CHRISTIS Command-Line Tool

A New Way to Handle Users in The Kubernetes Cluster

Lab Report

by

Armin Aminian
3152789

submitted to
Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik IV
Arbeitsgruppe für IT-Sicherheit

in degree course
Informatik (M.Sc.)

First Supervisor: Dr. Matthias Wübbeling
University of Bonn

Bonn, March 30, 2021

Acknowledgement

I wish to record my deep sense of gratitude and profound thanks to my research supervisor Dr. Matthias Wübbeling for introducing me this interesting subject and inspiring guidance with my work during all stages.

Abstract

In this report, we introduce our tool *Christis* which makes user handling efficient and easy when the number of users who need access to the Kubernetes cluster is not small, and they have different usage patterns. This tool will help administrator to have more secure clusters. We first describe the backgrounds that are required to understand our tool. Then we focus on the tool design and which components it has. We will provide information regarding how to deploy and run our tool in your Kubernetes cluster. In the end, we describe a real and straightforward scenario that can be handled by our tool. In the Appendix, we describe **OpenID Connect Token** Authentication strategy that is very useful to get all functionalities of Christis.

Contents

1 INTRODUCTION 1

2 BACKGROUND AND OBJECTIVES 2
2.1 BACKGROUND . . . 2
2.1.1 KUBERNETES . . . 2
2.1.2 KUBERNETES CLUSTER COMPONENTS . . . 3
2.1.3 KUBERNETES AUTHENTICATION AND AUTHORIZATION . . . 4
2.1.4 HELM . . . 6
2.1.5 KEYCLOAK . . . 7
2.2 OBJECTIVES . . . 8

3 CHRISTIS CLI DESIGN AND COMPONENTS 9
3.1 THE REASON BEHIND THE CHRISTIS . . . 9
3.2 DESIGN AND COMPONENTS . . . 10
3.2.1 DESIGN IN ABSTRACT . . . 10
3.2.2 COMPONENTS . . . 11
3.2.3 DESIGN IN DETAIL AND HOW CHRISTIS WORKS . . . 12

4 HOW TO DEPLOY AND RUN CHRISTIS 16
4.1 Christis Requirements . . . 16
4.2 DEPLOY CHRISTIS . . . 16
4.3 DEPLOY CHRISTIS COMPONENTS . . . 17
4.3.1 CHRISTIS DATABASE . . . 17
4.3.2 CHRISTIS API . . . 17
4.3.3 CONFIGURE CHRISTIS CLI TO ACCESS CHRISTIS API . . . 18

5 CHRISTIS IN A REAL SCENARIO 19
5.0.1 SCENARIO . . . 19
5.0.2 IMPLEMENTATION . . . 19

6 CONCLUSION AND FUTURE WORKS 27
6.1 CONCLUSION . . . 27
6.2 FUTURE WORKS . . . 27

7 APPENDIX: IMPLEMENT AN AUTHENTICATION STRATEGY: OIDC TOKEN 28
7.1 IMPLEMENT INGRESS CONTROLLER . . . 28
7.2 CERT MANAGER . . . 28
7.3 SETUP AND CONFIGURE KEYCLOAK . . . 28
7.3.1 DEPLOY KEYCLOAK . . . 28
7.3.2 CONNECT KEYCLOAK TO ACTIVE DIRECTORY . . . 29
7.3.3 CONFIGURE MAPPER FOR EMAIL VERIFICATION . . . 30
7.3.4 CREATE AN OIDC CLIENT . . . 31
7.3.5 DEPLOY AND CONFIGURE GANGWAY . . . 32
7.3.6 CONFIGURE API SERVER . . . 37

References 38

List of Figures 39

Listing 40

1 INTRODUCTION

By emerging the containerized applications, many companies move to the containerized model. They leverage all of the benefits that this model provides, like fast and easy deployment. The container engine is responsible for running the containerized applications. However, it does not provide us features to manage containers automatically, like the situation in which a container needs to be replaced due to the failure. To answer these shortages, The Container Orchestrators tools are introduced which one of the best of these tools is Kubernetes.

In the abstract, to access the cluster, a user needs to be authenticated and authorized. For these purposes, we should implement Authentication and Authorization strategies that are handled by plugins. In the Kubernetes, there is an assumption that the number of users who access the cluster is not too much and can be small. Therefore,there is no actual user object or built-in user manager in the Kubernetes and the users should be handled outside of the cluster. It cannot be effortless and challenging If we have so many users with different usage patterns who need access to the cluster. It also becomes more complicated in the situation in which company policies and regulations should be applied.

To make user management more straightforward and efficient, we introduce a new tool *Christis* that mainly focuses on implementing policies and regulations in big companies or universities which many users should be managed. The Christis assign the concept of Christis Role to the user and trace this assignment. This Christis Role can be anything possible to be deployed in a cluster, like PodSecurity, Role. This report mainly focuses on this tool and its features.

2 BACKGROUND AND OBJECTIVES

2.1 BACKGROUND

In this section, we try to describe all the background knowledge and components needed to run and use **Christis CLI**.

2.1.1 KUBERNETES

At this time, Kubernetes and containerization technologies are viral. Many companies use them in order to deliver their software or products. As shown in Figure 1 before moving to the container deployment, we had traditional and virtualized deployment.

In the traditional model, all applications run on a physical server. There is no boundary between these applications, which causes some problems; for instance, one application could take all server resources so that the other application will face starvation. From a security perspective, one application can access other application processes, which is a security problem because a security flaw in one application can affect others.

By moving to the virtualized model, most traditional models' problems are solved, like better resource utilization and application isolation. However, this model has its problems; for instance, a *Virtual Machine* (VM) still has its *Operating System* (OS), which first needs management and second puts loads on the server. A VM is not portable and lightweight from a software development perspective; therefore, by using VM software development life cycle is still complicated.

In the containerized technologies, some techniques let containers be isolated and at the same time use one underlying OS. A container is the same as a VM but much more lightweight; for instance, it could be around 100Mb. The containers are portable thanks to the container image concept. In a classic scenario, one application can be a container image that can be is build from a container image file (a text document that describes all commands in order to build an image). The central part of containerized technologies is the container engine which manages and runs containers.

In a production environment, we can run many applications in container form. It could reach more than 100 containers, therefore managing and maintaining these numbers of containers are very complex. The container engine only gives us options to run and manage containers, but it does not offer automation tasks like replacing containers in case of failure or scale containers to meet our traffic load needs. A new system was introduced to answer these needs, which makes container and application deployment, management, and scaling automatically. These systems are called Container Orchestration systems, and Kubernetes is one of the most popular container

orchestrators. Kubernetes also helps to span container deployment to more than one server, making applications more highly available and fault tolerance. [21t]

Kubernetes is a very complex system that we cannot cover all of its concepts in this report. In the next chapter, we suppose that you have basic knowledge of Kubernetes and how to use it.

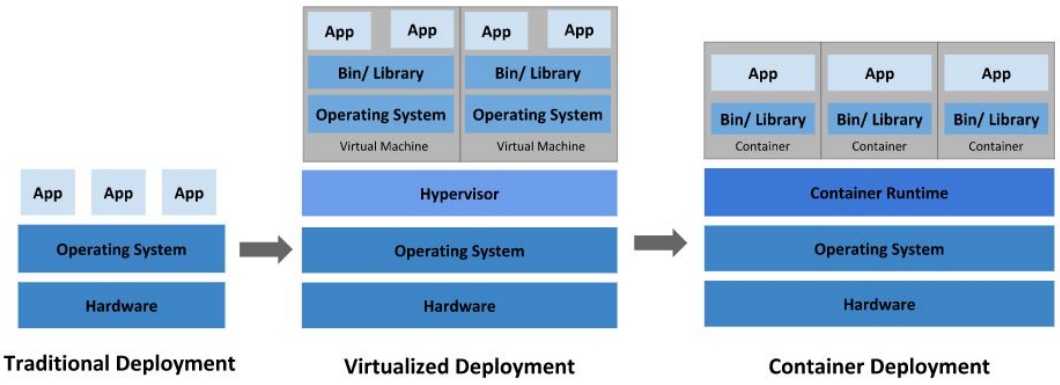

FIGURE 1: *Deployment Eras [21t]*

2.1.2 KUBERNETES CLUSTER COMPONENTS

Kubernetes cluster consists of some components. Some nodes are responsible for running containerized applications. The central part of the Kubernetes cluster is the control plane that itself consists of some components. The Control plane provides all Kubernetes capabilities that we know. In the figure 2 a Kubernetes cluster and its cluster can be seen. [21l]

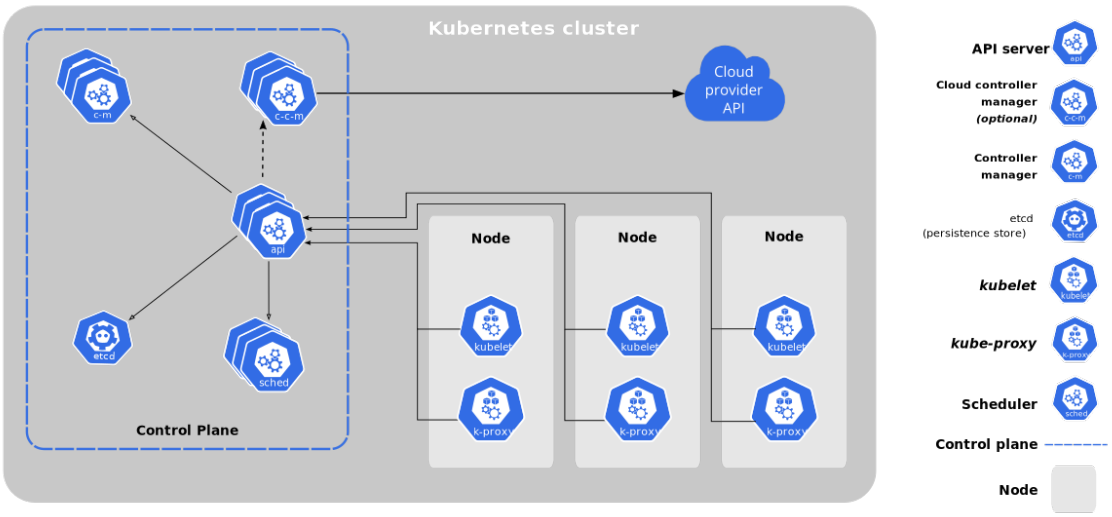

FIGURE 2: *Kubernetes cluster components [21l]*

The API Server is the component that we interact with, and it is the central part that we can talk to the cluster via it. When we run a command to run a Pod, we are interacting with the API Server. The API Server is responsible for Authentication and Authorization.

2.1.3 KUBERNETES AUTHENTICATION AND AUTHORIZATION

In the Kubernetes, we have two kinds of users, **Normal users** and **Service account**. The Service account is managed by Kubernetes and is used by the containerized application, so we do not focus on it in this report.

When we run a command against the Kubernetes cluster and talk to API Server, we are the Normal users. Kubernetes assumes that other services are managing the Normal users outside of the cluster. Therefore, it does not manage the Normal users. There is no object for Normal users in the Kubernetes and It is not possible to add Normal users to the cluster. [21j]

The examples of other services that can manage user could be:

- In the simple case: List of username and password file
- In the complex case: Social network accounts like Google, Facebook
- In the insecure case: Distribute private keys
- In the optimal and complex case: Use LDAP to manage users

This report will focus on the last option, and we will use LDAP to store and manage our users.

When we perform a command against the Kubernetes cluster, our request will be sent to the API Server, and our request will be authenticated via an authentication plugin. Kubernetes uses client certificates, bearer tokens, an authenticating proxy, or HTTP basic auth to authenticate API requests through authentication plugins. [21j]

One of the ways to authenticate requests and users is using **OpenID Connect Tokens**. OpenID Connect is a superficial identity layer on top of the OAuth 2.0 protocol. It allows clients (here is our Kubernetes cluster) to verify the end user's identity based on the authentication performed by an Authorization Server and obtain necessary profile information about the end-user. [21p]

To make it easy to understand, the figure 3 can be useful.

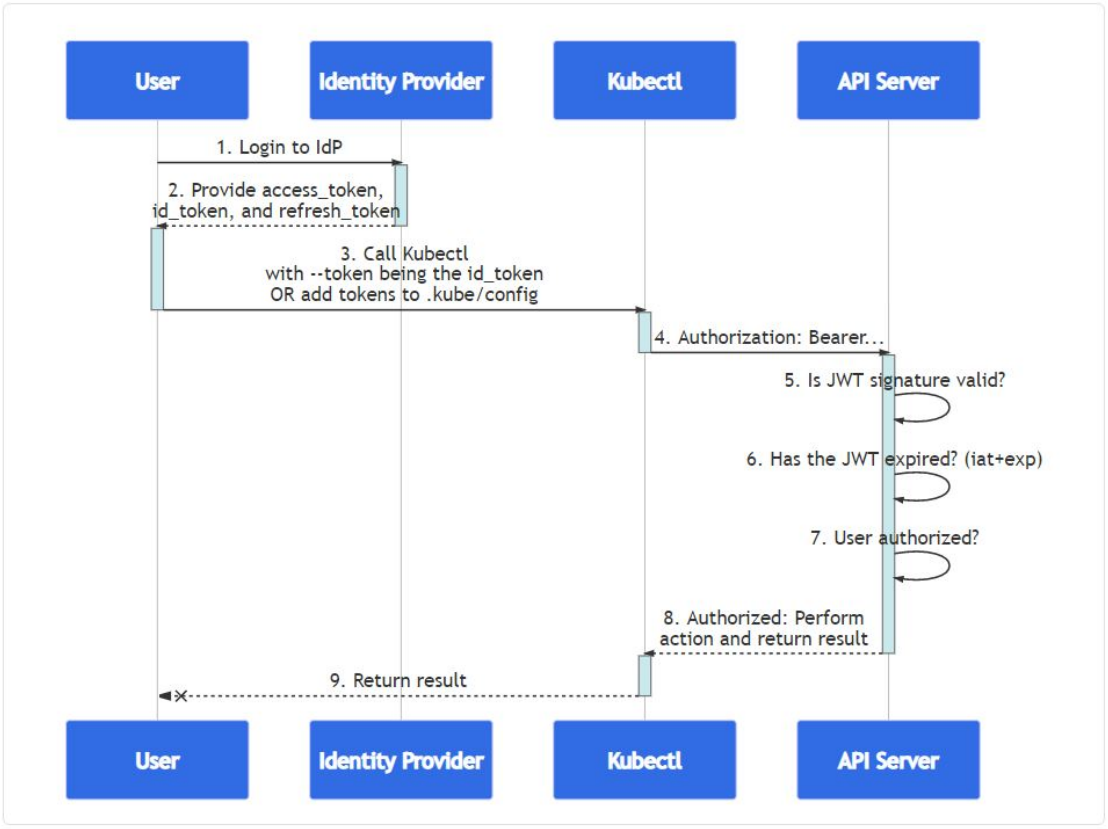

FIGURE 3: *OIDC Authentication [21j]*

We do not want to dig into the details because our solution **Christis** can be worked by any authentication strategies. However, we use OpenID Connect to authenticate our users in our scenario, so it is worth the procedure to make configuration understandable.

The first user needs to login into the Identity Provider. In our scenario, we will use **Keycloak** as an Identity Provider. After successful login and authentication, the Identity provider gives the user a set of tokens that can be used by *Kubectl* to send to the API Server to be authenticated. The API Server checks the token's validity. If it is okay, the user will be authenticated, and in the final step, the user's request is checked against the Authorization plugin to know that the user is allowed to make the request or not. Then the API Server sends the result of the request to the user. In this diagram, there are no relations between API Server and Identity provider, but behind the scenes, there is trust between API Server and Identity provider that let the API Server trust generated tokens by Identity provider.[21j]

After successful authentication, the API Server will authorize user requests. It evaluates all requests' attributes against all policies to determine that the user is allowed to make that request or not. There are two attributes in the request that is very important for us, the **user** and **group**. The user attribute shows who is the user and which groups the user belongs to. These two attributes will be used to define our policies; for instance, users who are in the admin groups can be cluster

admin[21k]. Due to the complexity and details in the requests' attributes, we cannot go into the detail, but the reference can be checked for more information.

API Server uses one of the available authorization modes to authorize a user request. The recommended and most standard mode is *Role-based access control* (RBAC) which gives access to the user based on the role that a user can assume. For instance, user *Max* had a cluster-admin role which means he can manage the cluster without any obligation. In this mode, first, we create our role (Role or ClusterRole) and then use binding (RoleBinding or ClusterRoleBinding) to specify who can assume the role [21k]. In the following chapters, we will use this mode for our scenario, but the **Christis** can be used for other modes.

2.1.4 HELM

In simple terms, Helm is a package manager for Kubernetes like APT or YUM package manager for the Linux distributions. Helm helps us to provision complex Kubernetes applications or scenarios in a straightforward way. Helm uses a concept of Chart that can be considered as multiple Kubernetes manifest files in a template format. A Chart can be an entity that we will use to deploy our scenario in the Kubernetes cluster. The Chart can be easily created, versioned, shared, and published. [21r]

To make everything clear, The diagram of Figure 4 can be seen. The Chart is like a template of our scenario and application that needs to be converted to the real Kubernetes manifest files. To do that, we specify all the values that are needed for this conversion (In our solutions, the values are provided by **Christis**). Helm uses Chart and values to generate Kubernetes manifest file and deploy our application or scenario in the cluster as a Release.

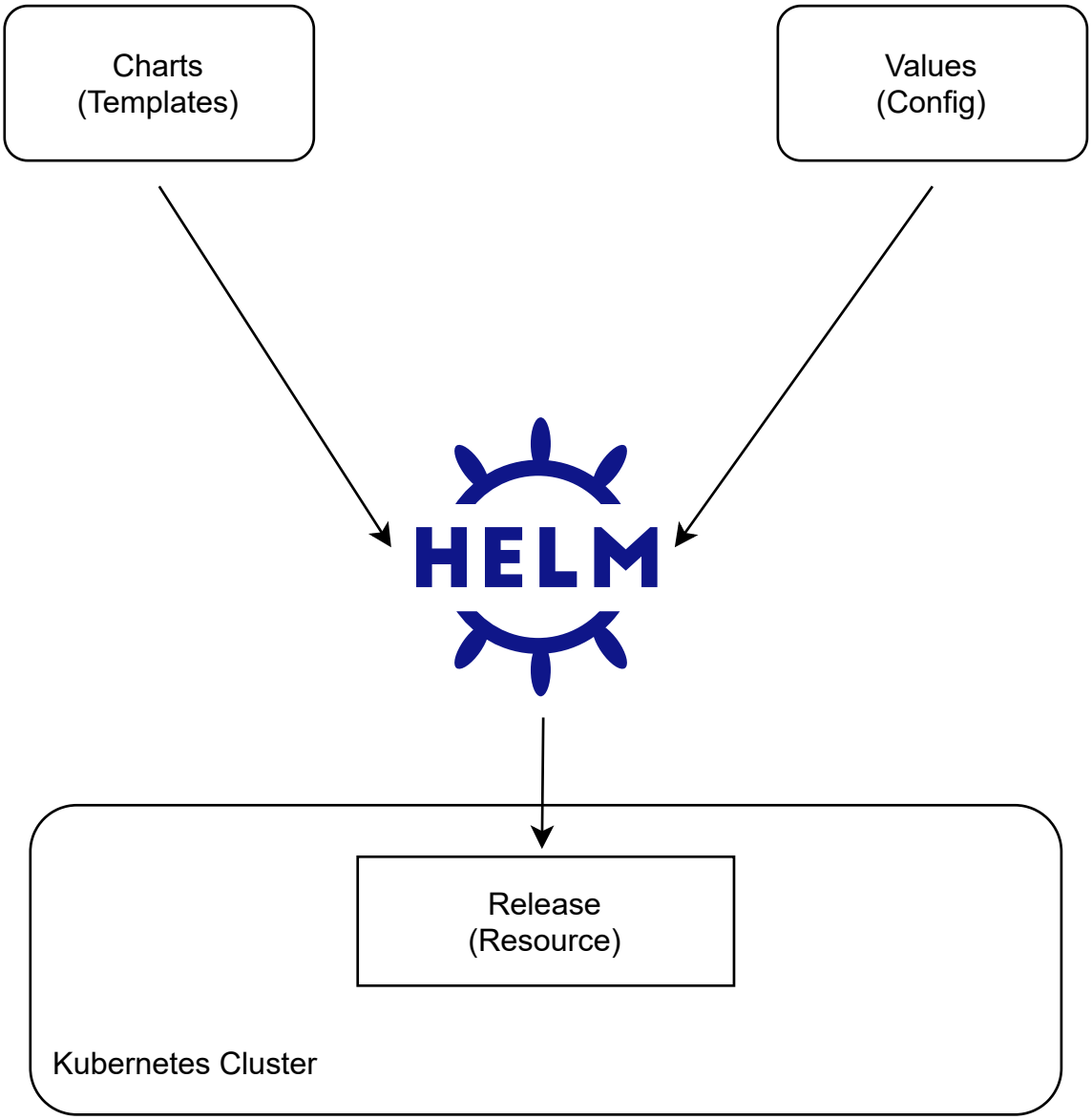

FIGURE 4: *Helm in abstract*

The Helm is a complex tool that we cannot go into detail in this report, but to get familiar with it, this website can be checked.[21g]

2.1.5 KEYCLOAK

Keycloak is an Identity and Access Management solution which helps securing applications and services. According to our scenario for authentication in the Kubernetes, the Keycloack has the Identity Provider's role.[21s] Keycloack has many features, but we will use two of them in our scenario. The first one is **User Federation** which gives us an option to connect existing LDAP or Active Directory servers to Keycloack, so the users who need to access the cluster are LDAP or Active Directory users. The second one is **Client Adapters**, which gives us the ability to register

our API Server as an OpenID Connect client to authenticate the user and verify tokens generated by Keycloak. [21s]

To have better knowledge about Keycloak, this documentation can be checked. [21i]

2.2 OBJECTIVES

This report has the following objectives:

- Why the Christis has been designed and how it can help
- The design and components of Christis
- Deploy Christis and its components
- Implement authentication strategy for a Kubernetes cluster
- Explain and implement a real scenario with the help of Christis

3 CHRISTIS CLI DESIGN AND COMPONENTS

In this section, we first explain the problems that inspired us to think of design **Christis CLI**, Then we move to the actual Christis design and the components that we used.

3.1 THE REASON BEHIND THE CHRISTIS

We have a user story that motivates us to design Christis. In our university Computer Science IV group, one Kubernetes cluster is needed to be provisioned and implemented. This cluster will be used by students and people who are work like supervisors. Some requirements should be fulfilled for this cluster.

- Users will use the cluster for a different purpose; for instance, one user needs to have only one Pod, whereas others might need their Namespace
- The number of users who need access to the cluster is not small, and they have different access types and roles
- There should be some policies to make the clusters secure, and policies can be different based on the user role
- The resource consumption by each user should be controlled and follows the policies
- Implementing a user-friendly Authentication and Authorization strategies

If you have used Kubernetes before, you noticed that from Kubernetes perspective, the cluster would be used by a small number of trusted users and mostly with the same needs, accesses, and purposes. However, if we need to manage a large group of users with different needs, accesses, and purposes, it could be complicated and error-prone for administrators. Kubernetes gives us some features like Role, PodSecurity, ResourceQuota, LimitRange, and NetworkPolicy to implement our policies, and it is challenging and complex to manage all of them. We designed the Christis to make these tasks easy and it will let us implement a secure cluster that follows defined policies and regulations.

The Christis is useful for any scenario that has the same requirements that we mentioned. The Christis also can be used to deploy any predefined and pre-configured Pod for users and make everything much easier for users.

3.2 DESIGN AND COMPONENTS

In this chapter, we will focus on the design and components of Christis.

3.2.1 DESIGN IN ABSTRACT

In the Christis, We consider two concepts in our design. **User** and **Role**. The Role is different from Kubernetes Role; in our design, it is an abstract term.

User is a concept that specifies the actual users who need to access the cluster, and we need their attributes to create our Role and deploy Roles for them. In our implementation until now, the actual users are coming and syncing from the Active Directory, then we store users and their attributes into Christis Database. So until now, we have the users and their attributes. Attributes that we extract from users and store in the Database are:

- Email
- Common Name (CN)
- Distinguished Name (DN)

Moves to the Role, Role is a Helm Chart that we can create, and it can be a combination of any policies like Role, PodSecurity, ResourceQuota, LimitRange, and NetworkPolicy.Even the role can contain the actual application deployments. The user attributes mentioned above can be used in the role design, and consider that these attributes will be provided by Christis, which means they do not need to be specified precisely in the Chart by the administrator. If we categorize users based on their needs, policies, and purposes, the role is an entity that can implement them for each category. For instance, we can have the role for the students, supervisors, and professors. The role will be stored in the Christis Database.

By using Christis, we can assign a role to the user, and this assignment is captured into the Database. Each user can be assigned to more than one role, which gives the administrator flexibility to design the role.

To clarify the whole procedure, the diagram in Figure 5 can be helpful. In the next chapter, we will go into more details about Christis.

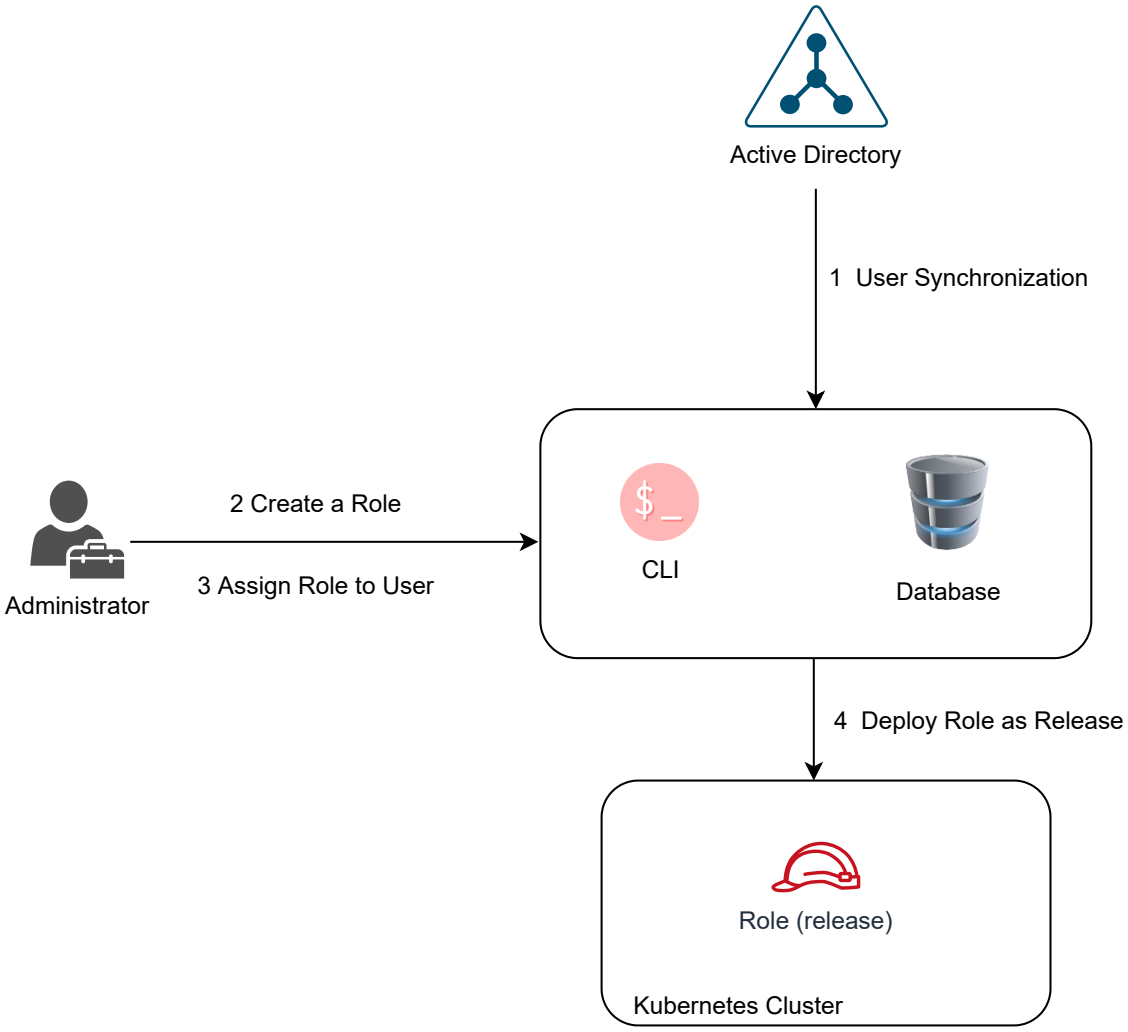

FIGURE 5: *Christis in abstract*

3.2.2 COMPONENTS

The Christis consists of some components. These components are:

- Christis API
- Christis Database
- Christis CLI
- Helm

As we mentioned in the previous sections, the Active Directory users and their attributes are synced to the Christis Database. The **ChristisAPI** is a component that connects to the Active Directory and provides a Restful API that Christis CLI will use to get information about users and their attributes. The Christis API is run in a container.

Christis Database is a NoSQL database that Christis CLI uses to store information that are needed, like user information, roles, and role assignments. In our design, we consider MongoDB as NoSQL database.

Christis CLI is the central part of Christis that the administrator interacts with to create the role, sync user, and assign the role.

Helm is the heart of Christis CLI responsible for deploying a role after the administrator assigns roles to users.

3.2.3 DESIGN IN DETAIL AND HOW CHRISTIS WORKS

In our design, we consider that the Christis only focuses on a specif group in Active Directory and only syncs users from that group. We called this group **K8sGroup** in the rest of the report.

When the Christis CLI starts the user's synchronization process, It asks Christis API to provide it the users who are the member of K8sGroup. Christis API connects to the Active Directory, gets users' information, and sends it to the Christis CLI. After that, Christis CLI adds those users to a specific table in the Christis Database called **Stage**. The Stage table is the exact capture of K8sGroup at any time, which means removing a user from K8sGroup; the user will also be removed from the Stage table. Figure 6 shows all the explained steps in a diagram.

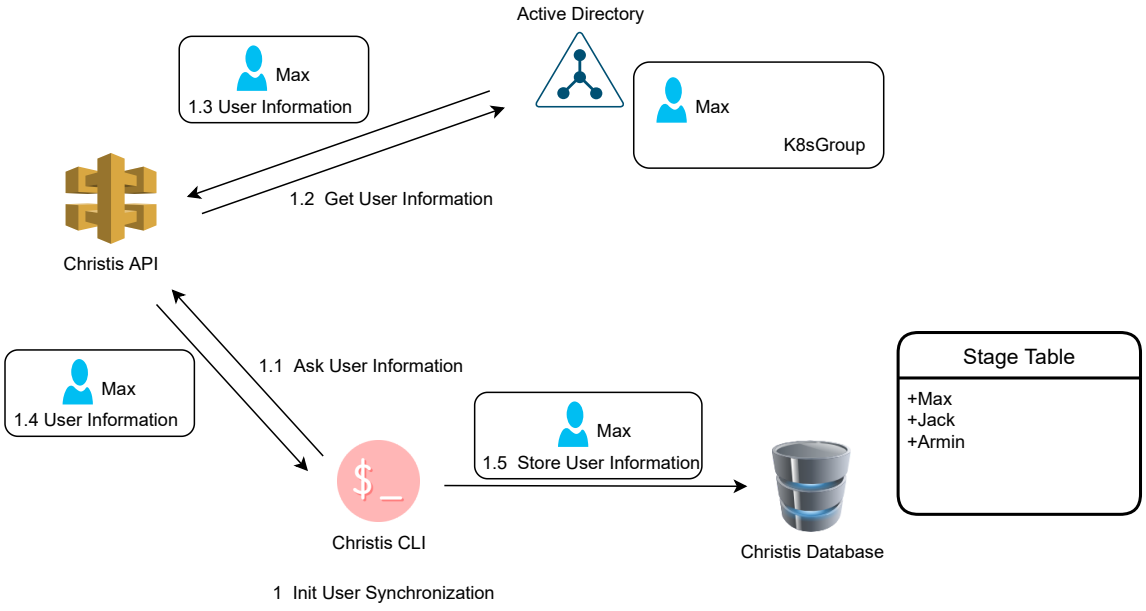

Figure 6: *User Synchronization*

The next most crucial part is how to add a **role**. First, we need to create a Helm chart as the role, and all Kubernetes features can be used in the role. It should be considered that the user attributes such as*Email*, *Common Name*, *Distinguished Name*, *K8sGroup Name*, *Release Name*, *ID* can be used as variables in the chart template because Christis provide them during role assignment. (In the next version of Christis, more attributes will be supported). After we use Christis CLI to add our role

into the database in the **Role** table, to add a role to the database, two mandatory role attributes should be defined which are **Name** and **Version**. Figure 7 is depicted the procedures.

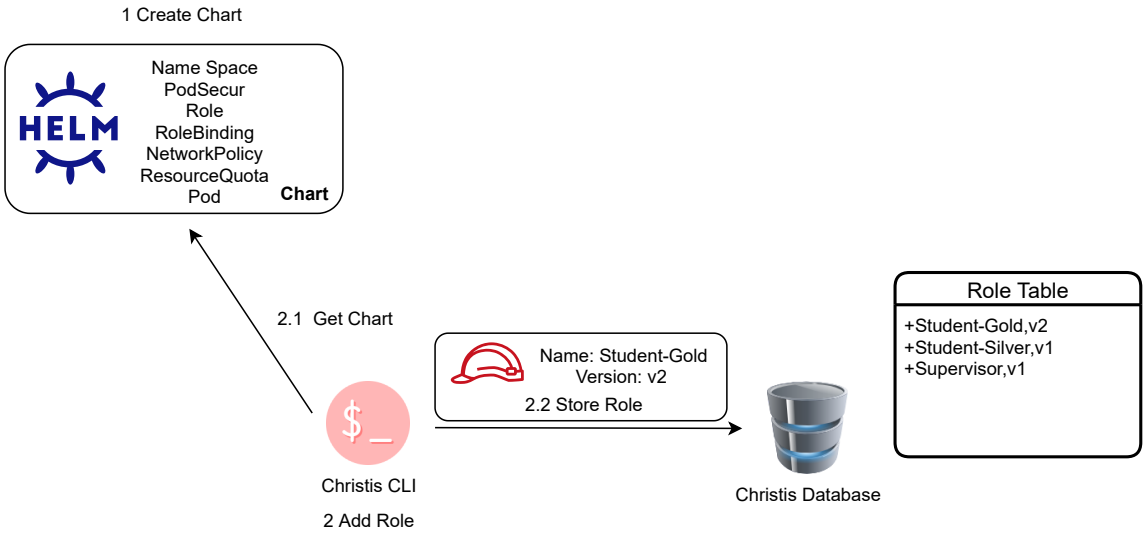

Figure 7: *Role creation and add*

The user who is in the Stage table is ready to get roles. To assign a role, we will use Christis CLI and specify the user who needs role to and the role name and version. During the role assignment, Christis CLI deploys or installs the role as Helm release with the provided user attributes as variables. After assigning role, one entry related to this assignment will be created into a specific table called **Main**. A user can have more than one role, and also, if we have a role with two versions, both of these two versions can be assigned to the user. Figure 8 shows a whole idea behind the role assignment.

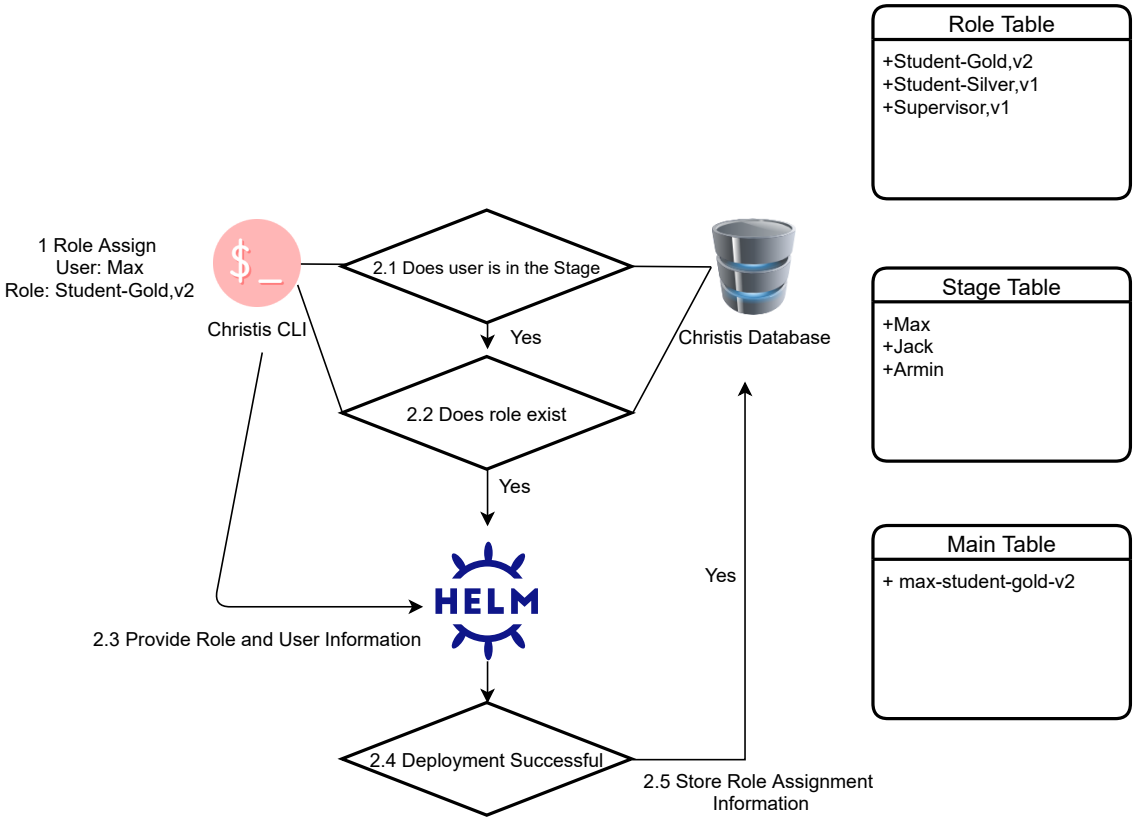

Figure 8: *Role assignment*

As an administrator, we can unassign a role from the user, so Christis CLI uses Helm to remove the role release and removes the role assignment from the Main table.

There is a scenario that we have a role assignment for a user, and after a while, the user is removed from the K8sGroup. For handling this situation, we consider that the Christis CLI preserves the role assignment, which means nothing will be removed. However, the users move from the Stage table to the specific table called **Isolated Users**, which means the user cannot get any new role. We give the administrator *purge* option to remove all roles of the user who is in the Isolated Users table. If the user will be added to K8sGroup, it is moved back to the Stage table and can get the role again. Figure 9 shows all details to make everything clear.

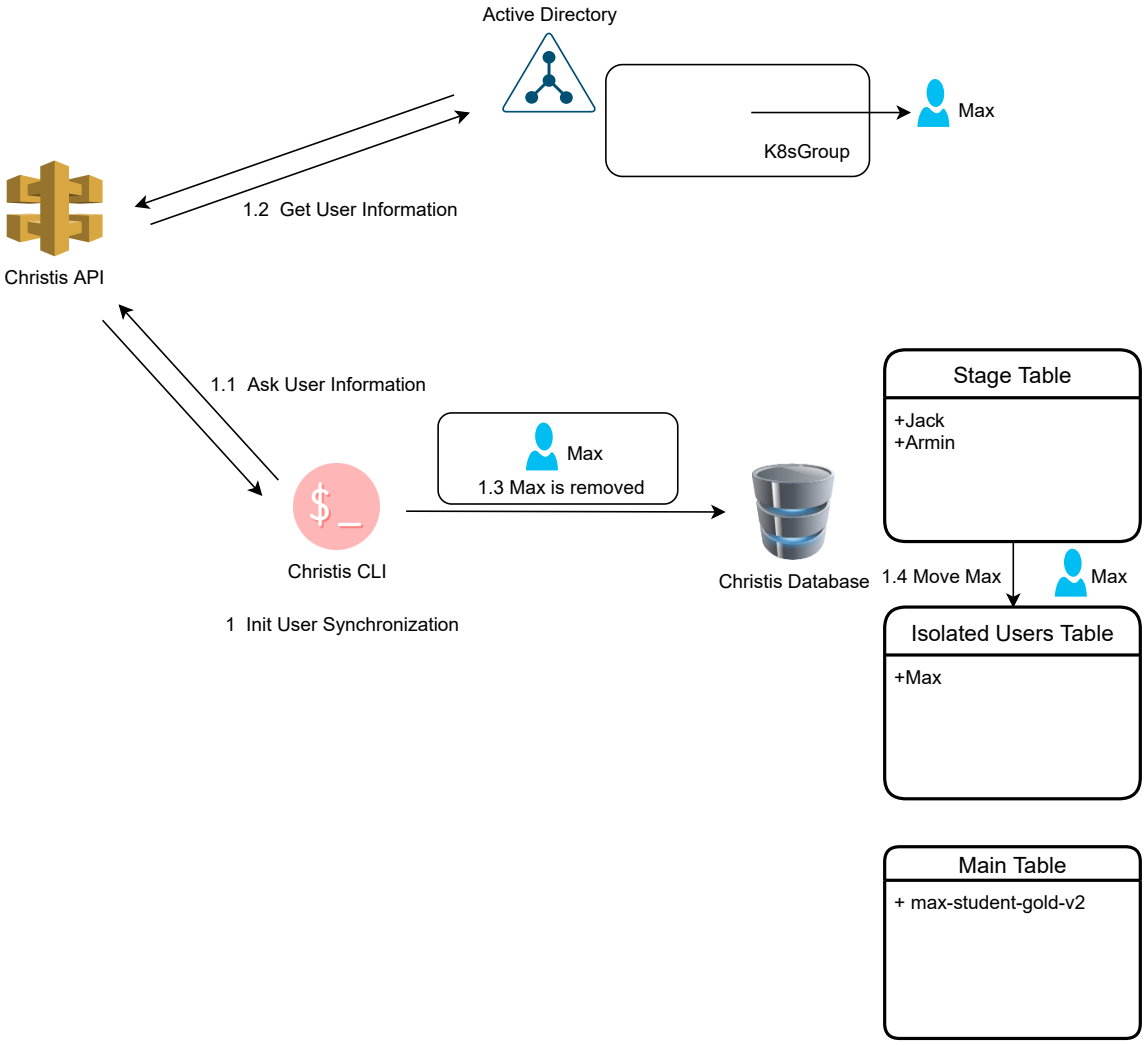

FIGURE 9: *Removing an user from K8sGroup*

4 HOW TO DEPLOY AND RUN CHRISTIS

In this section, we will focus on the Christis requirements and how to deploy Christis.

4.1 Christis Requirements

To use and deploy the Christis, some requirements should be fulfilled that we explain in this part.

1. Kubernetes cluster should be ready
2. The **Kubectl** should be configured (As Cluster Admin user) for the user that needs to use Christis. We recommend that setup a Jump Host for only Christis
3. Deploy a **MongoDB** as Christis Database. Database user needs to be able to create Database and Collections
4. Christis API is a Containerized application which means it should be deployed in the cluster or outside of it
5. The **Helm** should be installed on the Jump host
6. The Active Directory and users who access to the users and groups

In the following sections, we will show how to deploy all components and Christis.

4.2 DEPLOY CHRISTIS

First, we will deploy the **Christis CLI** and move to deploy requirements because we need to use Christis CLI to generate some configurations for components.

To install the Christis, we need to use the *pip* as can be seen in the following command:

```
1 pip install christis
```

Listing 4.1: *Install Christis*

4.3 DEPLOY CHRISTIS COMPONENTS

Before using the Christis CLI, the requirements should be deployed and ready to work.

4.3.1 CHRISTIS DATABASE

The first component that should be deployed is **Christis Database**. As we mentioned, MongoDB must be used. In order to make our setup simple, we use a simple Docker command to setup the Database.

```
docker run -d --name mongo \
-e MONGO_INITDB_ROOT_USERNAME=mongoadmin \
-e MONGO_INITDB_ROOT_PASSWORD=secret \
-p 27017:27017 \
-v $(pwd)/mongo-data:/data/db \
mongo
```

Listing 4.2: *Deploy Christis Database*

> **Note:**
> The Database must be accessible from port 27017. In the next version of Christis, we provide an option to make it possible to change the port too.

> **Note:**
> The Root username, password, and the MongoDB database's IP are needed to configure the Christis CLI. For more information related to MongoDB options,this documentation can be checked. [21m]

> **Note:**
> The provided deployment is very simple. In the Christis wiki in our repository, we will provide more realistic deployment. [21c]

The Christis CLI needs to know the information related to how to access the Christis Database. The Christis CLI has a configuration generator that can be used to generate all configurations that are needed. The Christis CLI needs a configuration file to know how to access the Christis Database. This command can generate this configuration file:

```
christis config database --help #get a help related to options
christis config generate database \
 --mongodb-address localhost \
 --mongodb-user mongoadmin \
 --mongodb-password secret
```

Listing 4.3: *Generate a Configuration file for Christis CLI to access Database*

> **Note:**
> Do not copy or move the generated file; this file will be located in the right place.

4.3.2 CHRISTIS API

The next component is **Chrisits API**, to make it simple, we use a simple Docker deployment:

```
docker run --name christisapi \
-p 5000:5000 \
```

```
--env CHRISTIS_CONFIG_FILE=/opt/k8sChristis/config.yaml \
-v /opt/k8sChristis/config.yaml:/opt/k8sChristis/config.yaml heiran/christisapi
```

Listing 4.4: *Deploy Christis API*

The ChristisAPI needs a configuration file to connect to the Active Directory and query users information. The *CHRISTIS_CONFIG_FILE* variable specifies that where API can find the configuration file. A sample configuration file can be seen in following:

```
ldap:
  ldap_base_dn: dc=informatik,dc=local
  ldap_k8s_group: k8s
  ldap_password_connector: Abc12345678@@
  ldap_server: ldap://192.168.23.133
  ldap_user_connector: CN=Armin,CN=Users,DC=informatik,DC=local
```

Listing 4.5: *Christis API configuration file*

> **Note:**
> It is not recommended to create this file manually, and it is recommended to use Christis CLI to generate it.

This Configuration file can be created by **Christis CLI** too. To do that, we can use this command:

```
christis config generate christis-api --help #get a help related to options
christis config generate christis-api \
--ldap-server-address LDAP://192.168.23.133 \
--ldap-user CN=Armin,CN=Users,DC=informatik,DC=local \
--ldap-password Abc12345678@@ \=
--ldap-base-dn dc=informatik,dc=local \
--ldap-k8s-group k8s
```

Listing 4.6: *Use Christis to generate the ChristisAPI configuration file*

This command save the configuration file in this location: `$HOME\.christis\ldap.yaml`. This configuration file can be copied and used for running the Christis API.

4.3.3 CONFIGURE CHRISTIS CLI TO ACCESS CHRISTIS API

After deploying all components, we need to configure the Christis CLI and give it the other necessary information. To do that, we should use the Christis CLI and run the following command:

```
christis config generate cli --help #get a help related to options
christis config generate cli \
--api-address 127.0.0.1 \
--api-port 5000 \
--k8s-group k8s
```

Listing 4.7: *Configure Christis CLI*

api-address is the address of Christis API and the *api-port* is its port. *k8s-group* is the group name of K8sGroup.

> **Note:**
> Do not copy or move generated file; this file will be located in the right place.

5 CHRISTIS IN A REAL SCENARIO

In this chapter, we will focus on a real scenario and describe a user story to how we can use Christis CLI.

> **Note:**
> The Christis functionality will be complete if we have an authentication strategy. In the Appendix, we will implement the OIDC Token authentication strategy and use it for the rest of the chapter.

First, we describe a real scenario and use Christis to reach our goals. At the end of this chapter, we know about:

- How to sync user between Active Directory and Christis
- How to create a Role
- How to assign a Role
- How to unassign a Role

5.0.1 SCENARIO

Our user story is straightforward because we want to focus on Christis functionality, not Kubernetes features. In our repository, we will have more advanced scenarios [21c]. In this scenario, we have one user Max that needs access to our cluster. This user will need admin access to our cluster. In the end, we use Gangway and Keycloak to authenticate our user and access cluster.

5.0.2 IMPLEMENTATION

In this section, we focus on Christis and implementation.

USER SYNCHRONIZATION

In our Active Directory, we have a group called *k8s*, this is our *K8sGroup* that users who are members of this group can access to the cluster, and we can assign roles to them.

As can be seen in Figure 10 we add our user Max into the *K8sGroup* k8s.

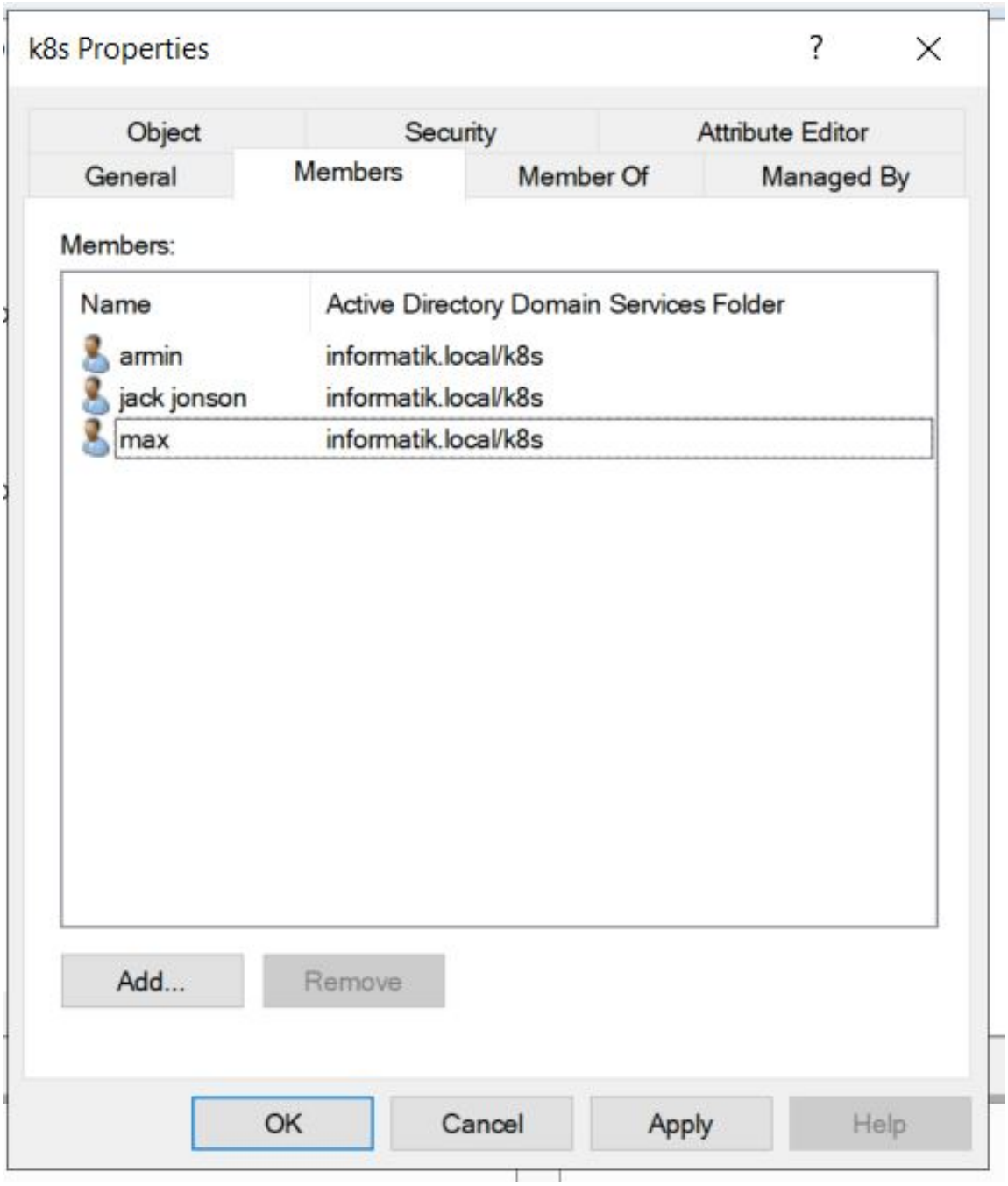

Figure 10: *Add user to K8sGroup*

All the commands and sub-commands have a help that can be accessible by *–help* option, as can be seen in the Figure 11 we checked the Database command help to find out which sub-command we should use to sync our users.

```
> christis database --help
Usage: main.py database [OPTIONS] COMMAND [ARGS]...

  Commands relate to databsae sync and view tables

Options:
  --help  Show this message and exit.

Commands:
  sync  Sync users from K8sGroup in your Active Directory to your Christis...
  view  Commands for viewing tables like stage,main,...
```

Figure 11: *Database help command*

So we should use *Sync* command to sync our user, as can be seen in the Figure 12 we use this command, and *Max* is synced from the Active Directory to our Database (in the Stage table), and we can assign the role to him.

```
> christis database sync
No user was deleted from k8s group. No candidate to delete!
The following users will be added to the Stage :
User 1 >>>  Email : max@informatik.local | DN : CN=max,OU=k8s,DC=informatik,DC=local
```

Figure 12: *Database Sync*

> **Note:**
> Christis CLI will ignore users who have no specified email in their attributes in Active Directory to sync

> **Note:**
> The *Sync* command can be defined as *Cron Job* to have periodic synchronization

CREATE A ROLE

In this part, we create a role that assigns our users the Kubernetes role of *ClusterAdmin*. First, we need to create a Helm chart for this purpose. As shown in the Figure 13, we can use *helm create* command to create a chart.

```
> helm create cluster-admin
Creating cluster-admin
rmin@Home-PC:~/Helm-Role$ ls
admin  bronze  cluster-admin  gold  silver
rmin@Home-PC:~/Helm-Role$
```

Figure 13: *Create a Chart*

Note:
Remove unnecessary files from the chart. We remove all the files in the template directory.

We create a template for assigning our cluster-admin role to the user, and this template can be seen in the Figure 14. This template binds the ClusterAdmin role to the user.

```yaml
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: {{ .Values.user_cn }}-admin
subjects:
- kind: User
  name: {{ .Values.user_email }}
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

Figure 14: *Role binding Template*

In the subject, we provide the user email as the user identifier because we configure our Authentication strategy to use *Email* as the username of a user. The value of variables.Values.user_cn and .Values.user_email will be provided by Christis CLI, and we do not need to worry about them.

We can use these variables that describe below in our Chart and make sure that they are provided by Christis CLI.

- .Release.Name : It is the combination of some values <K8sGroup>-<User Common Name>-<User Domain Part>-<Role Name>-<Role Version>
- .Values.user_id : an unique id for user
- .Values.user_cn : User Common Name (CN)
- .Values.user_dn : User Distinguished Name (DN)
- .Values.user_email : User Email Address
- .Values.user_k8_group : The K8sGroup Name

Now we use the *role create* command to add our Chart as Christis Role. Figure 15 is showing this command. As can be seen, the linting is successful, which means the role is added.

```
> christis role create --name clusteradmin --version final --location /home/rmin/Helm-Role/cluster-admin --description "An AdminClusterRole"
Role's linting is sucessful.
```

FIGURE 15: *Create Role*

> **NOTE:**
> Helm Chart can also be in the package form (.tgz). The Christis CLI is linting the Chart, and if there is an error, it does not add role.

> **NOTE:**
> Role name and version must not contain the asterisks "*" and must be in lowercase form.

We can see all of our available roles by using *role view* command that can be seen in the Figure 16.

```
> christis role view
+--------------+-----------------------------------+---------+
|  Role Name   |              Location             | Version |
+--------------+-----------------------------------+---------+
|     gold     |   /home/rmin/Helm-Role/gold/gold  |  alfa   |
|     beta     | /home/rmin/Helm-Role/silver/silver|  beta   |
|    bronze    | /home/rmin/Helm-Role/bronze/bronze|  final  |
|    admin     |  /home/rmin/Helm-Role/admin/admin |  beta   |
| clusteradmin | /home/rmin/Helm-Role/cluster-admin|  final  |
+--------------+-----------------------------------+---------+
```

FIGURE 16: *View Role*

ASSIGN A ROLE TO USER

Before we assign our role to the user Max, we can check the users in the Stage table with the command of *user view stage*. The users that can be seen in the Figure 17 can be assigned roles.

```
> christis user view stage -vv
+------------------------+-------------+-----------------------------------------------+
|         Emails         |   User CN   |                    User DN                    |
+------------------------+-------------+-----------------------------------------------+
| jonson@informatik.local | jack jonson | CN=jack jonson,OU=k8s,DC=informatik,DC=local |
| armin@informatik.local  |    armin    |    CN=armin,OU=k8s,DC=informatik,DC=local    |
|  max@informatik.local   |     max     |     CN=max,OU=k8s,DC=informatik,DC=local     |
+------------------------+-------------+-----------------------------------------------+
```

FIGURE 17: *View Stage table*

Now we want to assign *ClusterAdmin* role to the user with command of *user role assign*. This command needs to specify whom we want to find our user, and our options are email, user CN, and user DN. Here we use email. The *role* options should get the combination of *role name* and *role version* that are separated by an asterisks("*"). The role assignment command can be seen in the Figure 18

```
> christis user role assign --attribute-value max@informatik.local \
> --attribute-type email \
> --role clusteradmin*final
Assigning Role to the user: email=max@informatik.local ...
ReleaseName : k8s-max-informatik-local-clusteradmin-final
The role clusteradmin with the version of final is assigned.
Processing  [####################################]  100%
```

Figure 18: *Role assignment*

Now we can check the role assignment with the command of *user role view*, as can be shown in the Figure 19, the role of *Cluster Admin* is assigned to Max.

```
> christis user role view --attribute-value max@informatik.local \
> --attribute-type email
+--------------------+---------+-------------------------------------+--------------+--------------+-----------------------------------------+
|     User Email     | User CN |              User DN               |  Role Name   | Role Version |              Release Name               |
+--------------------+---------+-------------------------------------+--------------+--------------+-----------------------------------------+
| max@informatik.local |   max   | CN=max,OU=k8s,DC=informatik,DC=local | clusteradmin |    final     | k8s-max-informatik-local-clusteradmin-final |
+--------------------+---------+-------------------------------------+--------------+--------------+-----------------------------------------+
```

Figure 19: *Role assignment view*

USE GANGWAY TO AUTHENTICATE

In this part, we will use Gangway that we configured to test our role assignment and OIDC token Authentication strategy. First, we need to go to the Gangway, as shown in the Figure 20 and sign in.

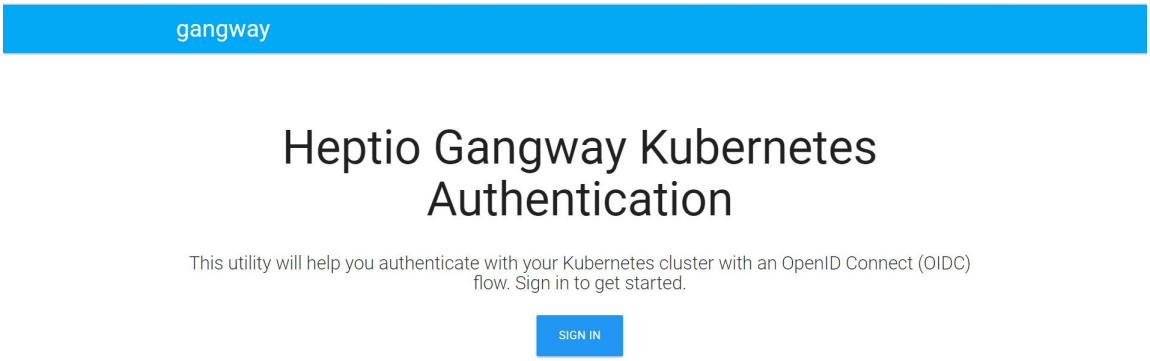

Figure 20: *Gangway Sign-in*

The Gangway will redirect us to the Keycloak to be authenticated, and we need to log in with our Active Directory credential (In our configuration, we used *SamAccountName* as the username. However, it can change to email) as can be seen in the Figure 21.

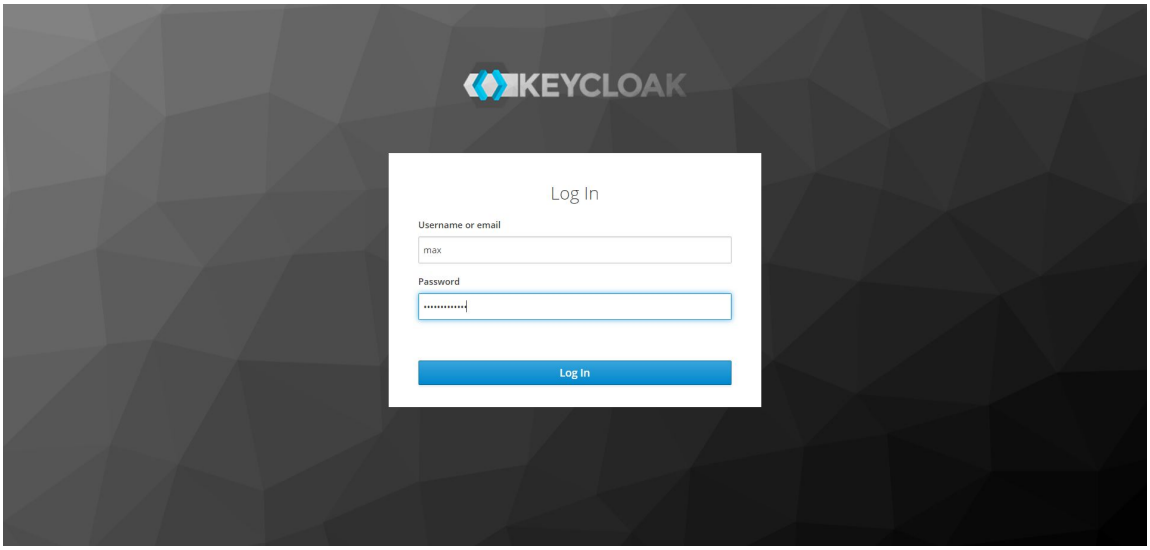

Figure 21: *Authenticate in the Keycloak*

After successful authentication, Gangway will show us a page like the Figure 22 that we can download the *Kubectl* configuration, and we should put it in the *.kube* directory in our home.

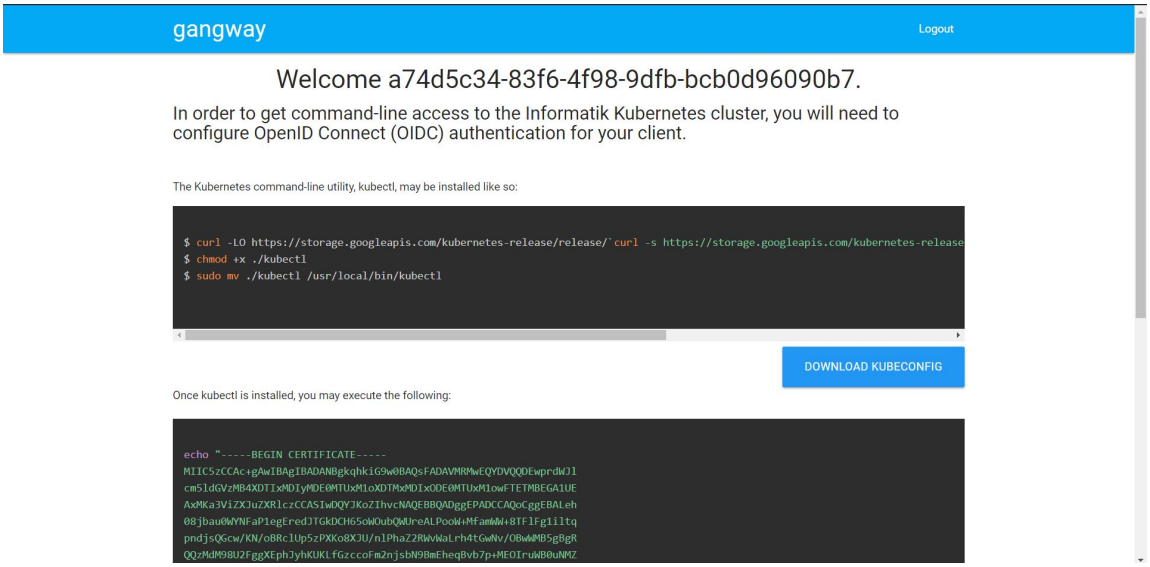

Figure 22: *Get Kubectl configuration from Gangway*

Now our users are authenticated, and because we assigned him a role, he will be authorized as Cluster Admin as can be seen in the Figure 23. Now we implement both Authentication and Authorization strategies with the help of Christis and other services.

```
> kubectl get nodes
NAME     STATUS   ROLES                  AGE   VERSION
master   Ready    control-plane,master   16d   v1.20.4
worker   Ready    <none>                 15d   v1.20.4
```

Figure 23: *Test user role assignment and authentication strategy*

UNASSIGN A ROLE

In this part, we want to unassign the role of *Cluster Admin* from our user Max. We should use command *user role unassign*, and the result of this command can be seen in the Figure 24.

```
> christis user role unassign \
> --attribute-value max@informatik.local \
> --attribute-type email \
> --role clusteradmin*final
The release name : k8s-max-informatik-local-clusteradmin-final
The role clusteradmin with the version of final is unassigned.
```

Figure 24: *Unassign Role*

Now we can check the command *kubectl get nodes* and as can be seen in the Figure 25 whereas the user is authenticated but is not authorized to access the cluster.

```
> kubectl get nodes
Error from server (Forbidden): nodes is forbidden: User "max@informatik.local" cannot list resource "nodes" in API group "" at the cluster scope
```

Figure 25: *Test role unassignment*

6 CONCLUSION AND FUTURE WORKS

6.1 CONCLUSION

We introduce a new tool that makes user management very easy and fast in the situation that the number of users is not small like the university scenario that the students need to access to the Kubernetes cluster. Using the Christis tool lets cluster administrators define roles based on company policies and regulation and only used Christis as a centralized tool to assign roles to users. Therefore each admin does not let to create Kubernetes objects manually, which makes tracing easy. In general, Christis helps us have a more secure Kubernetes cluster that follows best practices and compliance.

6.2 FUTURE WORKS

- Implement Christis Tool as API, which makes it easy to create UI for it
- Add features to manage more than one Kubernetes cluster
- Support more than one group as K8sGroup
- Support more user sources like GitLab and Keycloak, not only Active Directory

7 APPENDIX: IMPLEMENT AN AUTHENTICATION STRATEGY: OIDC TOKEN

In this chapter, we will implement the OIDC Authentication strategy for our Kubernetes cluster.

7.1 IMPLEMENT INGRESS CONTROLLER

We recommend deploying an Ingress Controller for the cluster because it will configure access to other services that are needed to implement the Authentication strategy.

In our scenario, we will use the **Nginx Ingress Controller**. In order to get more information, its documentation can be checked. [210]

> **Note:**
> Setup the ingress controller in a Bare-Metal is complicated. We recommend checking Bare-Metal consideration. [21n] We used HAProxy to handle external access to our services. [21f]

7.2 CERT MANAGER

In the following sections, we will deploy two services that provide us the Authentication. It is recommended to deploy them securely and use TLS. Therefore we need a way to provide the certificates. In order to provide the certificate, we use the Cert-Manager. [21b] Cert Manager is a certificate management controller that helps us to issue certificates from various resources such as Let's Encrypt[21a]. A **ClusterIssuer** should be created for cluster.

7.3 SETUP AND CONFIGURE KEYCLOAK

In this section, we configure Keycloak that is **Identity Provider** in our Authentication strategy.

7.3.1 DEPLOY KEYCLOAK

First, we need to set up Keycloak in our cluster to do this documentation can be useful [21q]. For other options and database configuration, this documentation should be checked. [21h]

In our scenario, we use the address of **keycloak2.cloudarmin.de** for our Keycloak and configure our ingress as:

```
1 apiVersion: networking.k8s.io/v1beta1
2 kind: Ingress
```

```
3  metadata:
4    name: keycloak
5    annotations:
6      kubernetes.io/tls-acme: "true"
7      cert-manager.io/cluster-issuer: "letsencrypt-prod"
8  spec:
9    tls:
10     - hosts:
11         - keycloak2.cloudarmin.de
12       secretName: keycloak
13   rules:
14     - host: keycloak2.cloudarmin.de
15       http:
16         paths:
17           - backend:
18               serviceName: keycloak
19               servicePort: 8080
```

Listing 7.1: *KeyCloack Ingress*

7.3.2 CONNECT KEYCLOAK TO ACTIVE DIRECTORY

After deploying the Keycloak. we need to go to **User Federation** and select **LDAP**. as can be seen in the Figure 26

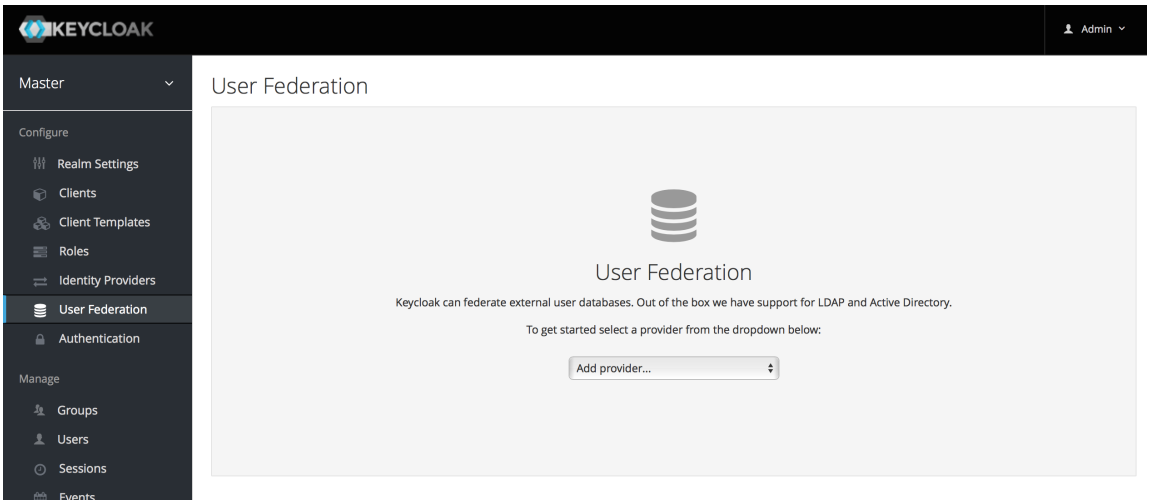

Figure 26: *LDAP Federation*

Then we configure the integration as can be seen in the Figure 27

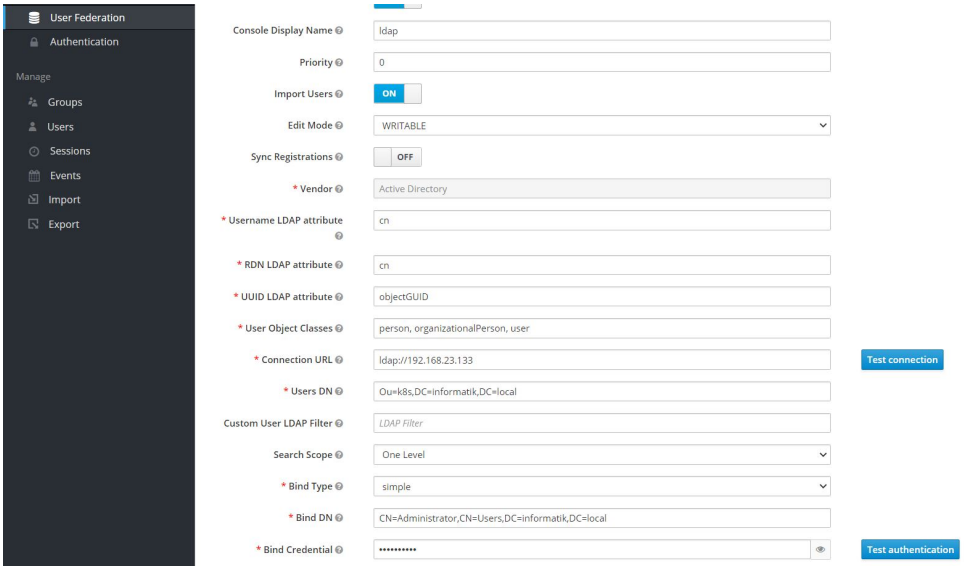

Figure 27: *Configure LDAP Connection*

After that, we configure the *Sync settings* to enable periodic synchronization, and at the end, we sync all the users.

7.3.3 CONFIGURE MAPPER FOR EMAIL VERIFICATION

By default, all of the users synced from LDAP or Active Directory are considered unverified users, which means that they need to verify their email addresses. A user who is not verified cannot access the cluster. We need to change this behavior and specify that all synced users from Active Directory are verified. To do that, we need to create a mapper.

First, we should go to the mapper part of our LDAP settings, as can be seen in the Figure 28

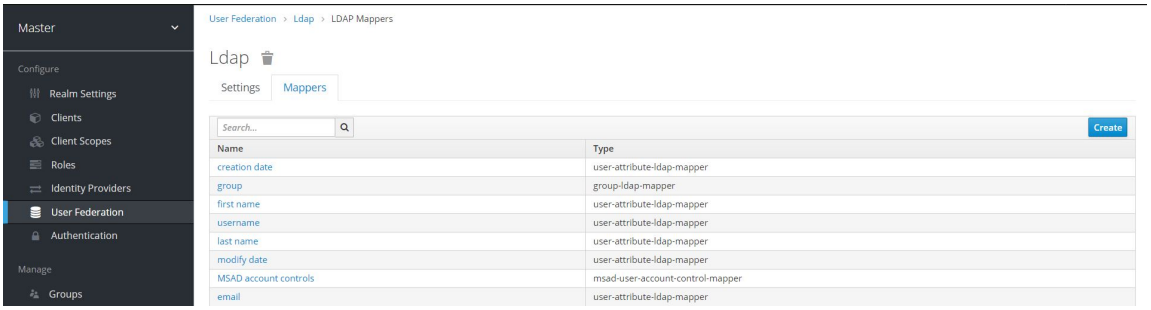

Figure 28: *Mapper Settings*

Then we create a mapper and configure it the same as the Figure 29

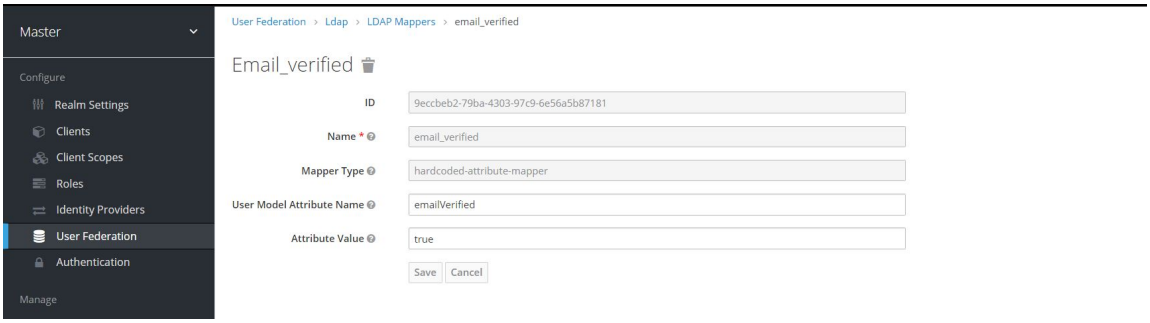

Figure 29: *Email Verified Mapper*

7.3.4 CREATE AN OIDC CLIENT

We need to configure an OIDC Client in our Keycloak for Kubernetes and **Gangway**. Gangway is an application that can be used to enable authentication flows via OIDC for a Kubernetes cluster easily. It provides us a UI that users can use to authenticate themselves against Keycloak and, after authentication, give all of the **Kubectl** configuration that the user needs to access to the cluster. [21d]

We need to go to the Client part and create a client with the configuration of Figure 30. Keycloak will use the *Redirect URL* to send token details, and in our configuration, it sends token information to our Gangway (its address is gangway.cloudarmin.de) and Gangway uses them to build the Kubectl configuration.

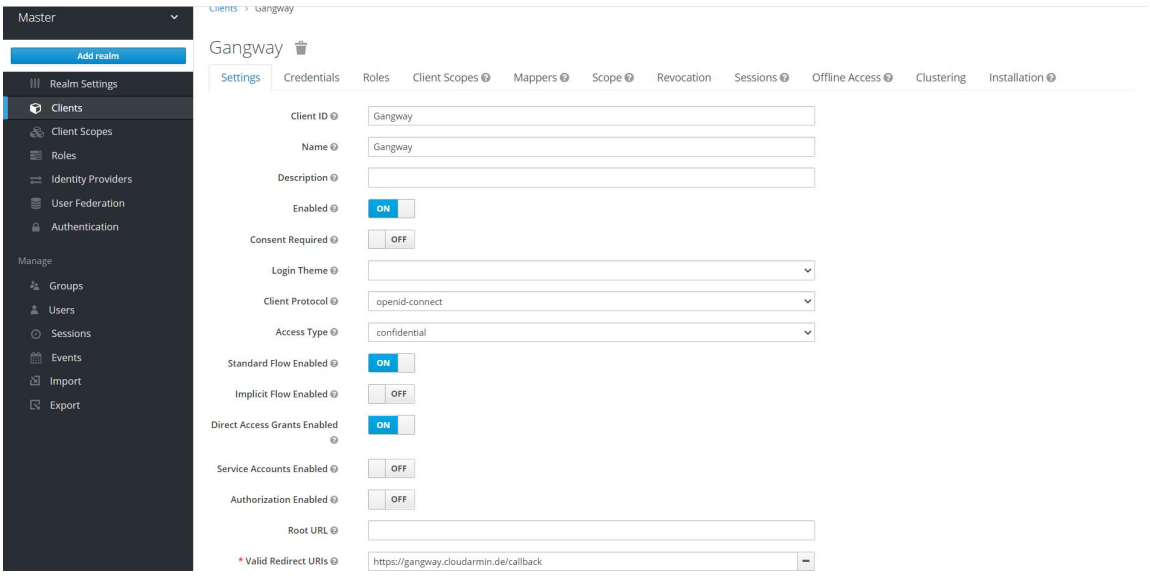

Figure 30: *OIDC Client Configuration*

As shown in the Figure 31, the Client Secret can be accessed from Credential Tab. we need it to configure Gangway.

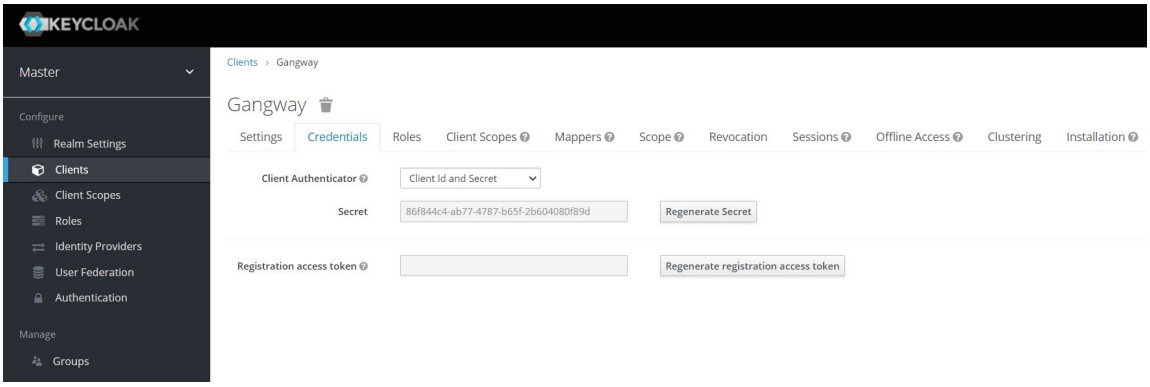

Figure 31: *OIDC Client Configuration*

7.3.5 DEPLOY AND CONFIGURE GANGWAY

CREATE NAMESPACE

We create a dedicated Namespace for Gangway.

```
apiVersion: v1
kind: Namespace
metadata:
  name: gangway
```

Listing 7.2: *Gangway Namespace*

CREATE CONFIGMAP

Now we create a configMap for and put Gangway configuration on it.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: gangway
  namespace: gangway
data:
  gangway.yaml: |
    # The address to listen on. Defaults to 0.0.0.0 to listen on all interfaces.
    # Env var: GANGWAY_HOST
    # host: 0.0.0.0
    # The port to listen on. Defaults to 8080.
    # Env var: GANGWAY_PORT
    # port: 8080
    # Should Gangway serve TLS vs. plain HTTP? Default: false
    # Env var: GANGWAY_SERVE_TLS
    # serveTLS: false
    # The public cert file (including root and intermediates) to use when serving
    # TLS.
    # Env var: GANGWAY_CERT_FILE
    # certFile: /etc/gangway/tls/tls.crt
    # The private key file when serving TLS.
    # Env var: GANGWAY_KEY_FILE
```

```
23      # keyFile: /etc/gangway/tls/tls.key
24      # The cluster name. Used in UI and kubectl config instructions.
25      # Env var: GANGWAY_CLUSTER_NAME
26      clusterName: "Informatik"
27      # OAuth2 URL to start authorization flow.
28      # Env var: GANGWAY_AUTHORIZE_URL
29      authorizeURL: "https://keycloak2.cloudarmin.de/auth/realms/master/protocol/openid-
        connect/auth"
30      # OAuth2 URL to obtain access tokens.
31      # Env var: GANGWAY_TOKEN_URL
32      tokenURL: "https://keycloak2.cloudarmin.de/auth/realms/master/protocol/openid-
        connect/token"
33      # Endpoint that provides user profile information [optional]. Not all providers
34      # will require this.
35      # Env var: GANGWAY_AUDIENCE
36      # audience: "https://oauth.example.com/userinfo"
37      # Used to specify the scope of the requested Oauth authorization.
38      # scopes: ["openid", "profile", "email", "offline_access"]
39      # Where to redirect back to. This should be a URL where gangway is reachable.
40      # Typically this also needs to be registered as part of the oauth application
41      # with the oAuth provider.
42      # Env var: GANGWAY_REDIRECT_URL
43      redirectURL: "https://gangway.cloudarmin.de/callback"
44      # API client ID as indicated by the identity provider
45      # Env var: GANGWAY_CLIENT_ID
46      clientID: "Gangway"
47      # API client secret as indicated by the identity provider
48      # Env var: GANGWAY_CLIENT_SECRET
49      clientSecret: "9fce6af7-21d6-4269-91d2-7c2ebb375ffc"
50      # The JWT claim to use as the username. This is used in UI.
51      # Default is "nickname".
52      # Env var: GANGWAY_USERNAME_CLAIM
53      usernameClaim: "sub"
54      # The JWT claim to use as the email claim. This is used to name the
55      # "user" part of the config. Default is "email".
56      # Env var: GANGWAY_EMAIL_CLAIM
57      emailClaim: "email"
58      # The API server endpoint used to configure kubectl
59      # Env var: GANGWAY_APISERVER_URL
60      apiServerURL: https://192.168.23.180:6443
61      # The path to find the CA bundle for the API server. Used to configure kubectl.
62      # This is typically mounted into the default location for workloads running on
63      # a Kubernetes cluster and doesn't need to be set.
64      # Env var: GANGWAY_CLUSTER_CA_PATH
65      # cluster_ca_path: "/var/run/secrets/kubernetes.io/serviceaccount/ca.crt"
66      clusterCAPath: "/var/run/secrets/kubernetes.io/serviceaccount/ca.crt"
```

Listing 7.3: *Gangway ConfigMap*

Most options are explained well in the file. The *emailClaim: "email"* means we used email as subject of RoleBinding.The *redirectURL* should be change based on your domain name. *apiServerURL* specifies the address of Kubernetes API server. *clientSecret* specifies the client secret that we got it from Keycloak. The *authorizeURL* and *tokenURL* can be found in *Endpoints in Realm settings* of Keycloak and should be changed based on your domain.

GENERATE SESSION KEY

We should generate a session key that Gangway will use. We can use the following command:

```
kubectl -n gangway create secret generic gangway-key \
--from-literal=sessionkey=$(openssl rand -base64 32)
```

Listing 7.4: *Generate Session Key*

DEPLOY GANGWAY

In order to deploy the Gangway, the configuration in the Figure 32 should be used:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gangway
  namespace: gangway
  labels:
    app: gangway
spec:
  replicas: 1
  selector:
    matchLabels:
      app: gangway
  template:
    metadata:
      labels:
        app: gangway
        revision: "1"
    spec:
      containers:
        - name: gangway
          image: gcr.io/heptio-images/gangway:v3.0.0
          imagePullPolicy: Always
          command: ["gangway", "-config", "/gangway/gangway.yaml"]
          env:
            - name: GANGWAY_SESSION_SECURITY_KEY
              valueFrom:
                secretKeyRef:
                  name: gangway-key
                  key: sessionkey
          ports:
            - name: http
              containerPort: 8080
              protocol: TCP
          resources:
            requests:
              cpu: "100m"
              memory: "128Mi"
            limits:
              cpu: "200m"
              memory: "512Mi"
          volumeMounts:
            - name: gangway
              mountPath: /gangway/
          livenessProbe:
            httpGet:
              path: /
              port: 8080
            initialDelaySeconds: 20
            timeoutSeconds: 1
            periodSeconds: 60
            failureThreshold: 3
          readinessProbe:
            httpGet:
              path: /
              port: 8080
            timeoutSeconds: 1
            periodSeconds: 10
            failureThreshold: 3
      volumes:
        - name: gangway
          configMap:
            name: gangway
```

The actual YAML file can be found here. [21e]

> **Note:**
> In the deployment file of in Gangway repository the version of v3.2.0 is used but you should changed it to v3.0.0 same as the our configuration in the Figure 32

CONFIGURE SERVICE

To access Gangway, we need to create a service as follow:

```
kind: Service
apiVersion: v1
metadata:
  name: gangwaysvc
  namespace: gangway
  labels:
    app: gangway
spec:
  type: ClusterIP
  ports:
    - name: "http"
      protocol: TCP
      port: 80
      targetPort: "http"
  selector:
    app: gangway
```

Listing 7.5: *Create a Service*

The Manifest file can be found in the Gangway repository too.

CONFIGURE INGRESS AND CERTIFICATE

We used the name of *gangway.cloudarmin.de* for our Gangway and setup ingress and certificate for it as follow:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: gangway
  namespace: gangway
  annotations:
    kubernetes.io/tls-acme: "true"
    cert-manager.io/cluster-issuer: "letsencrypt-prod"
spec:
  tls:
  - secretName: gangway
    hosts:
      - gangway.cloudarmin.de
  rules:
    - host: gangway.cloudarmin.de
      http:
        paths:
          - backend:
            serviceName: gangwaysvc
```

```
20              servicePort: http
```

Listing 7.6: *Create Ingress and Certificate*

7.3.6 CONFIGURE API SERVER

We need to configure the Kubernetes API Server and inform it to use OIDC as an Authentication strategy and how to use it. We need to modify the API Server manifest file in this location `/etc/kubernetes/manifests/kube-apiserver.yaml` and add the following options to API Server:

```
1 --oidc-issuer-url=https://keycloak2.cloudarmin.de/auth/realms/master  #The issuer URL
      of Tokens
2 --oidc-client-id=Gangway # OIDC Client that we creat in Keycloak
3 --oidc-username-claim=email # the attribute of email will be used as username
4 --oidc-groups-claim=groups # the attribute of groups specifies user group memebership
```

Listing 7.7: *Configure Kubernetes API Server*

By editing and save the *Kubelet* will restart the API Server and deploy it with the new configuration.

References

[21a] *Cert Manager*. 2021. URL: https://cert-manager.io/docs/ (visited on 03/03/2021).

[21b] *Cert-Manager Documentation*. 2021. URL: https://cert-manager.io/docs/installation/kubernetes/ (visited on 03/17/2021).

[21c] *Christis Repository*. 2021. URL: https://github.com/araminian/christis (visited on 03/28/2021).

[21d] *Gangway*. 2021. URL: https://github.com/heptiolabs/gangway (visited on 03/03/2021).

[21e] *Gangway Deployment Manifest File*. 2021. URL: https://github.com/heptiolabs/gangway/blob/master/docs/yaml/03-deployment.yaml (visited on 03/17/2021).

[21f] *HAProxy*. 2021. URL: http://www.haproxy.org/ (visited on 03/17/2021).

[21g] *Helm Documentation*. 2021. URL: https://helm.sh/docs/ (visited on 03/17/2021).

[21h] *KeyCloak Configuration Options*. 2021. URL: https://hub.docker.com/r/jboss/keycloak/ (visited on 03/17/2021).

[21i] *KeyCloak Documentation*. 2021. URL: https://www.keycloak.org/documentation (visited on 03/17/2021).

[21j] *Kubernetes authentication*. 2021. URL: https://kubernetes.io/docs/reference/access-authn-authz/authentication/ (visited on 03/03/2021).

[21k] *Kubernetes authorization*. 2021. URL: https://kubernetes.io/docs/reference/access-authn-authz/authorization/ (visited on 03/03/2021).

[21l] *Kubernetes components*. 2021. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 03/03/2021).

[21m] *MongoDB Documentation*. 2021. URL: https://hub.docker.com/_/mongo (visited on 03/17/2021).

[21n] *Nginx Ingress Bare-Metal Considerations*. 2021. URL: https://kubernetes.github.io/ingress-nginx/deploy/baremetal/ (visited on 03/17/2021).

[21o] *Nginx Ingress Documentation*. 2021. URL: https://kubernetes.github.io/ingress-nginx/deploy/ (visited on 03/17/2021).

[21p] *OpenID Connect*. 2021. URL: https://openid.net/connect/ (visited on 03/03/2021).

[21q] *Setup KeyCloak on Kubernetes*. 2021. URL: https://www.keycloak.org/getting-started/getting-started-kube (visited on 03/17/2021).

[21r] *What is Helm*. 2021. URL: https://helm.sh/ (visited on 03/03/2021).

[21s] *What is Keycloak*. 2021. URL: https://www.keycloak.org/about (visited on 03/03/2021).

[21t] *What is kubernetes*. 2021. URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 03/03/2021).

List of Figures

1 Deployment Eras [21t] . . . 3
2 Kubernetes cluster components [21l] . . . 3
3 OIDC Authentication [21j] . . . 5
4 Helm in abstract . . . 7

5 Christis in abstract . . . 11
6 User Synchronization . . . 12
7 Role creation and add . . . 13
8 Role assignment . . . 14
9 Removing an user from K8sGroup . . . 15

10 Add user to K8sGroup . . . 20
11 Database help command . . . 21
12 Database Sync . . . 21
13 Create a Chart . . . 21
14 Role binding Template . . . 22
15 Create Role . . . 23
16 View Role . . . 23
17 View Stage table . . . 23
18 Role assignment . . . 24
19 Role assignment view . . . 24
20 Gangway Sign-in . . . 24
21 Authenticate in the Keycloak . . . 25
22 Get Kubectl configuration from Gangway . . . 25
23 Test user role assignment and authentication strategy . . . 26
24 Unassign Role . . . 26
25 Test role unassignment . . . 26

26 LDAP Federation . . . 29
27 Configure LDAP Connection . . . 30
28 Mapper Settings . . . 30
29 Email Verified Mapper . . . 31
30 OIDC Client Configuration . . . 31
31 OIDC Client Configuration . . . 32
32 Deploy Gangway . . . 35

Listings

4.1 Install Christis 16
4.2 Deploy Christis Database 17
4.3 Generate a Configuration file for Christis CLI to access Database 17
4.4 Deploy Christis API 17
4.5 Christis API configuration file 18
4.6 Use Christis to generate the ChristisAPI configuration file 18
4.7 Configure Christis CLI 18

7.1 KeyCloack Ingress 28
7.2 Gangway Namespace 32
7.3 Gangway ConfigMap 32
7.4 Generate Session Key 34
7.5 Create a Service 36
7.6 Create Ingress and Certificate 36
7.7 Configure Kubernetes API Server 37