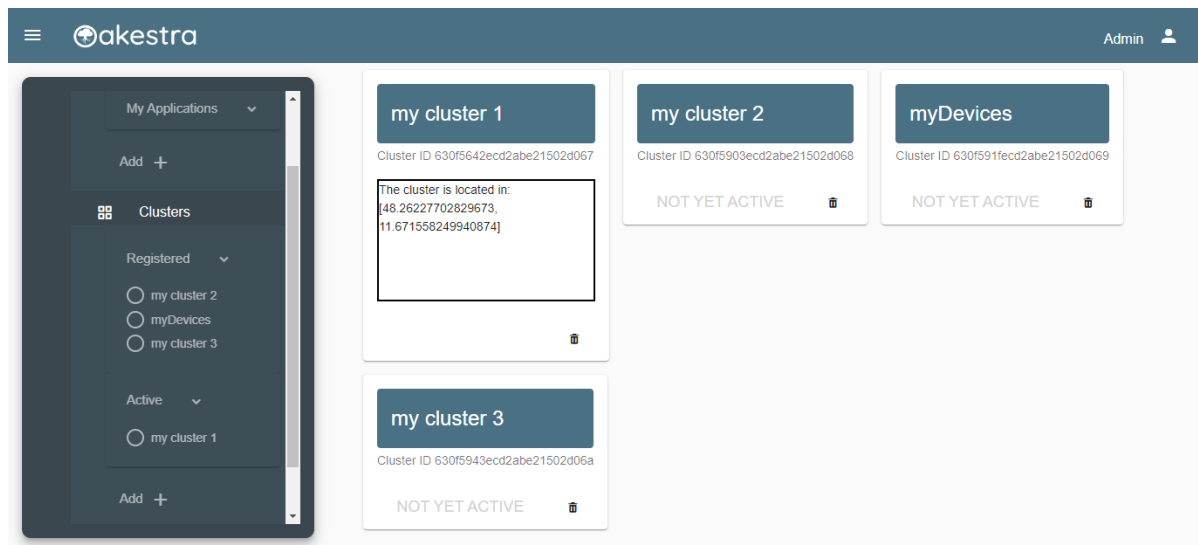


Figure 4.11: Cluster list view

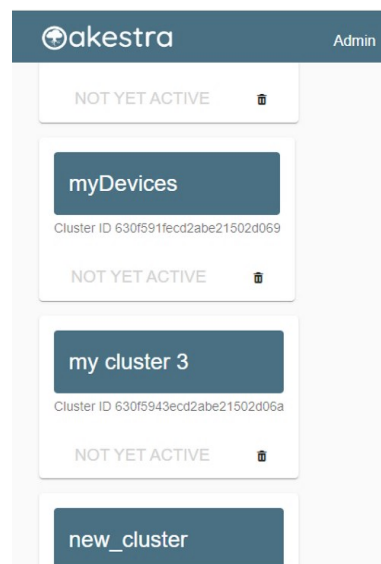


Figure 4.12: Responsive UI

To display the clickable icons to point to a location, we instantiate a `Marker` object given a geographical point. Then, we can easily get the coordinates (latitude, longitude) by using the function `addTo` to the instance we created on our map. To show the zone that covers our running cluster, we instantiate a `Circle` object and this time, in the options object, we specify the value of the chosen radius in pixels. Then, we add the marker to our map, as it is seen in the following code:

```
1 this.marker = L.marker( [e.latlng.lat, e.latlng.lng] )
2 this.circlemarker = L.circleMarker( [this.lat, this.lon], {radius: e.radius} )
```

In the following code, we can also see more useful features of the *Leaflet* library that we have implemented for our use case.

```
1 this.map.removeLayer(this.marker);
2 this.map.removeLayer(this.circlemarker);
3
4 let search = GeoSearchControl({
5   provider: new OpenStreetMapProvider(),
6   marker: {
7     draggable: true
8   },
9   });
10 this.map.addControl(search);
```

Using the function `removeLayer` (lines 1 and 2), we are able to remove location markers and circle markers. To manually look for a specific place on the map, a search feature is enabled, using the geocoder from *Leaflet.GeoSearch* (line 4), from *OpenstreetMap Nominatim* to locate places by address. The following figure 4.13 shows how it appears on the GUI:

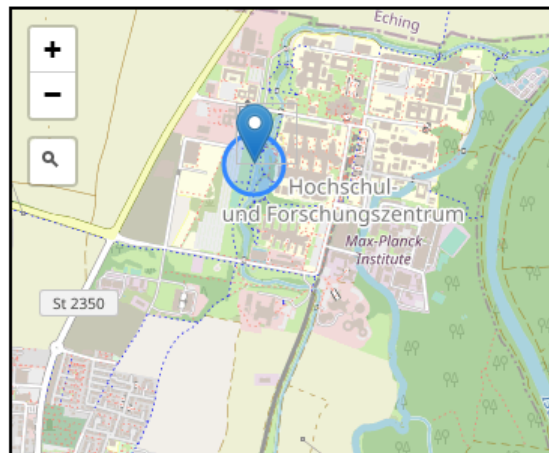


Figure 4.13: View of the map in Oakestra Dashboard

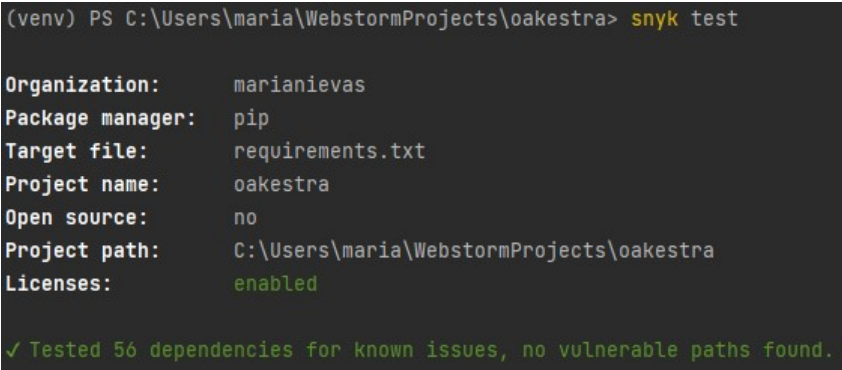
## 4.3 Evaluation

In this section, we will first evaluate our application through a static analysis testing tool to check if there is any vulnerability detected through the cluster pairing process, and report our results. Next, we will examine the security of the authorization tokens by modeling a scenario where we could encounter a cyberattack and see how the tokens could respond.

### 4.3.1 Static analysis evaluation

According to a recent security report [45], almost half of Python projects present vulnerabilities. It is clear from this statement that testing must be given weight as part of a development project so that an application is well secured and protected. Fortunately, around 90% of vulnerabilities have known fixes. The Snyk tool [46] is a developer security platform for securing code, dependencies, containers, and infrastructure as code. This tool supports many programming languages, including Python, and integrates with a variety of IDEs, CI/CD tools, and issue management tools. In essence, it analyzes the code of an application and tells if there are any vulnerabilities, particularly in the dependencies.

To test our application with the new cluster pairing process implemented, we apply a static analysis with the Snyk tool, and therefore check if there are defects in our code. As the first step, we must create an account on the Snyk website, and then install the CLI tool as an npm package in our local environment with the command `npm install snyk -g`. As a result, we can run the basic commands provided by the tool. To run the test we must first authenticate with the account we created in the first place, and then run the command `snyk test`. What this command does is test the dependencies within the project to seek issues. After running the command, and as anticipated, we find zero vulnerabilities among the 56 dependencies tested, as it is shown in figure 4.14. Furthermore, we use the Snyk plugin of Pycharm, the IDE where our work environment resides. As a result, we are able to check more security threats in terms of code security and code quality, classifying each potential vulnerability as low, medium, high, or critical severity. Once again, no security issue is found.



```
(venv) PS C:\Users\maria\WebstormProjects\oakestra> snyk test

Organization:    marianievas
Package manager: pip
Target file:     requirements.txt
Project name:    oakestra
Open source:     no
Project path:    C:\Users\maria\WebstormProjects\oakestra
Licenses:        enabled

✓ Tested 56 dependencies for known issues, no vulnerable paths found.
```

Figure 4.14: Snyk CLI test results



### 4.3.2 Security analysis of JSON Web Tokens

In the current state of the handshake procedure, we have implemented a solid authorization mechanism capable of preventing uninvited access to cluster resources. As explored in section 4.1.3, the JWT token is an encoded, URL-safe string that encapsulates some data and that is cryptographically signed [32]. While the JWT guarantees data ownership, it does not guarantee encryption, as the data inside the token is serialized. In other words, if someone intercepts the token, they are able to see what is inside. Moreover, there is the possibility that some attacker could gain access to Oakestra and intercept these tokens. If we are at risk of these attacks, how can we protect ourselves?

We previously saw in figure 4.3 that a JWT consists of three parts: the header, the payload and the signature. Our tokens have been generated using an algorithm specified in the header; let us explore it now. That is an HS256 (HMAC with SHA-256), a symmetric hashing algorithm that uses one secret key [47]. It is called symmetric since the secret key is shared between two parties, used both for generating and validating the signature. The following code is an example of how the token with the mentioned algorithm is generated [48]:

```
1 # Encode with the Base64-URL algorithm, and concatenate them
2 data = (base64urlEncode(header) + '.' + base64urlEncode(payload))
3 # Use the HS256 algorithm with "SECRET_KEY"
4 signature = HMACSHA256(data, SECRET_KEY)
5 # Complete the JWT token
6 JWT = data + "." + base64UrlEncode(signature)
```

The 'SECRET\_KEY' is needed to generate the JWT token, which is a very sensitive value as its exposure would compromise the entire security mechanism. The reason for this is that whoever owns this value can create valid tokens, and our system cannot distinguish between legitimate and malicious tokens (those created by an attacker). For example, an attacker could create a JWT that the application would consider valid. In our system, however, we generate this 'SECRET\_KEY' securely with a random text string of 32 Bytes in hexadecimal. It is important not to confuse this value with the *secret key* we use for cluster authentication. The randomly generated key is not shared with any party nor saved in any database, making it extremely secure.

An attacker's target would be to alter a field of the payload to gain more access rights [49]. An example would be trying to modify the *iat* claim, which stands for *issued at*. Typically, it is used to determine the age of a JWT by identifying the time at which it is issued. By modifying its content with an older date, the application might think the token is about to expire. As long as the token is still valid, the attacker could take advantage of it to gain access to resources. In addition, changing the *sub* claim could be attractive to the attacker to gain more privileges as well. This claim reveals the identity of the owner of the token, so for example, it could be changed from an Infrastructure Provider to an Admin, which is a more powerful role. However, the signature of a JWT prevents these two actions from taking place. JWTs are signed by the source and no middleman can modify them once sent, so servers can trust the data they contain.

We have seen that it is practically impossible for the attacker to alter the security of the system as long as he does not get the 'SECRET\_KEY'. In this way, we can be assured that the pairing key and the secret key we use to identify a registered cluster and run a cluster respectively to our framework are well-secure. As a final reminder, any Oakestra REST endpoint must always require the corresponding authorization token is always required in the header.

## 5 Conclusion and Future Work

With the approach presented in this thesis, setting up resources in a multi-cluster environment can be straightforward. In order to pair clusters within a running edge infrastructure, we developed an easy-to-follow three-step process. With this design in place, we built the Oakestra framework's interactions with fast response communications and assess their security. Moreover, we have enhanced the already existing GUI of Oakestra with new features, supporting multi-admin operations. The following features were achieved as a result of the project:

1. **Framework Interaction:** In Oakestra as infrastructure providers, we can register our cluster with its worker nodes, and attach them to a Root Orchestrator. Adding a cluster requires that the values entered are in the correct format, which makes the system more robust. Moreover, we have control of all of our integrated clusters in the framework from the system and from a user-friendly interface.
2. **Multi-Admin Design:** The final design supports multiple operators to manage their resources without interfering with each other. Our design was influenced by the principles and strategies presented in section 2.1.
3. **Access Control:** With a token-based authentication system control - using two cluster identifying keys to pair and run a cluster - the Oakestra framework secures the REST-API endpoints from unrestricted access.

### 5.1 Future Work

Through developing the project we have discovered further opportunities to improve the presented approach. Considering the interactions implemented in the framework, some technical aspects need to be addressed. It is first necessary to extend the CLI functions so that the entire handshake procedure can be carried out from there. Next, when it is time for the Cluster Orchestrator to attach its cluster, we collect the data registered from a configuration file, as seen in section 4.2.6, and copy the content into a local file where then the values are set. The approach could, however, be improved by automating it.

Further features could be integrated into the Dashboard UI to allow infrastructure providers to monitor their services more effectively. An example would be the visualization of resource spending per cluster with interactive graphs, the addition or deletion of worker nodes, and providing the needed details to make them run efficiently. Moreover, the endpoints defined to approach all these features must be well protected with the cluster secret key that identifies each cluster.

We evaluated the system with static analysis, however, this alone does not guarantee full protection to an application. In this way, it would be necessary to run security audits to evaluate the performance. An example of such a diagnosis would be penetration testing. This would test the application resilience and simulate the actions of a hacker by conducting several attacks on the process of pairing a cluster. An example would be to simulate an external party trying to run a cluster that they do not own. Additionally, fuzz testing is also convenient to identify potential failures by introducing intentionally malformed inputs into the system. This diagnosis would test the robustness and security risk posture of the whole cluster pairing procedure. Finally, Oakestra's user interface could also be tested with a usability test, which involves observing real users interacting with the system. We could then identify areas of confusion and discover opportunities to improve the overall user experience.

## List of Figures

2.1	Changes in the Edge federation proposal model adapted from [1] . . . . .	6
2.2	ClusterAPI high-level components adapted from [19] . . . . .	7
2.3	Oakestra architecture adapted from [2] . . . . .	9
3.1	Use case model of the infrastructure-provider . . . . .	14
3.2	Sequence diagram of the process of registering a cluster with the Root Orches- trator . . . . .	15
3.3	Sequence diagram of the process of attaching a cluster with the Root Orchestrator	17
3.4	Sequence diagram showing the periodic status updates of a cluster . . . . .	18
4.1	REST API schema . . . . .	19
4.2	API endpoints defining cluster management in Oakestra with Swagger UI . .	20
4.3	Information encapsulated in a JWT Access Token to pair a cluster to the system	22
4.4	Abstract flow diagram of how the pairing key is treated . . . . .	28
4.5	SocketIO Sequence Diagram for the Cluster attachment . . . . .	32
4.6	Oakestra component diagram of adding a cluster . . . . .	34
4.7	Dashboard main page with nothing selected . . . . .	35
4.8	Add a cluster to the Dashboard . . . . .	36
4.9	Dialog showing the pairing key needed to run the cluster from the Root Orchestrator . . . . .	36
4.10	Cluster selection view . . . . .	37
4.11	Cluster list view . . . . .	38
4.12	Responsive UI . . . . .	38
4.13	View of the map in Oakestra Dashboard . . . . .	39
4.14	Snyk CLI test results . . . . .	40

# Bibliography

- [1] X. Cao, G. Tang, Member, IEEE, D. Guo, S. Member, IEEE, Y. Li, and W. Zhang. *Edge Federation: Towards an Integrated Service Provisioning Model*. 2019.
- [2] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, and J. Ott. *Oakestra white paper: An Orchestrator for Edge Computing*. 4 Jul 2022.
- [3] B. L. Claus Pahl. *Containers and Clusters for Edge Cloud Architectures –aTechnology Review*. 2015.
- [4] *A business guide to hybrid/multi-cloud*. Canonical, 2022. URL: <https://ubuntu.com/cloud/multi-cloud>.
- [5] A. Nordhoff. *What is a Cluster? An Overview of Clustering in the Cloud*. CapitalOne, 2020. URL: <https://www.capitalone.com/tech/cloud/what-is-a-cluster/>.
- [6] S. Haase. *Manage Multicloud Kubernetes with Operators*. 2021. URL: <https://thenewstack.io/manage-multicloud-kubernetes-with-operators/>.
- [7] Á. MacDermott, Q. Shi, M. Merabti, and K. Kifayat. *Security as a Service for a Cloud Federation*. 2014.
- [8] *What is multi-access edge computing (MEC)?* RedHat, 2022. URL: <https://www.redhat.com/en/topics/edge-computing/what-is-multi-access-edge-computing>.
- [9] Y. Sun, J. Zhang, Y. Xiong, and G. Zhu. *Data Security and Privacy in Cloud Computing. International Journal of Distributed Sensor Networks*. 2014.
- [10] D. D. Jiangshui HongThomas. *An Overview of Multi-cloud Computing*. 2019.
- [11] G. Bartolomeo. *Enabling Microservice Interactions within Heterogeneous Edge Infrastructures*. Master’s Thesis. TUM, 15.09.2021.
- [12] *New ecosystem opportunities at the service provider edge*. RedHat, 2021. URL: <https://www.redhat.com/en/blog/new-ecosystem-opportunities-service-provider-edge>.
- [13] *Kubernetes*. URL: <https://kubernetes.io/>.
- [14] *Understanding Multiple Kubernetes Clusters*. Ambassador. URL: <https://www.getambassador.io/learn/multi-cluster-kubernetes/>.
- [15] A. Zavodovski, N. Mohan, W. Wong, and J. Kangasharju. *Open Infrastructure for Edge: A Distributed Ledger Outlook*.
- [16] H. Kokkonen, L. Lovén, N. H. Motlaghk, J. Partalay, A. González-Gil, E. Sola, I. Angulo, M. Liyanagex, T. Leppänen, T. Nguyen, V. C. Pujolzz, P. Kostakos, M. Bennisx, S. Tarkomak, S. Dustdarzz, S. Pirttikangas, and J. Riekk. *Autonomy and Intelligence in the Computing Continuum: Challenges, Enablers, and Future Directions for Orchestration*. 2022.

- [17] K. Gamanji0. *K8s Federation v2 — a guide on how to get started*. 2019. URL: <https://medium.com/condenastengineering/k8s-federation-v2-a-guide-on-how-to-get-started-ec9cc26b1fa7>.
- [18] Istio. 2022. URL: <https://istio.io/>.
- [19] K. Gamanji0. *ClusterAPI — a guide on how to get started*. 2019. URL: <https://medium.com/condenastengineering/clusterapi-a-guide-on-how-to-get-started-ff9a81262945>.
- [20] *Kubernetes Web View*. URL: <https://codeberg.org/hjacobs/kube-web-view>.
- [21] *EdgeNet: A Multi-Tenant and Multi-Provider Edge Cloud*. 2021.
- [22] *Authentication types*. Adobe, 2022. URL: <https://helpx.adobe.com/de/coldfusion/api-manager/authentication-types.html>.
- [23] D. Neal. *An Illustrated Guide to OAuth and OpenID Connect*. Okta Developer, 2019.
- [24] *Oakestra*. Github, 2022. URL: <https://github.com/oakestra>.
- [25] D. Mair. *Designing Robust Interaction Frontend for Decentralized Edge Infrastructures*. Bachelor's Thesis. TUM, 15.03.2022.
- [26] *What is Angular?* URL: <https://angular.io/guide/what-is-angular>.
- [27] *REST APIs*. IBM Cloud Education, 2019. URL: <https://www.ibm.com/cloud/learn/rest-apis>.
- [28] S. Gill. *HTTP API vs REST API: 3 Critical Differentiators*. 2021. URL: <https://hevodata.com/learn/http-api-vs-rest-api/>.
- [29] *Swagger*. SMARTBEAR. URL: <https://swagger.io/>.
- [30] *Flask RESTful*. 2020. URL: <https://flask-restful.readthedocs.io/en/latest/>.
- [31] Y. Balaj. *Token-Based vs Session-Based Authentication: A survey*. University of Prishtina "Hasan Prishtina", 2017.
- [32] *JSON Web Tokens*. URL: <https://jwt.io/introduction>.
- [33] R. Raghuwanshi. *JWT (JSON Web Tokens) Are Better Than Session Cookies*. 2017. URL: <https://dzone.com/articles/jwtjson-web-tokens-are-better-than-session-cookies>.
- [34] rpicard. *Explore Flask: blueprints.rst*. Github. URL: <https://github.com/rpicard/explore-flask/blob/master/source/blueprints.rst>.
- [35] L. Gilbert-Bland. *Flask-JWT-Extended's Documentation*. URL: <https://flask-jwt-extended.readthedocs.io/en/stable/>.
- [36] *Project description. Flask-JWT-Extended*. URL: <https://pypi.org/project/Flask-JWT-Extended/>.
- [37] *Custom Decorators*. URL: [https://flask-jwt-extended.readthedocs.io/en/stable/custom\\_decorators/](https://flask-jwt-extended.readthedocs.io/en/stable/custom_decorators/).
- [38] *Refreshing Tokens*. URL: [https://flask-jwt-extended.readthedocs.io/en/stable/refreshing\\_tokens/](https://flask-jwt-extended.readthedocs.io/en/stable/refreshing_tokens/).

- [39] *Flask-jwt-extended repository. Added support for access token freshness date.* URL: <https://github.com/vimalloc/flask-jwt-extended/pull/110>.
- [40] *How MQTT Works -Beginners Guide.* URL: <http://www.steves-internet-guide.com/mqtt-works/>.
- [41] L. Chebbi. *Real-Time in Angular: A journey into Websocket and RxJS.* 2020. URL: <https://javascript-conference.com/blog/real-time-in-angular-a-journey-into-websocket-and-rxjs/>.
- [42] M. Werlinder. *Comparing the scalability of MQTTandWebSocket communication protocols using Amazon Web Services.* School of Electrical Engineering and Computer Science.
- [43] *Flask-SocketIO.* URL: <https://flask-socketio.readthedocs.io/en/latest>.
- [44] *Leaflet.* URL: <https://leafletjs.com/>.
- [45] *Python security insights.* SNYK REPORT. September 2021.
- [46] *Snyk.* URL: <https://snyk.io/>.
- [47] W. Johnson. *RS256 vs HS256: What's The Difference?* 2022. URL: <https://auth0.com/blog/rs256-vs-hs256-whats-the-difference/>.
- [48] N. Tariq. *Attacking JSON Web Tokens (JWTs).* 2020. URL: <https://infosecwriteups.com/attacking-json-web-tokens-jwts-d1d51a1e17cb>.
- [49] N. Charushnikov. *Extending Virtual Mobility World (ViM) Platform with Flexible Analysis and Scalable User Access.* Bachelor's Thesis. TUM, 15.10.2021.