Technical University of Crete

School of Electrical & Computer Engineering

Diploma Thesis

Design of Access Control in STELAR Knowledge Lake Management System

This Thesis Was Submitted in Partial Fulfillment of the Requirements
for the Diploma in Electrical and Computer Engineering

Author:

Nikolaos I. Bakatselos

Thesis Committee:

Prof. Vasileios Samoladas, *Supervisor*

Prof. Evripidis Petrakis

Prof. Nikolaos Giatrakos

June 2025

Abstract

As data ecosystems grow in scale and complexity, ensuring secure and consistent access control becomes a critical challenge—especially in integrated platforms like the STELAR Knowledge Lake Management System (KLMS). Designed for the agri-food sector, STELAR brings together multiple services, including metadata cataloging (CKAN), object storage (MinIO), and workflow management, each with its own access control mechanisms. This diversity can lead to fragmented security policies, redundant configurations, and administrative overhead.

This thesis presents the design and implementation of a centralized, declarative access control framework for STELAR KLMS. The proposed system introduces a YAML-based policy specification language that allows administrators to define fine-grained, role-based access rules in a human-readable and maintainable format. Identity and access management are unified through Keycloak, enabling single sign-on and federated authentication using standards like OAuth 2.0 and OpenID Connect.

The access control architecture includes a core policy controller, real-time policy evaluation, and automated reconciliation mechanisms that synchronize permissions across all components. This approach ensures scalable, consistent, and context-aware access control while reducing configuration errors and improving overall platform security. The system has been evaluated in realistic deployment scenarios, demonstrating its effectiveness, performance, and administrative efficiency.

Acknowledgements

This thesis was the result of a long journey that would not have been possible without the support of many people. First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Vasilis Samoladas, for his invaluable guidance, support, and encouragement throughout this effort. His expertise and insights on the field have been instrumental in shaping this work. The love for research and dedication to excellence that he instilled in me will forever be a source of inspiration.

My heartfelt thanks go as well to the members of the committee of this thesis, Professor Evripidis Petrakis and Professor Nikoloas Giatrakos. During my university days, I have acquired a solid foundation in the field of computer science through their courses which has been essential for the completion of this thesis and will be a valuable asset in my future endeavors.

I cannot forget to mention the unconditional support I received from my colleagues, but most importantly friends, Dimitris Petrou and Michael Theologitis. Our countless discussions, brainstorming sessions, and shared experiences have not only enriched this work but also made the journey enjoyable and memorable. My appreciation extends to the entire team of the STELAR project, whose collaborative spirit and dedication to excellence have been a constant source of motivation.

But all these would not have been possible without the unwavering support of my family and friends. To my parents, who have always believed in me and encouraged me to pursue my dreams, I am forever grateful. To my friends, who have stood by me through thick and thin, providing encouragement and support when I needed it most, I am truly thankful.

I hope that this thesis serves as a testament to the power of collaboration, dedication, and the pursuit of knowledge. I wish to everyone who has been a part of this journey to know that their contributions have made a significant impact on my life and work. I dedicate this thesis to all of you, with heartfelt appreciation and gratitude.

This thesis was partially funded by the European Commission under the STELAR (HORIZON-EUROPE - Grant Agreement No. 101070122) project.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

We are living in the era of data ubiquity, where massive volumes of information are generated continuously from a variety of sources, including web platforms, IoT devices, enterprise applications, and scientific instruments. This data is often diverse in format—structured, semi-structured and unstructured—and requires scalable, flexible architectures for storage and processing.

Data lakes have emerged as a modern paradigm that supports the ingestion and long-term retention of raw data, without the need for upfront schema definitions. Building on this concept, knowledge lakes extend the capabilities of data lakes by incorporating metadata, semantic enrichment, data governance and enhanced discoverability. These systems aim to transform raw data into actionable knowledge through better contextualization and integration.

The STELAR Knowledge Lake Management System (KLMS) is one such platform, developed to support the agri-food data space with robust infrastructure for data cataloging, storage, workflow orchestration and analytics. It integrates open-source components such as CKAN (for metadata management), MinIO (for object storage) and Keycloak (for identity and access management).

In such distributed architectures, where multiple services interact across different domains and user contexts, access control becomes a central concern. Ensuring that only authorized users can access specific resources—and do so under well-defined conditions—is essential for maintaining security, compliance and operational efficiency. Centralized, flexible and fine-grained access control systems are thus crucial components of any scalable knowledge lake.

1.2 Problem Statement

While platforms like STELAR KLMS provide powerful capabilities for data storage, cataloging and processing, they often face critical challenges in enforcing coherent and consistent access control policies across their distributed components. Each service—such as CKAN, MinIO, or custom APIs—may implement its own model for authentication and authorization, leading to a fragmented security architecture. This fragmentation introduces several risks:

- **Inconsistency** in access policies across services,
- **Manual and error-prone configurations** for managing user roles and permissions,
- **Redundant credential handling**, reducing usability and increasing security exposure,
- **Difficulties in auditing and maintaining compliance** in multi-tenant environments.

Moreover, the need for fine-grained, dynamic and context-aware access policies becomes more pronounced in knowledge lakes, where resources are not limited to static files but include datasets, metadata entities, tools and multi-step workflows. Traditional access models often fail to capture the complex authorization requirements in such environments.

To address these issues, this thesis proposes the design and implementation of a declarative, centralized access control system for STELAR KLMS. The proposed solution seeks to unify identity management and authorization logic, enabling administrators to define and manage access policies using a YAML-based specification language that supports role-based control, dynamic evaluation and automated reconciliation across all platform components.

1.3 Thesis Outline

This thesis is organized into four chapters, each addressing a core aspect of the research and development work undertaken:

- **Chapter 2 - Related Work:** Reviews foundational concepts and existing technologies relevant to the access control problem space. Topics covered include authentication and authorization mechanisms, access control models (with an emphasis on Role-Based Access Control), federated identity protocols (e.g., OAuth 2.0 and OpenID Connect), policy-as-code frameworks (e.g., OPA) and Identity and Access Management (IAM) systems such as Keycloak. It also explores supporting technologies used in STELAR KLMS, including MinIO for object storage and CKAN for data cataloging.
- **Chapter 3 - Access Control in STELAR KLMS:** Presents the design and architecture of the proposed access control system. This includes the integration of centralized identity management using Keycloak, the definition of fine-grained permissions and the development of a YAML-based policy specification language. The chapter also explains the enforcement and reconciliation mechanisms that ensure consistent and scalable access control across the platform.
- **Chapter 4 - Implementation and Evaluation:** Details the implementation of the access control framework, including the parsing and application of policy documents, integration with external services and policy evaluation workflows. It also presents evaluation scenarios that demonstrate the correctness, security and operational efficiency of the proposed system.

Chapter 2

Related Work

2.1 Authentication and Authorization

Every modern system requires mechanisms to control and manage access to resources and services securely. From our personal devices to large-scale enterprise systems, the need to ensure the safety and integrity of data is paramount. In this context, two fundamental concepts emerge: **Authentication** and **Authorization**. Although often used interchangeably, they serve distinct purposes in the realm of security. Understanding the differences between these two concepts is crucial for anyone involved in system design, security or IT management. In this section, we will define both of these concepts, explore their differences, and discuss their significance in the context of modern systems.

2.1.1 What is Authentication?

Authentication is the process of verifying the identity of a user, device, or system. It the answer to the question: "Who are you?". The goal of authentication is to ensure that the entity attempting to access a system or resource is indeed who they claim to be. This is typically achieved through various methods, including:

- **Something you know:** This includes passwords, PINs, or answers to security questions.
- **Something you have:** This could be a physical token, smart card, or mobile device.
- **Something you are:** This refers to biometric data, such as fingerprints, facial recognition, or iris scans.
- **Somewhere you are:** This involves location-based authentication, where access is granted based on the user's geographical location.
- **Something you do:** This includes behavioral biometrics, such as typing patterns or mouse movements.

The most common form of authentication is the username and password combination, where users provide a unique identifier (username) and a secret (password) to gain access to a system. However, this method has its limitations, as passwords can be stolen, guessed, or compromised. To enhance security, many systems now implement multi-factor

authentication (MFA), which requires users to provide two or more forms of verification before granting access. MFA significantly reduces the risk of unauthorized access, as it adds an additional layer of security beyond just a password.

2.1.2 What is Authorization?

Authorization is the process of determining whether a user, device, or system has the right to access a specific resource or perform a particular action. It answers the question: "What are you allowed to do?". Authorization occurs after authentication and is based on the identity established during the authentication process.

A helpful way to conceptualize authorization is through the analogy of a theater performance. The theater building represents the system, and each part of the building—such as the stage, backstage, and audience area—corresponds to a **resource**. The actors, crew, and audience members represent the **users**. Upon entering the theater, every individual must present some form of identification (authentication) to gain access: audience members may show tickets, while actors and crew members may present ID badges.

After authentication, each person is granted different levels of access according to their role. For example:

- Actors may access the stage and backstage.
- Crew members may access the backstage and technical areas.
- Audience members may only access the audience area.
- Security personnel may access all areas to ensure safety.

These access rights represent a set of **permissions**, which are specific privileges granted to users or groups to perform certain actions on resources. A collection of permissions forms a **policy**. **Policies** [1] define the rules and conditions under which access is either granted or denied. Continuing the analogy, policies are similar to the theater's staff handbook, which outlines who can perform **what actions** on **which resources**.

Suppose the theater manager hires a decorator to renovate the backstage area. The decorator requests access specifically to that part of the theater. This scenario illustrates the concept of a **scope**—a defined context within which a permission is valid. Scopes limit the extent of a user's permissions, ensuring that actions are performed only within a designated environment.

Attributes also play an important role in authorization decisions. For example, if a theater performance is restricted to adults, verifying a ticket alone (authentication) is insufficient. The theater staff must also confirm the age of the audience member to enforce the access policy. Here, **age** is an **attribute**—a characteristic or property of the user. Other attributes might include name, language, or nationality.

When a user presents information such as an ID card stating their age, this information is referred to as a **claim**. Claims are statements about a user, issued by a trusted authority, and are used to make authorization decisions. In this case, the ID card provides a claim about the individual's age, which staff members use to determine eligibility for entry.

Thus, the ticket and ID card can be viewed as forms of **access tokens** that users present to the system (theater staff) to gain access to the resource (the performance).

In summary, authentication and authorization are two distinct but interrelated processes that work together to secure access to resources. Authentication verifies identity, while authorization determines access rights based on that identity. Understanding these concepts is essential for designing secure systems and ensuring that users can only access the resources they are entitled to.

2.2 Access Control Models

Access control models [2] define the frameworks and methodologies used to manage and control access to resources within a system. if the authorization process answers the question "What are you allowed to do?", access control models provide the rules and structures to help the system answer that question in a consistent and secure manner.

Access control models are essential for ensuring that only authorized users can access specific resources and perform certain actions. There are several access control models, each with its own principles and use cases [3]. The most common ones include:

- **Discretionary Access Control (DAC)**: In this model, the owner of a resource has the authority to grant or deny access to that resource. The owner can specify who can access the resource and what actions they can perform. This model is flexible but can lead to security risks if not managed properly.

- **Mandatory Access Control (MAC)**: In MAC, access rights are regulated by a central authority based on multiple levels of security. Users cannot change access permissions; instead, they are determined by the system based on predefined policies. This model is often used in environments where security is critical, such as military applications.

- **Role-Based Access Control (RBAC)**: RBAC assigns permissions based on user roles within an organization. Each role has specific permissions associated with it, and users are assigned roles based on their job functions. This model simplifies management and enhances security by ensuring that users only have access to the resources necessary for their roles.

- **Attribute-Based Access Control (ABAC)**: ABAC [4] uses attributes (characteristics) of users, resources, and the environment to make access control decisions. Policies are defined using these attributes, allowing for more granular and dynamic access control. This model is particularly useful in complex environments where user roles and resource characteristics may change frequently.

The selection of an access control model depends on the specific requirements of the system, the sensitivity of the data, and the organizational policies in place. For instance, a healthcare system may require a more stringent model like MAC to protect sensitive patient data, while a collaborative project management tool may benefit from the flexibility of DAC. Even if MAC can be considered a more secure model, introducing new users and resources can be cumbersome and time-consuming because of the need to define and manage security labels. This renders MAC less suitable for dynamic environments where users and resources frequently change. Also, DAC, even if provides more flexibility than MAC, can lead to security risks if not managed properly. This is because users can grant access to resources they own, potentially exposing sensitive data to unauthorized users. RBAC, on the other hand, strikes a balance between security and flexibility by allowing organizations to define roles and permissions based on job functions. This makes it easier to manage access control in dynamic environments, like big data infrastructures, while still maintaining a high level of security. RBAC is widely used in various applications, including enterprise systems, cloud computing and web applications. It provides a structured approach to access control, making it easier to manage user permissions and ensuring that users only have access to the resources that their role declares. This reduces the risk of unauthorized access and helps organizations comply with regulatory requirements.

2.2.1 Role-Based Access Control (RBAC)

The Role-Based Access Control (RBAC) model associates permissions with roles rather than assigning them directly to individual users. In RBAC, users are assigned to one or more roles based on their responsibilities, and each role is granted a specific set of permissions. This abstraction greatly simplifies access management in systems with a large number of users and resources.

Returning to the theater analogy, we can think of roles such as audience, actor, and director. Each role is associated with different access rights—for example, audience members are permitted to watch the performance, actors can access the stage and script, and directors have additional privileges such as modifying the play's structure. Instead of assigning access permissions to every individual, the system assigns them to roles, and then simply associates users with the appropriate roles. This enables centralized and scalable management of authorization.

RBAC also supports the concept of users having multiple roles. For instance, a person might simultaneously be an actor and a director, thereby inheriting the combined permissions of both roles. This flexibility allows RBAC to support more dynamic access scenarios while maintaining a clear organizational structure.

Key components of RBAC include:

- **Users**: The individuals who access the system.
- **Roles**: Named job functions or titles within the organization that define an authority level.
- **Permissions**: The approved operations or actions that roles are allowed to perform on resources.

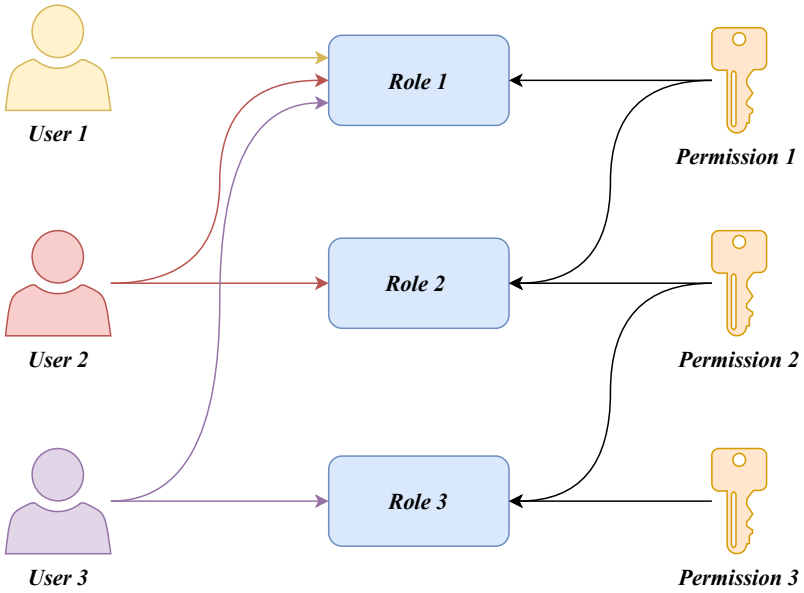

Figure 2.1: Role-Based Access Control (RBAC) Model

RBAC also allows for the definition of **role hierarchies**, where higher-level roles inherit permissions from lower-level ones. For example, a senior administrator role might include all the permissions of a data curator, along with additional privileges. Constraints can also be applied to enforce policies such as separation of duties, ensuring that conflicting roles (e.g., auditor and accountant) are not assigned to the same user.

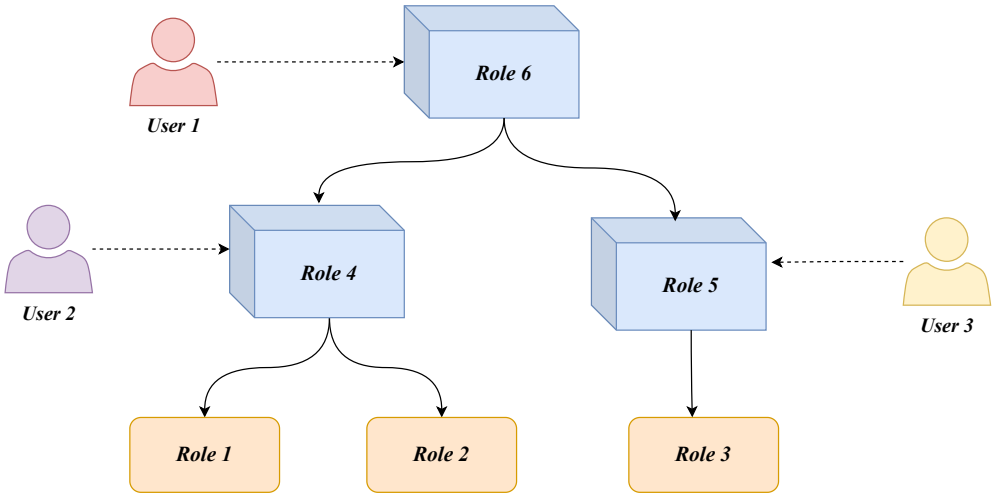

Figure 2.2: Role Hierarchy in RBAC

The advantages of RBAC are particularly relevant in structured environments like the one implemented in this thesis. It promotes:

- **Scalability**: By managing permissions at the role level, changes to user assignments or access rules are efficient and low-maintenance.
- **Security**: RBAC helps enforce the principle of least privilege, where users receive only the access necessary for their roles.

- **Auditability**: The use of roles and centralized policies simplifies security audits and access reviews.

While access models are essential because they provide a structured approach to define the level of access users have to resources, modern systems require mechanisms to determine users identity and how to verify it accross multiple services. In distributed systems authentication and authorization are managed by individual applications, providing centralized control, single sign-on (SSO) and federated identity management. To accomplish this, these application rely on a set of protocols such as OAuth, OpenID Connect.

2.3 Federated Identity and Authorization Protocols

Federated identity and authorization protocols are standards utilized by systems to faciliate secure identity and access management accross distributed systems. More specific, these protocols are designed to enable users to authenticate and authorize access to multiple applications or services using a single set of credentials. The most widely adopted protocols are OAuth 2.0 and OpenID Connect, which are often used in conjunction to allow services delegate authentication and authorization tasks to a trusted identity provider (IdP). This enables a centralized point of user management and seamless integration between different services.

2.3.1 OAuth 2.0

OAuth 2.0 [5] is an authorization framework that allows third-party applications to obtain limited access to a user's resources without exposing their credentials. Instead of authenticating against each service directly, users authorize a trusted application to act on their behalf by granting it an access token issued by an identity provider (IdP). By this way, the application can access the user's resources without needing to know the user's credentials. For example, let's say a user wants to login to a third-party application without creating a new account. The user can choose to log in using his Google account. The third-party application redirects the user to the Google authorization server, where the user is prompted to grant permission for the application to access his Google account information. Once the user grants permission, Google issues an access token to the third-party application, allowing it to access the user's Google account information without needing to know the user's password. By this way, the user decides which information to share with the third-part application, or the **scope** of access we mentioned in Authorization terminology presented at section 2.1.2, protecting his credentials and privacy. The OAuth 2.0 framework defines several roles:

- **Resource Owner**: The user who owns the resources and grants access to them.
- **Client**: The application requesting access to the resource owner's resources.
- **Authorization Server**: The server that issues access tokens to the client after successfully authenticating the resource owner and obtaining authorization.
- **Resource Server**: The server hosting the protected resources, which accepts and validates access tokens.

OAuth 2.0 Flow

The OAuth 2.0 flow is a series of steps that facilitate the authorization process between the resource owner, client, authorization server and resource server.

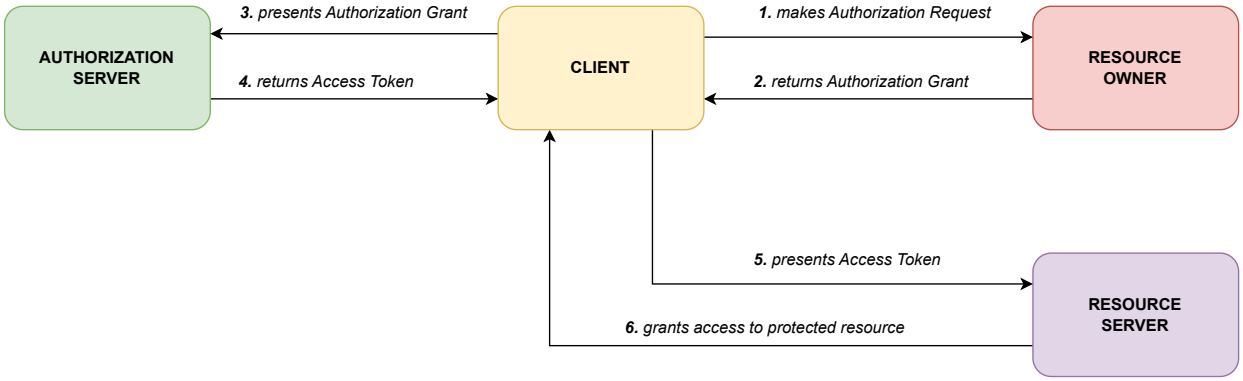

Figure 2.3: OAuth 2.0 Flow

According to diagram illustared in 2.3, the OAuth 2.0 flow consists of the following steps as described in the RFC 6749 [5]:

1. The **client** requests authorization from the **resource owner**. This request can be directly from the client or through a redirect to the authorization server.
2. The **client** receives an **authorization grant** from the **resource owner**. This grant can be in the form of an **authorization code**, **implicit token**, or other types defined by the OAuth 2.0 specification.
3. The **client** requests an **access token** by authenticating with the **authorization server** and presenting the **authorization grant**.
4. The **authorization server** authenticates the **client** and validates the **authorization grant**. If valid, it issues an **access token** to the **client**.
5. The **client** uses the **access token** to access the protected resources on the **resource server**.
6. The **resource** server validates the **access token** and grants or denies access to the requested resources.

Authorization Grant

Authorization grant is credentials representing the resource owner's authorization to access their resources. This credentials are used by the client in order to obtain an access token from the authorization server. According to the OAuth 2.0 specification, there are four types of authorization grants:

- **Authorization Code Grant**: Used for server-side applications. The client requests authorization directly from the authorization server by redirecting the resource owner to the authorization server's authorization endpoint. After successful authentication, the authorization server redirects the resource owner back to the client with an authorization code. This grant type is considered more secure because the resource owner authenticates directly with the authorization server, preventing the credentials to be exposed to the client. Also, the authorization code is exchanged for an access token, which is sent to the client over a secure channel.

- **Implicit Grant**: Used for client-side applications (e.g., single-page applications). The client receives an access token directly from the authorization server without an intermediate authorization code. Although implicit grant improves response time and reduces complexity, it is less secure than the authorization code grant because the access token is exposed to the client and can be intercepted by malicious actors.

- **Resource Owner Password Credentials Grant**: Used when the resource owner trusts the client with their credentials. The client exchanges the resource owner's username and password for an access token.

- **Client Credentials Grant**: Used for machine-to-machine communication. The client authenticates with the authorization server using its own credentials to obtain an access token.

Access Token

Access token represents a credential in string format, granted to the client by the authorization server on behalf of the resource owner, that is used by the client to access protected resources on the resource server. They represent specific scopes and durations of access, and they are typically short-lived to minimize security risks.

Tokens may contain specific identifiers used to obtain authorization information or contain the authorization information themselves. They provide a layer of abstraction as they replace other forms of credentials, such as passwords, with just a single piece of information that can be easily interpreted by the resource server. Access tokens can be in different formats, including:

- **Opaque Tokens**: These tokens are random strings with no inherent meaning. The resource server must validate them with the authorization server to obtain the associated user information.

- **JWT (JSON Web Tokens)**: These tokens are self-contained and can be verified without contacting the authorization server. They contain a header, payload, and signature, allowing the resource server to decode and validate the token. The payload typically includes claims about the user, such as their identity and permissions.

- **Bearer Tokens**: These tokens are used in the HTTP Authorization header to grant access to protected resources. The resource server validates the token and grants access based on its validity and associated permissions.

- **Refresh Tokens**: These tokens are used to obtain new access tokens without requiring the resource owner to re-authenticate. They are typically long-lived and can be used to refresh access tokens when they expire.

The content of a decoded JWT access token is shown in Listing 2.1.

```
{
  "header": {
    "alg": "HS256",
    "typ": "JWT"
  },
  "payload": {
    "sub": "1234567890", // Subject (user ID)
    "name": "John Doe",  // User's name
    "iat": 1516239022,   // Issued at (timestamp)
    "exp": 1516242622,   // Expiration time (timestamp)
    "scope": "read write" // Scopes granted to the token
  },
  "signature": "HMACSHA256(base64UrlEncode(header) + '.' +
   base64UrlEncode(payload), secret)"
}
```

Listing 2.1: Example of a JWT Access Token

2.3.2 OpenID Connect

OpenID Connect (OIDC) [6] is an identity layer built on top of the OAuth 2.0 protocol, designed to provide authentication capabilities in addition to the authorization mechanisms offered by OAuth. While OAuth 2.0 enables clients to access protected resources on behalf of users, it does not specify how to authenticate users or obtain their identity information. OIDC addresses this gap by introducing a standardized way for clients to verify a user's identity and retrieve basic profile details from a trusted identity provider (IdP).

A central element introduced by OIDC is the **ID Token**, which is a JSON Web Token (JWT) issued by the authorization server upon successful authentication. This token contains a set of **claims**, such as the user's unique identifier (`sub`), authentication timestamp and optional profile attributes like name, email or locale. Unlike access tokens, which are intended for use by resource servers to authorize API access, ID tokens are designed for the client application and serve as cryptographically verifiable proof of the user's identity.

OIDC also defines a number of standard endpoints to facilitate integration:

- The **Discovery Endpoint** provides metadata about the identity provider, such as token and authorization endpoints, supported scopes, and public keys.

- The **UserInfo Endpoint** allows the client to retrieve additional user profile information using the access token.

By leveraging these mechanisms, OpenID Connect supports key features such as **Single Sign-On (SSO)**, session management, and secure delegation of identity verification across multiple services. This makes it a robust and interoperable protocol for modern identity and access management.

OpenID Connect Flow

The OpenID Connect flow is an extension of the OAuth 2.0 flow and and consists of the following steps:

1. The **client** requests authorization from the **resource owner** by redirecting them to the authorization server's authorization endpoint, including the `response_type=code` parameter to indicate that an authorization code is requested.
2. The **resource owner** authenticates with the **authorization server** and grants permission for the client to access their resources.
3. The **authorization server** redirects the resource owner back to the client with an authorization code and an ID token.
4. The **client** exchanges the authorization code for an access token and an ID token by making a request to the authorization server's token endpoint.
5. The **authorization server** validates the authorization code and issues an access token and ID token to the client.
6. The **client** uses the access token to access protected resources on the resource server and can also use the ID token to verify the user's identity.
7. The **client** can optionally call the UserInfo endpoint to retrieve additional user profile information using the access token.

While OAuth 2.0 and OpenID Connect are the protocols defining the authorization and authentication processes, they primarly address the how of authentication and delegation. The actual permissions and rules that governs the access to resources needs to defined and implemented by access control policies. For this puprose, several policy-as-code languages emerged, which treats policies as structured, declerative artifacts that can be versioned, tested, and deployed like any other code. In the next section, we will explore several frameworks that enable this functionality, leveraging the power of policy-as-code to manage access control in a more efficient and scalable manner.

2.4 Policy-as-code and Authorization languages

Earlier in this chapter we defined authorization policies as a set of access rules that determines the level of access a user/role has to resources. More specific, we used the analogy of a handbook that contains all the rules under which the theater establishment can operate. In modern systems, this analogy is not far from the truth. Every system needs a structured and machine-readable way to define the rules that govern the access

to its resources. So Policy-as-code approach emerged as a way to satisfy this need. There are several languages and frameworks that allow the definition of policies, such as AWS policies and OPA. Each one of these languages has its own syntax and semantics, but they all share the same goal: to provide flexibility and expressiveness in the definition of policies.

2.4.1 OPA - Open Policy Agent

OPA [7] is an open-source policy engine that allows the definition of policies in a high-level declarative language called Rego. Rego is a language that belongs to the broader family of logic programming languages and it is designed to be easy to read and write. It provides a set of built-in functions and operators that allow the definition of complex policies in a simple way. OPA can be used to enforce policies in a variety of systems, such as Kubernetes, Istio, Envoy and many others. It can be integrated with these systems using a REST API, which allows the system to query OPA for policy decisions. OPA can also be used as a standalone service, which allows it to be used in any system that can make HTTP requests. An example of a policy defined in Rego is illustrated at 2.4.1

.

```
package example
default allow = false
allow {
    input.method = "GET"
    input.path = ["v1", "data"]
}
allow {
    input.method = "POST"
    input.path = ["v1", "data"]
    input.user = "admin"
}
allow {
    input.method = "PUT"
    input.path = ["v1", "data"]
    input.user = "admin"
    input.body = {"action": "update"}
}
allow {
    input.method = "DELETE"
    input.path = ["v1", "data"]
    input.user = "admin"
}
```

Listing 2.2: Example of a policy defined in Rego

Policies in Rego use the **input** object to define the conditions under which the action is allowed. Input contains the **method**, **path**, **user**, and **body** of the request. The policy uses the **allow** rule to define the conditions under which the action is allowed. The default rule is set to false, which means that if none of the conditions are met, the action is denied. In this specific example are defined four rules that determine whether

the action is allowed. The first rule allows GET requests to the v1/data path. The second rule allows POST requests to the v1/data path if the user is admin. The third rule allows PUT requests to the v1/data path if the user is admin and the body of the request contains an action field with the value update. The fourth rule allows DELETE requests to the v1/data path if the user is admin. The policy is defined in a **package** called example, which allows the policy to be organized into modules. The package name is used to identify the policy when it is queried by the system.

The use of OPA and Rego is suitable for services that don't have a built-in policy engine, of for those that need a more flexible and expressive way to define policies.

2.4.2 AWS IAM Policies

AWS IAM policies are a way to define permissions for AWS resources. They are written in JSON and consist of a set of statements that define the actions that are allowed or denied for a specific resource. The policies can be attached to users, groups, or roles, and they are evaluated by the AWS IAM service when a request is made to access a resource.

AWS IAM policies are suitable for AWS resources and services, and they provide a simple way to define permissions. We will dive into more details about AWS IAM policies and more specifically about AWS S3 policies in section 2.6.

2.5 Identity and Access Management systems

So far we presented the protocols, languages and frameworks that allows as to define the authorization and authentication processes in a system. Their successful implementation, although, relies on a capable and standard-compliant Identity and Access Management system (IAM). But the question that arises is: "What is an IAM system?". An IAM system is a software solution that provides a centralized way to manage user identities, roles and permissions across multiple applications and services. It allows organizations to enforce security policies, streamline user management, and ensure compliance with regulatory requirements. In modern systems, IAM solutions are essential for user management and access control across distributed environments. In this section, we will explore the concept of Identity and Access Management (IAM) systems and their role in modern applications.

2.5.1 Keycloak

Keycloak [8] is part of the broader category of Identity and Access Management (IAM) solutions. It is an open-source platform that provides comprehensive authentication and authorization services, adaptable to the specific requirements of various applications. Designed with flexibility and extensibility in mind, Keycloak supports standard authentication and authorization protocols such as OAuth 2.0, OpenID Connect and SAML, making it suitable for diverse deployment scenarios.

As an Identity Provider (IdP), Keycloak manages user identities and issues credentials to client applications. It offers features such as Single Sign-On (SSO), identity brokering, multi-factor authentication (MFA) and user federation. Through its fine-grained Role-Based Access Control (RBAC) capabilities, developers can define and manage users, groups and permissions either via a web-based administrative console or programmatically using its robust RESTful Admin API.

A notable architectural feature of Keycloak is its support for realm-based isolation. A realm serves as a self-contained authentication domain with its own users, clients and policies, allowing different applications or tenant groups to operate independently within a single Keycloak instance. This design is especially beneficial in multi-tenant environments with varying security requirements.

In addition to administrative features, Keycloak provides a user-friendly interface for end-users, enabling account self-service tasks such as password resets, profile updates and session management. It is built for scalability and high availability, making it a suitable choice for both small- and large-scale systems.

2.5.2 Keycloak Architecture

Keycloak's architecture is centered around several key concepts that facilitate modular identity management and secure access control. These components enable administrators to structure authentication domains, define clients, assign roles and tailor token content to the needs of the application. The main architectural components of Keycloak as depicted in Figure 2.4 are:

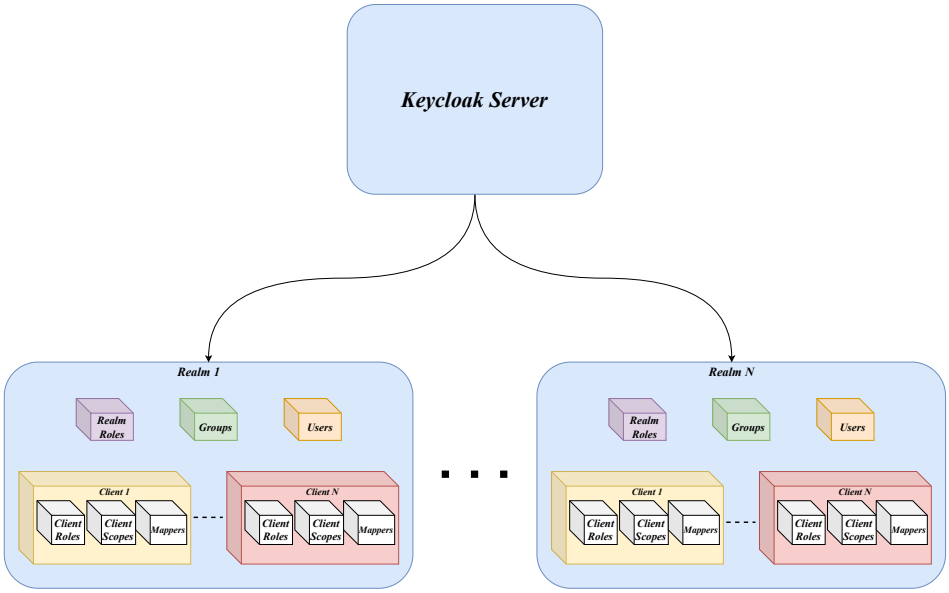

Figure 2.4: Keycloak Architecture Overview

- **Realm**: A realm is an isolated authentication and authorization domain within Keycloak. Each realm maintains its own set of users, roles, clients and configurations. This separation allows multiple applications or tenants to coexist without sharing identity data or security policies.

- **Client**: A client represents an application or service that relies on Keycloak for authentication. Clients are configured within a realm and registered with necessary protocol settings, such as redirect URIs and client secrets. Depending on the use case, clients can be public (e.g., browser-based apps) or confidential (e.g., server-side services).

- **Roles**: Roles define permissions that can be assigned to users or groups. Keycloak supports two types of roles:
  - **Realm Roles**: Global roles available throughout the realm, suitable for defining broad access controls (e.g., administrators, auditors).
  - **Client Roles**: Roles specific to a particular client, used to manage access within the scope of that application (e.g., `viewer`, `editor`, `admin` for a data catalog).
- **Groups**: Groups provide a way to organize users and apply access policies collectively. Assigning roles to groups simplifies permission management, especially in systems with many users, as users inherit the roles assigned to their group.
- **Mappers**: Protocol mappers define how identity and role information is included in the tokens issued by Keycloak. They allow for customization of ID and access tokens by mapping user attributes, roles, or claims into the token payload, enabling client applications to make informed authorization decisions based on token content.
- **Scopes**: In Keycloak, scopes are part of the OAuth 2.0 and OpenID Connect (OIDC) protocols and are used to specify the level of access a client application is requesting from the authorization server. Scopes are typically strings that represent specific permissions or access levels, such as openid, profile, email, or custom scopes defined by administrators. In Keycloak, scopes can be used to trigger the inclusion of specific protocol mappers. Keycloak distinguishes between:
  - **Default Client Scopes**: These scopes are automatically included when a client requests an access token. They are predefined in the realm and can include common claims such as openid, profile and email.
  - **Optional Client Scopes**: These scopes must be explicitly requested by the client using the scope parameter in the token request. Optional scopes allow clients to request additional claims or permissions beyond the default set.

Together, these architectural components make Keycloak a flexible and powerful IAM solution, capable of supporting complex identity and access control scenarios in distributed environments.

2.6 Storage Services

Up to this point it is well established that security of resources is a major concern in modern systems. From user-generated content and transactional records to research datasets and system logs, the integrity, availability and confidentiality of data are critical standards to achieve operational success and trust. So we need storage systems that offer these guarantees. There are many options available, such as Amazon S3, Google Cloud Storage and Microsoft Azure Blob Storage, with each one providing its own set of feautures to satisfy security and performance requirements. In this section, we will explore MinIO, an open source solution that is compatible with the S3 API and designed for high performance and scalability.

2.6.1 MinIO

MinIO [9] is an object storage server which is compatible with the Amazon S3 API [10]. It can be deployed on various platforms, including on-premises, cloud environments and edge devices and its lightweight design makes it a suitable tool for a wide range of use cases, from small-scale applications to large-scale enterprise deployments. MinIO's compatibility with the S3 API allows developers to use existing S3-compatible tools and libraries, making it easy to integrate into existing workflows and applications. Apart from being a high-performance and scalable object storage solution, MinIO also provides IAM-style policies that resemble those used in AWS IAM, allowing fine-grained access control to resources.

2.6.2 MinIO basic components and architecture

MinIO introduces the concept of buckets, which are similar to directories in a file system. Buckets are used to organize and store objects, and each bucket can contain multiple objects. Objects are the individual files stored in MinIO, and they can be of any size, from small text files to large binary files. MinIO also supports versioning, allowing users to keep track of changes to objects over time. This is particularly useful for applications that require data retention and auditing. MinIO's architecture is designed for high availability and fault tolerance. It uses a distributed architecture that allows multiple MinIO instances to work together as a single storage system. This provides redundancy and load balancing, ensuring that data is always available and accessible, even in the event of hardware failures or network issues. An architecture overview of MinIO is illustrated in Figure 2.5.

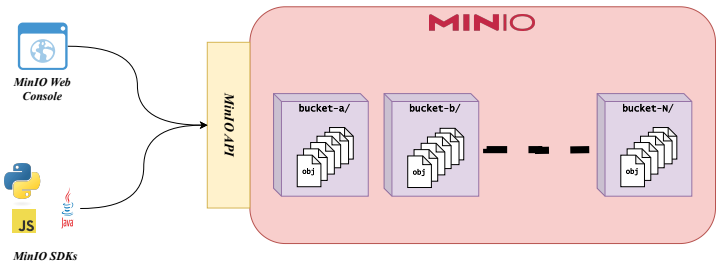

Figure 2.5: MinIO Architecture Overview

2.6.3 MinIO access control

MinIO provides a flexible and powerful access control system based on AWS IAM-style policies. These policies allow administrators to define fine-grained permissions for users and groups, controlling access to specific buckets and objects. It supports both user-based and group-based access control, allowing administrators to assign permissions to individual users or groups of users. This flexibility allows organizations to implement role-based access control (RBAC) 2.2.1 and enforce the principle of least privilege, ensuring that users only have access to the resources they need. MinIO also supports temporary credentials, allowing users to obtain short-lived access tokens for accessing resources. This

is particularly useful for applications that require temporary access to resources without exposing long-lived credentials.

Minio's access control policies are defined using JSON-based policy documents, which specify the actions that are allowed or denied for specific resources. An example policy document might look like this: A policy document consists of a **version**, a **statement**,

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::example-bucket"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::example-bucket/*"
      ]
    }
  ]
}
```

Figure 2.6: Example MinIO Policy Document

and an **effect**. The version specifies the version of the policy language being used, while the statement contains the actual policy rules. Several statements can be included in a policy document, each specifying a different set of permissions. The effect can be either **allow** or **deny**, indicating whether the specified actions are permitted or denied for the specified resources. The action specifies the operations that are allowed or denied, such as listing objects in a bucket or uploading files to a bucket. MinIO supports a wide range of s3 actions, including `s3:ListBucket`, `s3:GetObject`, and `s3:PutObject`. The **resource** field contains specific resources to which the policy applies, such as a specific bucket or object.

In the example presented above, the policy allows the user to list the contents of the bucket named `example-bucket` and to get or put objects within that bucket. This means that the user can view the list of objects in the bucket and upload or download files to and from it.

2.7 Data Catalogs

In large-scale data-driven systems, the ability to organize, describe and discover datasets is just as important as storing them. A data catalog is a foundational component in modern data architectures serving as a centralized registry for datasets and their associated metadata. It provides users with a searchable interface to explore available data assets, understand their structure and provenance and determine their relevance to specific use cases.

Beyond metadata indexing, modern data catalogs often support features such as dataset versioning, tagging, ownership tracking, usage metrics and access control integration. A data catalog is essential for transforming raw data storage into a usable and navigable knowledge layer. It facilitates semantic enrichment, policy enforcement, and cross-service discoverability, enabling both technical and non-technical users to engage meaningfully with the data ecosystem.

2.7.1 CKAN Data Catalog

CKAN (Comprehensive Knowledge Archive Network) [11] is a widely adopted open-source data management system designed to serve as a central data catalog. Initially developed for open government data portals, it has since evolved into a flexible and extensible platform for publishing, managing and discovering datasets in diverse domains. An insight into CKAN's architecture is illustrated in Figure 2.7.

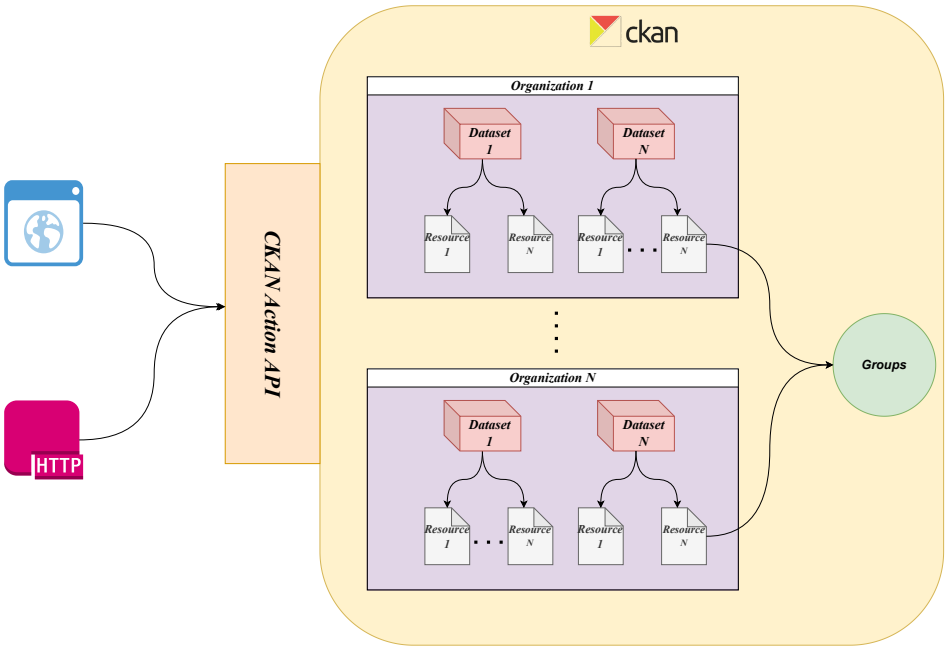

Figure 2.7: CKAN Architecture Overview

CKAN structures data around several key entities:

- **Datasets**: A dataset is the central unit of organization in CKAN. Each dataset represents a collection of related data and includes metadata such as title, description, tags, creation date, license and more. Datasets serve as containers for one or more data resources.

- **Resources**: Resources are the actual data files or data access points (e.g., CSV files, JSON documents) associated with a dataset. A dataset may include multiple resources, each described with its own metadata such as format, size and URL.
- **Organizations**: Organizations group datasets and users under a common administrative scope. They define dataset ownership and enable delegation of roles such as administrator, editor or viewer at the organizational level.
- **Groups**: Groups are thematic or functional collections of datasets, independent of organizational boundaries. They are often used for curating data around specific topics, projects or user communities.

CKAN's internal model is inherently *package-based*: each dataset is technically implemented as a `package` and most of CKAN's extensibility revolves around extending or customizing this core package abstraction. This model underpins how CKAN stores, indexes and renders data catalog entries, and it is leveraged heavily by extensions that introduce new entity types or workflows.

Through this structured and extensible model, CKAN provides an effective way to organize, document and share datasets within and across institutions. It also includes a built-in role-based access control system that regulates who can create, edit or view datasets at both the organization and dataset levels. CKAN provides a web-based interface for users to create, manage and search datasets, along with a robust API for programmatic access.

2.8 Knowledge Graphs

A knowledge graph is a semantic data model that organizes information as a graph, where nodes represent real-world entities (such as people, places, organizations, or concepts) and edges represent the relationships between them (such as *works at*, *located in* or *is a subtype of*). What distinguishes a knowledge graph from a simple graph structure is it's grounding in ontologies—formal vocabularies that define the types of entities and relationships, as well as the rules and constraints that govern them. This enables both humans and machines to interpret the data meaningfully. Knowledge graphs are typically expressed using RDF (Resource Description Framework) and queried using SPARQL, a query language specifically designed for graph data. They support semantic interoperability across heterogeneous data sources, allowing for data integration, enrichment and reasoning. By encoding not only data but also its meaning, knowledge graphs facilitate intelligent applications such as question answering, recommendation systems and data analytics, making them a foundational technology in areas like the Semantic Web, AI, and data integration.

2.8.1 Ontop: An Ontology-Based Data Access System

While knowledge graphs offer a powerful way to model and query data semantically, building and maintaining large, materialized graphs can be costly and redundant—especially when the underlying data already exists in well-established relational databases. Ontop [12] addresses this challenge through Ontology-Based Data Access (OBDA). It allows users to construct virtual knowledge graphs, where semantic access to data is provided without physically replicating it into RDF format.

Ontop achieves this by connecting an ontology, which defines the conceptual model of the domain, with an existing relational database, through a set of declarative mappings. These mappings specify how classes and properties in the ontology correspond to tables and columns in the database. When a SPARQL query is issued over the ontology, Ontop automatically translates it into SQL using the mappings and ontology reasoning. The SQL is executed directly on the database and the results are then presented as RDF triples, as if the data had come from a native knowledge graph.

This virtual approach allows organizations to benefit from the expressiveness and flexibility of knowledge graphs while leveraging existing relational infrastructure. Ontop supports OWL 2 QL, a profile of the Web Ontology Language (OWL) optimized for efficient querying, making it suitable for large-scale data integration, business intelligence and semantic search applications.

2.8.2 Ontop query pipeline

It would be more intuitive to describe the process of querying a knowledge graph using Ontop by following an example scenario. Let's consider we have a relational database containing information about employees, departments and projects in a company. The database schema includes tables like *Employees*, *Departments* and *Projects*, with relevant columns such as *employee_id*, *name*, *department_id* and *project_id*. A table can be represented below:

employee_id	**name**	**department_id**
1	Alice	101
2	Bob	102
3	Charlie	101

Table 2.1: Example Employees Table

In this scenario, we have an ontology that defines classes and properties related to employees, departments, and projects. For example, we might have an *Employee* class with properties like *hasName* and *worksInDepartment* and a *Department* class with a property *hasName*.

The next step is to create mappings that link the ontology to the relational database. These mappings specify how the classes and properties in the ontology correspond to the tables and columns in the database. For instance, we might have a mapping that states:

```
mappingId: EmployeeMapping
target: :Employee/{id} a :Employee ;
    :hasName "{name}" .
source: SELECT id, name FROM employees
```

This mapping indicates that each row in the *employees* table corresponds to an instance of the *Employee* class in the ontology, with the id column mapped to the *id* property and the *name* column mapped to the *hasName* property.

When a user issues a SPARQL query against the ontology, such as:

```
SELECT ?employee ?name WHERE {
    ?employee a :Employee ;
              :hasName ?name .
}
```

Ontop processes this query through several steps:

1. **Query Parsing:** The SPARQL query is parsed to understand its structure and the variables involved.

2. **Mapping Translation:** Ontop translates the SPARQL query into an SQL query based on the defined mappings. For our example, it would generate an SQL query that retrieves the *id* and *name* from the *employees* table:

```
SELECT id, name FROM employees
```

3. **SQL Execution:** The generated SQL query is executed directly against the relational database, retrieving the relevant data.

4. **Result Transformation:** The results from the SQL execution are transformed into RDF triples according to the ontology definitions. Each row in the result set becomes a set of RDF triples representing instances of the *Employee* class with their corresponding properties.

5. **Result Return:** Finally, the RDF triples are returned to the user as the result of the SPARQL query that would look like this:

```
:Employee/1 a :Employee ;
            :hasName "Alice" .
:Employee/2 a :Employee ;
            :hasName "Bob" .
:Employee/3 a :Employee ;
            :hasName "Charlie" .
```

This process allows users to query the relational database using the expressive power of SPARQL and the semantic richness of the ontology, without needing to replicate the data into a separate knowledge graph.

2.9 Data Lakes

Without a doubt, we live in the age of big data. From social media platforms to e-commerce websites, the amount of data generated every second is staggering. Data streams from various sources, including sensors, devices and user interactions, creating a complex web of information that organizations must manage. This information comes in different formats, such as structured, semi-structured and unstructured data and it is generated at an unprecedented velocity. So the traditional data storage solutions such as relational databases are no longer sufficient to handle the scale and complexity of modern data. Relational databases are designed to store structured data in a tabular format, which makes them less suitable for handling the diverse and dynamic nature of big data.

Several solutions have emerged to address these challenges, such as distributed file systems, NoSQL databases and data warehouses. Distributed file systems, such as Hadoop

Distributed File System (HDFS), allow organizations to store and process large volumes of data across multiple nodes in a cluster. NoSQL databases, such as MongoDB and Cassandra, provide flexible schemas and horizontal scalability, making them suitable for handling unstructured and semi-structured data. Data warehouses, such as Amazon Redshift and Google BigQuery, are designed for analytical workloads and can handle large volumes of data efficiently. However, these solutions often come with their own set of challenges, such as complexity, cost and vendor lock-in.

The need for a new approach to data storage and management has led to the emergence of Data Lakes ([13], [14], [15]). Data lakes are designed to store vast amounts of raw data in its native format, allowing organizations to retain all the data they collect without the need for upfront schema definitions. This flexibility enables organizations to store heterogeneous formats of data in a single repository, making it easier to analyze and derive insights from them. They are built on distributed storage systems, such as Hadoop or cloud-based object storage solutions like Amazon S3 or Google Cloud Storage. The ability to handle the scale and complexity of big data makes them the most suitable structure for organizations who need flexible and cost-effective solutions for storing and managing their data. Due to their high scalability they allow the storage and processing of petabytes of data without the need for complex infrastructure. Microsoft Azure Data Lake [16], Google's data lake [17] and data lakes based on AWS S3 are some of the most popular data lake solutions available today.

2.9.1 Data Lakes components

Data lakes are a combination of several layers, with each one playing a specific role in the overall architecture. Every layer is designed to handle a specific aspect of data storage, processing and analysis. The main components of a data lake architecture include:

- **Data Ingestion Layer**: This layer is responsible for collecting and ingesting data from various sources, such as databases, applications and IoT devices. It can handle batch and real-time data ingestion, ensuring that data is continuously fed into the data lake.

- **Storage Layer**: The storage layer is where the raw data is stored in its native format. This layer can be built on distributed file systems or cloud-based object storage solutions, providing scalability and durability.

- **Processing Layer**: This layer is responsible for processing and transforming the raw data into a more usable format. It can include batch processing frameworks like Apache Hadoop or real-time processing frameworks like Apache Kafka or Apache Flink.

- **Security and Governance Layer**: This layer ensures that the data lake is secure and compliant with regulatory requirements. It includes access control mechanisms, encryption and auditing capabilities.

- **Data Access Layer**: This layer provides APIs and interfaces for users and applications to access the data stored in the data lake. It can include SQL query engines, REST APIs and SDKs for various programming languages.

2.9.2 Knowledge Lakes

While data lakes are designed to store large volumes of heterogeneous data in a centralized repository, the concept of a knowledge lake emphasizes semantic organization, contextualization and data discoverability. A data lake typically ingests data in its native format without enforcing a schema. However, this often leads to difficulties in locating, interpreting, and reusing data — a phenomenon sometimes referred to as a **"data swamp"**.

In contrast, a knowledge lake augments the raw storage layer with metadata, classification, relationships and access policies that make data more understandable and actionable. It integrates tools for data cataloging, governance and semantic enrichment, enabling users not only to find datasets but also to understand their origin, structure and intended use. The focus shifts from simply storing data to managing knowledge about the data.

A knowledge lake can be seen as a more advanced and organized version of a data lake, where the emphasis is on making data not just available but also meaningful and usable. The goal is to create a system that not only stores data but also provides context, relationships and insights that can be derived from the data.

For this purpose, knowledge lakes introduce several new layers and components that enhance the traditional data lake architecture.

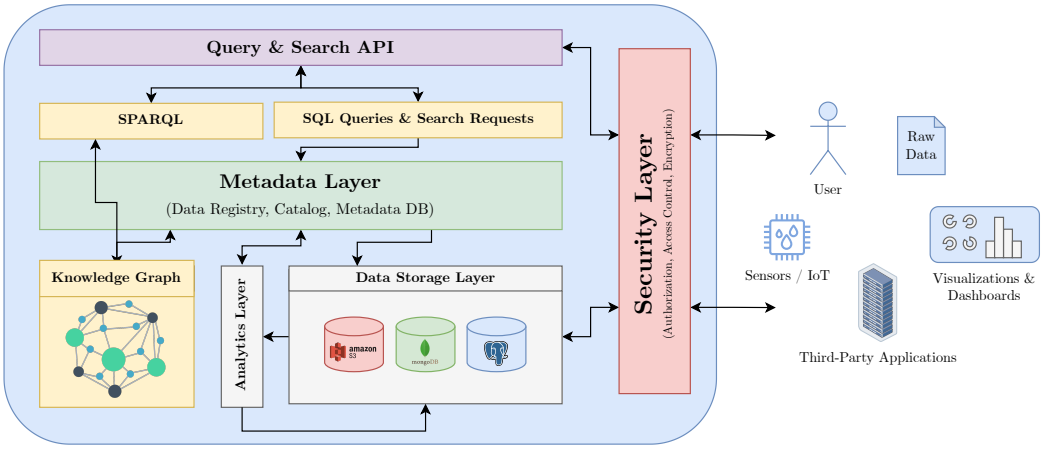

Figure 2.8: Knowledge Lake Architecture

As depicted in figure 2.8, a knowledge lake incorporates the following additional layers on top of the traditional data lake architecture:

- **Knowledge Graph**: The knowledge graph layer provides a structured representation of the relationships between different datasets and entities within the data lake. It enables users to navigate and explore the data in a more meaningful way, facilitating discovery and analysis.
- **Metadata Layer**: It provides metadata management and data discovery capabilities. It helps users find and understand the data stored in the data lake, making it easier to access and analyze.
- **Analytics Layer**: The analytics layer provides tools and frameworks for analyzing the processed data. This can include data warehousing solutions, machine learning platforms and business intelligence tools.

- **Data Visualization Layer**: The data visualization layer provides tools for visualizing and exploring the data stored in the data lake. This can include dashboards, reports and interactive visualizations.

2.9.3 STELAR KLMS

The STELAR(Spatio-TEmporal Linked data for the AgRi-food data space) [18] is a project funded by the European Union's Horizon 2020 research and innovation program. The project aims to develop a knowledge lake management system (KLMS) that elevates the concept of raw data lakes into Knowledge lakes directed towards the agri-food domain. To achieve this goal, STELAR introduces tools and features that enhance the derivation of analytics and insights from the data stored in the knowledge lake. In combination with it's robust and secure platform, STELAR stands out as must-have solution for organizations operating in the agri-food field.

Regarding it's architecture, STELAR does not differentiate itself from the concept of Knowledge lakes. It uses MinIO object storage as the underlying storage layer, a ally when it comes to scalability and performance. As a knowledge lake on it's core, it employs CKAN for the data catalog layer, a powerful service providing metadata management and data discovery capabilities. For a every piece of data stored with MinIO, CKAN provides a metadata schema that describes the data, including its origin, structure and intended use.

When it comes to security and governance, STELAR incorporates Keycloak, who acts as the central IAM system. leveraging keycloak's out of the box support for the OpenID Connect Protocol along side with custom authorization implementations, every user is capable of interacting with the platforms services in a secure and controlled manner, only using one set of credentials.

In addition, STELAR integrates a knowledge graph to provide a semantic layer over its data, enabling enhanced data discovery, interoperability and reasoning. This layer is powered by Ontop, which allows users to execute SPARQL queries over the existing relational infrastructure without data replication. By leveraging ontology-based mappings, STELAR enables efficient and secure semantic access to agri-food data, fully aligned with its access control model.

The one feature that makes STELAR stand out from the rest of the Knowledge lakes is the ability to define multi-steps workflows. These workflows are designed to automate complex data processing tasks, allowing users to define a series of steps that can be executed in sequence or in parallel. Alongside with the rich palette of tools that contains and the ability to communicate with external workflow engines, can be considered a powerful solution for data manipulation and analysis.

Through its rich restful API, users can interact with the platform and perform various operations, such as uploading data, querying metadata and executing workflows. For programming purposes, STELAR provides a Python SDK Client that simplifies the interaction with the platform's API through a rich set of methods, allowing developers to build applications that leverage the capabilities of the platform. Regarding the user interaction, although, the feature that elevates STELAR is the web-based user interface, which provides a user-friendly and flexible way to interact with platform services. The console allows users to easily navigate through the data catalog, search for datasets, visualize data, managing workflows, access analytics tools and manage user accounts with the ease of a few clicks.

The architecture of STELAR is designed to be modular and extensible, allowing for easy integration with other systems and services. A representation of the architecture is illustrated in figure 2.9.

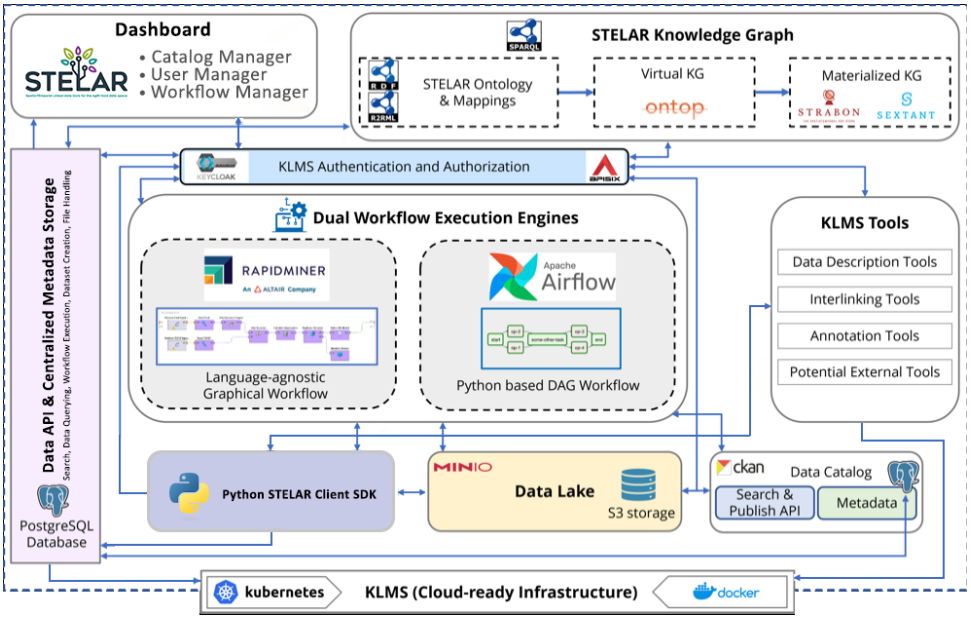

Figure 2.9: STELAR Architecture

2.9.4 Data Catalog in STELAR KLMS

The implementation of the Data Catalog is based on a synergy between CKAN and the additional logic implemented in the context of STELAR KLMS. It leverages CKAN's robust metadata management capabilities while extending its functionality to support the recording of rich metadata, not only for datasets but also for entities specific to STELAR workflows—such as tools and workflow executions. As there is no direct analogy for such entities in CKAN, the Data Catalog had to provide its own support for these entities. One of our decisions was to implement these entities through CKAN extensions. As referenced in section 2.7.1 CKAN's codebase is inherently package oriented, while it offers some facilities for extending the package concept. Building on this extensibility, our design has settled to support four package-based entity types that compose, partially, the entity scheme of STELAR KLMS:

- **Datasets**
- **Tools**
- **Workflows**
- **Processes**

With respect to their schema, these entities have several attributes that are common in all, but also attributes that are specific to each entity type.

Grouping of entities in Data Catalog

As mentioned in section 2.7.1 CKAN allows for grouping the entities accommodated in the system within logical constructs, termed as *groups* and *organizations*. These structures allow for categorizing entities and controlling access to them. STELAR's Data Catalog fully leverages this set of features. In the following Chapters of this thesis, we are going to gain a clear understanding of how the Authorization Design of STELAR KLMS strongly relies on this CKAN's grouping scheme. By utilizing CKAN's native grouping constructs, STELAR is able to implement fine-grained access control policies that align with the system's metadata governance requirements.

Having established a strong foundation of the entities composing the ecosystem of the Data Catalog, we can now provide a complete illustration of their structure within it, as shown in figure 2.10.

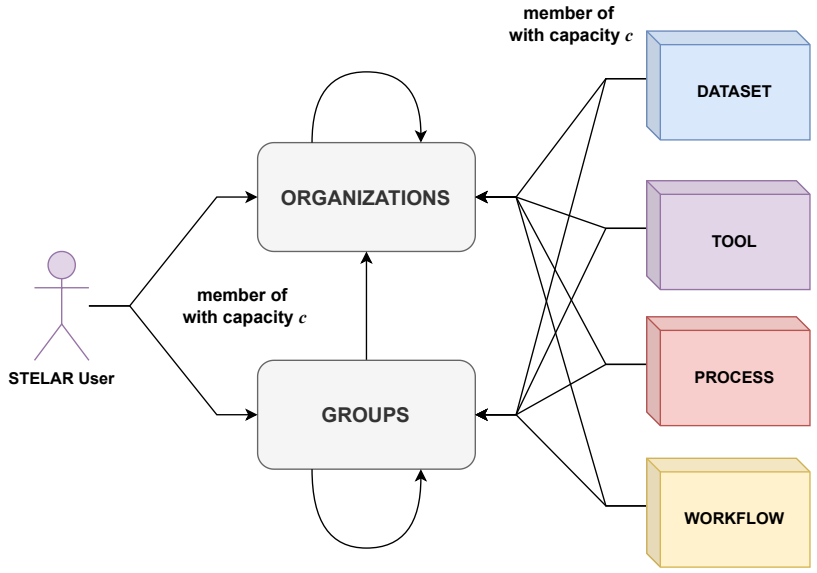

Figure 2.10: STELAR Data Catalog Entities Grouping Structure

2.9.5 Authorization/Authentication in STELAR KLMS

The services that make up the STELAR Knowledge Lake Management System (KLMS) are designed to work together seamlessly, providing a unified platform for managing and analyzing data. However, each of these components originally operated with its own distinct—and often incompatible—authorization and authentication mechanisms. This fragmentation posed a significant challenge for administrators, who were required to manually configure and synchronize access policies and user credentials across multiple services.

On the authentication side, users had to log in separately to each service, resulting in a fragmented user experience and increased overhead for identity management. The users had to manage multiple sets of credentials, which not only complicated the login process but also raised security concerns.

Also ensuring consistent authorization across the system needed precise configurations for Keycloak and the other STELAR services. Manually maintaining these config-

urations was not only time-consuming but also error-prone, leading to potential security vulnerabilities and inconsistencies in access control.

To address these challenges, in the next chapter we introduce the design of a access control system that unifies and streamlines both authentication and authorization across the STELAR KLMS, ensuring constistency, security and administrative efficiency.

Chapter 3

Access Control in STELAR KLMS

3.1 Access Control Design

The design of Access Control in STELAR KLMS, as mentioned in the previous chapter, was driven by the need to unify and centralize authentication and authorization processes across the platform's diverse services. To overcome the operational complexity and security risks introduced by fragmented configurations, the system introduces a declarative, YAML-based policy specification language. This language allows administrators to express high-level access control intents—such as defining roles, assigning permissions and specifying access scopes—in a unified and human-readable format. These policies are automatically translated into service-specific configurations, eliminating the need for manual setup and ensuring consistency across all components. This foundation not only streamlines authorization but also prepares the ground for a cohesive, token-based authentication model that allows users to securely interact with the entire platform using one identity. In the following sections, we will delve into the authentication and authorization mechanisms that underpin the STELAR platform.

3.2 Authentication Management

Authentication management in STELAR KLMS is handled entirely by Keycloak 2.5.1. The IdP stores information on users, groups, roles and every other information needed by each individual service to clarify the user's identity and determine the access rights. This fact, positions Keycloak as a sigle source of truth, eliminating the need for each service to manage users individualy.

3.2.1 Authentication flow

The authentication flow in STELAR KLMS adheres to the OAuth 2.0 and OpenID Connect standards introduced in sections 2.3.1 and 2.3.2. In figure 3.1 is illustrated the authentication flow that takes place when a user tries to login to the KLMS platform.

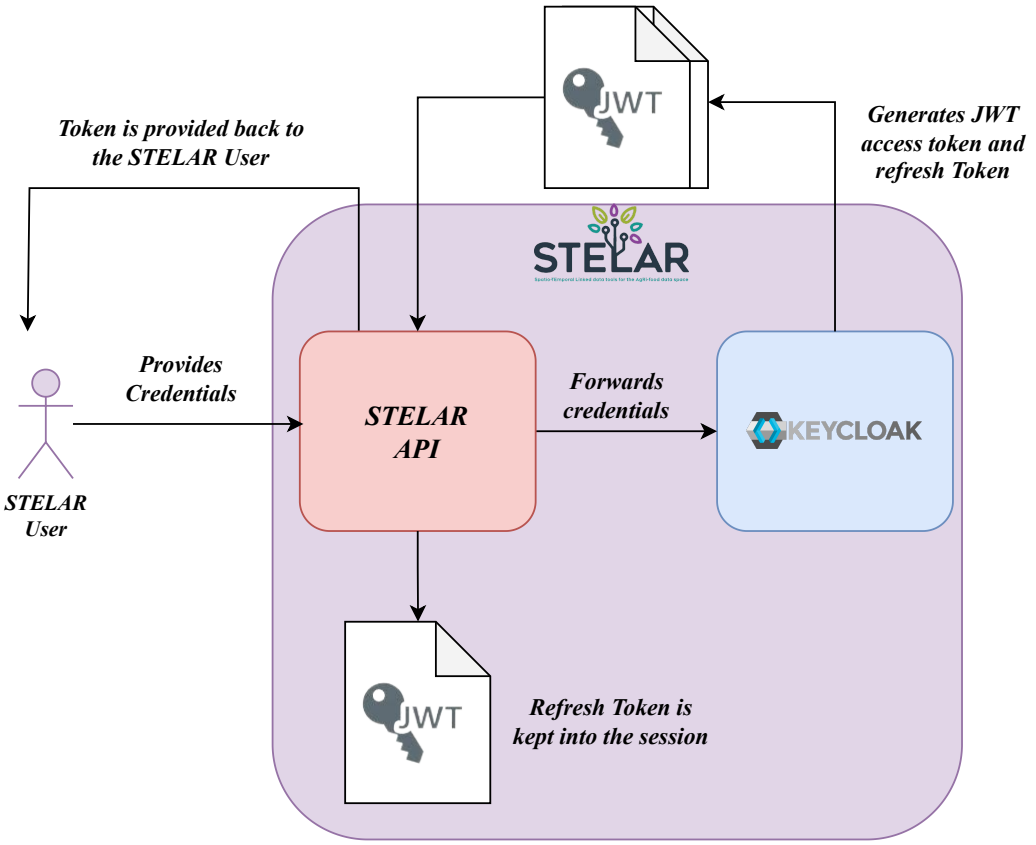

Figure 3.1: Authentication flow in STELAR KLMS

The authentication flow begins when the user provides their **credentials** either through the STELAR console, STELAR Python Client or with a direct request to the STELAR RESTful API. The credentials are then sent to the Keycloak server through the STELAR API, which acts as a proxy between the user and Keycloak. Keycloak verifies the credentials and, if valid, issues a JWT access token. The access token is then sent back to the STELAR API, which forwards it to the user or stores it in an internal session.

In addition to the access token, Keycloak also issues a refresh token. This refresh token is used to obtain a new access token when the current one expires. In contrast to the access token, refresh token is not sent to the user, but it is stored in the STELAR API's internal session. When the access token is close to expiration, the STELAR API uses the refresh token to obtain a new one from Keycloak. Due to its longer expiration time, it allows the user to remain authenticated without having to re-enter their credentials.

In case the user wants to log out, the STELAR API sends a logout request to Keycloak, which invalidates the access and refresh tokens. The STELAR API then clears the internal session, effectively logging the user out of the KLMS platform.

Inside the generated access token are included several claims that provide infromation about the user, such as the user's ID, roles, groups and several other information that are needed by the services to determine the access rights. The user can then use this access token to access the various services of the KLMS platform.

3.3 Authorization Management

In STELAR, we implement an authorization logic based on Role-Based Access Control (RBAC) 2.2.1 The level of abstraction provided by RBAC allows us to group all the service-specific permissions under one high-level role. By assigning a user to a role, we can grant them access to all the resources and actions associated with that role.

Roles can be assigned to users either directly or through their membership in groups. Groups serve as a way to organize users who share similar access needs and they simplify permission management by allowing roles to be applied collectively. In this structure, roles serve as containers for permissions and groups act as containers for roles, creating a flexible yet controlled access model. In STELAR, we have adopted this RBAC model and adapted it to suit our unique operational requirements. Our implementation ensures that access is granted in a way that is both secure and aligned with the practical procedures of our users, giving us a scalable and maintainable approach to permission management. To understand how these roles and permissions come into play during real-time interactions, we now shift our focus to the authorization flow—the process by which access decisions are made whenever a user attempts to interact with a protected resource. This flow illustrates how the system validates user identity, evaluates roles and permissions and ultimately grants or denies access based on the configured rules.

3.3.1 Authorization flow

The authorization process within the STELAR architecture operates as follows:

1. **User Authentication:** Users authenticate via the Authentication and Authorization (AA) system, typically using Keycloak or the embedded login service in GUI.

2. **Role Collection:** Post-authentication, user roles are identified based on direct assignments and group memberships.

3. **Permission Derivation:** Identified roles are used to compute the complete set of permissions available to the user, expressed as action-resource pairs.

4. **JWT Generation:** The AA system generates a JSON Web Token (JWT), embedding user identity and derived permissions as named tags.

Each STELAR service interprets JWT permission tags in its specific context. For instance, services handling file storage (such as MinIO, compliant with AWS S3 API) interpret permissions as actions like GetObject, PutObject, and ListObjects. In contrast, resources such as datasets, workflows and the rest of the metadata entities are using standard CRUD (Create, Read, Update, Delete) operations defined by STELAR API endpoints. Permissions can dynamically reference multiple operations or resource sets using conditions based on JWT attributes (e.g., username) or resource metadata. For example, a permission might allow users to delete only files they personally own. To maintain the integrity and reliability of the authorization flow, it's essential that the underlying policies are consistently defined and applied across all components. This brings us to Authorization Policy Management, which focuses on how STELAR centralizes and streamlines the definition, transformation and distribution of access policies across the system.

3.3.2 Authorization Policy Management

The STELAR KLMS employs a centralized Authorization Policy Management system to streamline the definition, transformation and distribution of access policies across its various components. This system utilizes a declarative YAML-based policy specification language to define the roles and permissions applicable across all STELAR resources and operations. The Authorization Policy Management system is designed to be flexible and extensible, allowing for the addition of new roles, permissions and resources as needed. The system is composed of several components that work together to provide a seamless experience for users and administrators alike.

Policy Management System Architecture

Policy Management System's architecture is simple and easy to understand, yet powerful enough to meet the needs. The system consists of components that are responsible for parsing the policy specifications, generating the necessary configurations, monitoring and updating access policies and enforcing them across the various services of the STELAR KLMS. The overall architecture is illustrated in Figure 3.2.

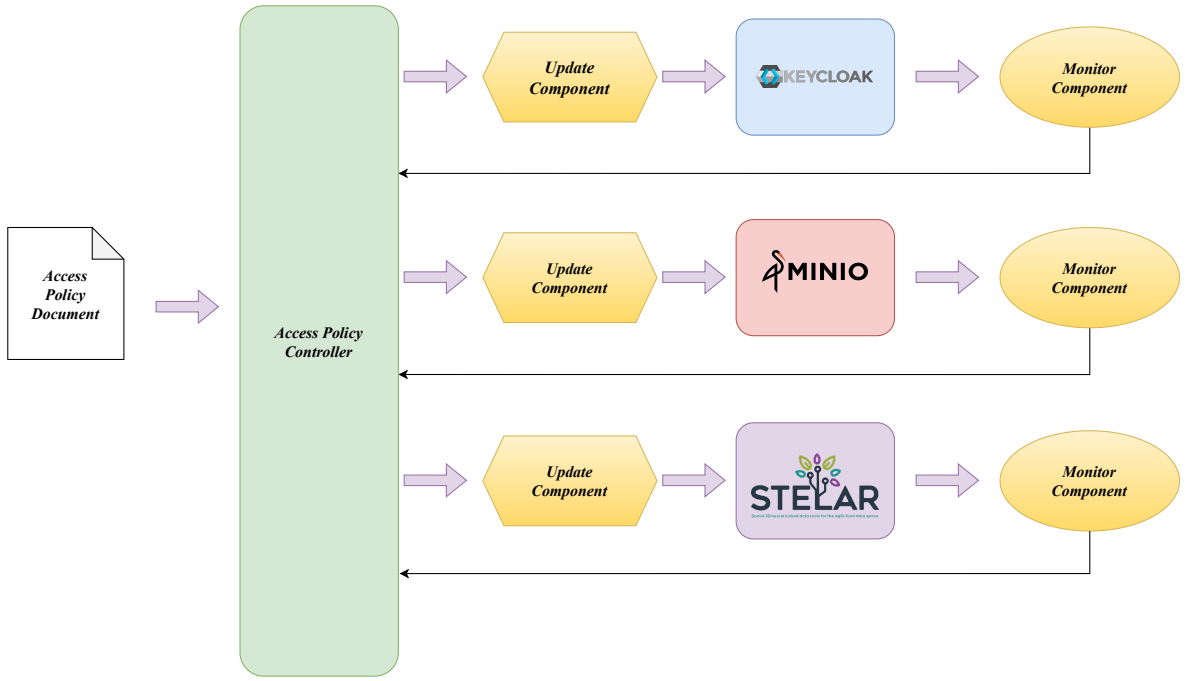

Figure 3.2: System Architecture of the Authorization Management System

As depicted in the figure, the system is composed of four main components with each having its own specific role:

- **Access Policy Document**: This is the input of the system. In this document, the administrator of the system describes the high-level authorization intents, including roles, resources and actions, using the YAML-based policy description language.
- **Access Policy Controller**: The core of the system, responsible for parsing the policy document and generating the necessary configurations for the underlying systems. Additionally, this is where the policy evaluation takes place. Every time a user tries to access a resource, the controller evaluates the policy document to determine whether the access should be granted or denied.
- **Monitor Component**: Monitor component is responsible to keep track the current applied policies for each service. This component works in conjuction with the reconciliation component to ensure that the policies are up-to-date and consistent across all services.
- **Update Component**: This is the component that is responsible for updating the policies if a change is detected in the policy document. It ensures that the policies are always in sync with the current state of the system.

In chapter 4, as we dive more into the functionality of the system, we will explore how each of these components cooperate and communicate with each other, to programmatically orchistrate the authorization flow.

3.3.3 YAML-based Policy Specification Language

A crucial step was to determine a user-friendly and expresssive language to describe the access policies. This language had to fulfill the following requirements:

- **Human-readable**: The language should be easy to read and understand for users, even if they are not experts in programming or access control.
- **Expressive**: The language should be able to express complex access control policies, including conditions and constraints.
- **Extensible**: The language should allow for the addition of new features or constructs as needed in the future.
- **Declarative**: The language should focus on what the policy is rather than how it is enforced, allowing for a higher level of abstraction.

The Rego language 2.4.1 was our first though as it is a language that is expressive and declarative enough to give us the freedom to define complex policies that meet our requirements. However, Rego is not human-readable and requires a steep learning curve for users who are not familiar with programming languages.

To this extent, and in search of a more accessible and expressive policy definition format, we proposed a custom policy description language based on YAML. Before justifying this choice, it is important to introduce the YAML format and explain its key characteristics.

YAML - Yet Another Markup Language

YAML [19] is a widely used data serialization format known for its readability and simplicity. It is commonly adopted in configuration files and for data exchange between systems with heterogeneous structures. By using indentation to represent hierarchical relationships, YAML offers a visually intuitive way to organize complex data. This makes it particularly suitable for non-programmers or domain experts who need to author or inspect structured documents An example of YAML syntax is illustrated in 3.1.

```yaml
# Example of YAML syntax
name: John Doe
age: 30
address:
  street: 123 Main St
  city: Anytown
  state: CA
  zip: 12345
phone_numbers:
  - type: home
    number: 555-1234
  - type: work
    number: 555-5678
  - type: mobile
    number: 555-8765
```

Listing 3.1: Example of YAML syntax

The example above illustrates how YAML can be used to describe a person's information in a clear and structured format. Simple key-value pairs such as name and age are easy to interpret, while the address field showcases nested objects through indentation. The phone-numbers field demonstrates YAML's support for lists of objects, each with multiple properties. This blend of simplicity and expressive capability made YAML a compelling choice for our use case.

Adhering to these principles, we designed a language capable of expressing complex access control policies, avoiding to overwhelm users with unnecessary overhead.

Policy Specification Language Structure

The language provides specific fields designed to support the definition of fine-grained access control policies. Figure 3.3 illustrates the fundamental structure of STELAR's policy representation in YAML format. In this section, we will examine each field in detail, explaining its purpose and how it contributes to the overall policy model.

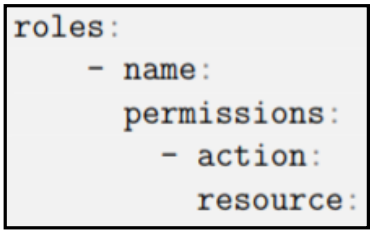

Figure 3.3: Fundamental structure of STELAR's policy YAML

The core component of the policy specification is the **roles** field, where the roles to be created are defined. Multiple roles can be declared, each consisting of a **name** and a set of **permissions**. Permissions are expressed as pairs of **action** and **resource**, indicating what operations a role is allowed to perform and on which group of resources. The structure supports various action and resource specifications, enabling administrators to define access rules that align with the authorization mechanisms required by each individual STELAR component.

Actions and Aliases

In addition to role and permission definitions, the language supports the use of the **actions** field, which allows administrators to define aliases for individual actions or action groups. These aliases act as descriptive identifiers for commonly used operations and can represent either a single action or a collection of related actions. By abstracting low-level operations into reusable and meaningful names, action aliases simplify permission management, enhance readability and promote consistency across policy definitions. Listing 3.2 illustrates the general structure for defining action aliases.

```
actions
    alias_name: ["action1", "action2", ...]
```

Listing 3.2: Definition of actions aliases in Policy YAML

According to the example presented at listing 3.3, readAll alias aggregates multiple actions, specifically: **read_Dataset**, **read_Workflow** and **read_Process**. A role named member is created, with the permission to perform the aggregated action readAll on the specified resources.

```
actions:
    readAll: ["read_Dataset", "read_Workflow", "read_Process"]
roles:
    - name: "member"
      permissions:
        - action: "readAll"
          resource:
            ...
```

Listing 3.3: Example of action aliases in Policy YAML

Once actions and aliases are defined, they become integral to setting up permissions within the platform. This brings us to the next big section of Access Control in STELAR KLMS: Permissions—which combine actions (or their aliases) with resource specifications to determine what operations a role is authorized to perform. We will explore the different types of permissions available in STELAR and how they can be defined utilizing the YAML-based policy specification language.

3.4 Permissions in STELAR KLMS

Permissions define the actions that a role is authorized to perform on the platform's resources. As described earlier, a permission is composed of two main components: an **action** and a **resource specification**. In STELAR KLMS, we distinguish three major types of permissions, each tailored to a distinct component of the platform:

- **Permissions on Catalog/STELAR API resources**
- **Permissions on Storage layer's resources**
- **Special Permissions**

STELAR adopts a **default-deny** (also known as **implicit deny**) access control model. Under this approach, access to a resource is granted only if an explicit permission is assigned. If a permission is not defined for a role, access to the corresponding resource is automatically denied—there is no need to declare explicit denial rules. This ensures a secure baseline where privileges are granted strictly on an as-needed basis. The following sections provide a detailed breakdown of each permission type, including the supported actions, the resource specification schemes and their representation using YAML.

3.4.1 Permissions on Catalog/STELAR API resources

These permissions govern access to the data catalog (e.g., datasets) and STELAR-native entities such as processes and tasks.

Actions

An action defines the operation permitted on a particular entity. In STELAR KLMS, actions are entity-specific—for example, `read-Dataset` and `read-Workflow`. While both actions represent a read operation, they apply to different resource types. In addition to the standard CRUD operations, actions that apply only on specific entities are also supported. Table 3.1 enumerates all supported actions across STELAR entity types.

	STELAR Entities							
Action	Dataset	Workflow	Process	Tool	Group	Organization	Task	User
Create	✓	✓	✓	✓	✓	✓	✓	✓
Read	✓	✓	✓	✓	✓	✓	✓	✓
Update	✓	✓	✓	✓	✓	✓		✓
Delete	✓	✓	✓	✓	✓	✓		✓
Purge	✓	✓	✓	✓	✓	✓		
Edit-membership	✓	✓	✓	✓	✓	✓		✓
Add-member					✓	✓		
Add-task			✓					
Exec				✓				
Terminate			✓					
Edit-roles								✓
Kill							✓	
Read-log							✓	

Table 3.1: Available actions per entity type in the STELAR KLMS platform.

Resource Specification

Each permission includes a resource specification, which identifies the exact set of resources on which the defined actions can be applied. For catalog-related permissions, STELAR supports four types of resource specifications. Some of these leverage the grouping and organizational models described in Section 2.9.4 to support flexible, fine-grained access control.

- **Attribute Resource Specification**
- **Group Membership Resource Specification**
- **Organization Membership Resource Specification**
- **User Membership Specification**

Attribute Resource Specification

This specification targets resources based on metadata attributes. For instance, datasets include attributes such as name, creation date and identifier. Attribute-based filtering allows for defining access conditions such as **datasets created on 2023-10-01**. Figure 3.4 illustrates the structure of this specification.

```
permissions:
  - action:
    resourceSpec:
      - attr:
        operation:
        value:
```

Figure 3.4: Attribute Resource Specification

Attribute resource specification is defined by three fields:

- **attr:** Specifies the resource attribute used for constructing the access rule, such as the name, creation date or description of the resource.
- **operation:** Indicates the operation to evaluate the relationship between the attribute and its value. Operations can be:
  - `equals`: Equal to
  - `like`: Similar to (matching a pattern)
- **value:** Represents the specific value the resource attribute must meet or match.

For instance, if we want to define a permission that allows a role to update datasets created on 2023-10-01, we can use the following YAML code 3.4:

```yaml
permissions:
    - action: "update-Dataset"
      resourceSpec:
        - attr: "creation-date"
          operation: "equals"
          value: "2023-10-01"
```

Listing 3.4: Example of Attribute Resource Specification

Group Membership Resource Specification

This specification identifies resources based on their association with a specific group, as well as their type and functional role (capacity) within that group. Figure 3.5 depicts it's basic structure.

```yaml
permissions:
   - action:
     resourceSpec:
       - type:
         group:
         capacity:
```

Figure 3.5: Group Membership Resource Specification

Like the previous specification, this one is also defined by three fields:

- **group:** Specifies the group to which the resources must belong.
- **type:** Indicates the type of resource, such as Dataset, Workflow, Process, Tool, Group, Organization, Task or User.
- **capacity:** Distinguishes resources within the specified group based on their unique characteristics or roles.

A permission that allows a role to delete packages that are type of *Dataset* and belong to the group *STELAR* under the capacity *mainDataset* can be defined as illustrated at listing 3.5:

```
permissions:
    - action: "delete-Dataset"
      resourceSpec:
        - type: "Dataset"
          group: "STELAR"
          capacity: "mainDataset"
```

Listing 3.5: Example of Group Membership Resource Specification

Organization Membership Resource Specification

Similar to group-based specifications, this type filters resources by their organizational context, including resource type and capacity. See Figure 3.6.

```
permissions:
    - action:
      resourceSpec:
        - type:
          org:
          capacity:
```

Figure 3.6: Organization Membership Resource Specification

Similar to group membership specification, this one is also defined by three fields:

- **org:** Specifies the organization to which the resources must belong.
- **type:** Indicates the type of resource, such as Dataset, Workflow, Process, Tool, Group, Organization, Task or User.
- **capacity:** Distinguishes resources within the specified organization based on their unique characteristics or roles.

The permission presented in listing 3.6, permits patching Datasets belonging to *foo-organization* with the capacity *mainDataset*.

```
permissions:
    - action: "patch-Dataset"
      resourceSpec:
        - type: "dataset"
          org: "foo-organization"
          capacity: "mainDataset"
```

Listing 3.6: Example of Organization Membership Resource Specification

User Membership Specification

This specification defines access based on a user's membership within a group or organization and the user's specific capacity. Figure 3.7 contains the general structure of this specification both for organization and group membership.

```
permissions:
    - action:
      resourceSpec:
        - org:
          capacity:
```

```
permissions:
    - action:
      resourceSpec:
        - group:
          capacity:
```

Figure 3.7: User Organization/Group Membership Resource Specification

Two fields are used to define this specification:

- **org/group:** Specifies the organization or group to which the user must belong.
- **capacity:** Distinguishes users within the specified organization or group based on their unique characteristics or roles.

The permission presented in listing 3.7, allows a user with the capacity mainDataset to read all datasets belonging to the organization fooorganization.

```
permissions:
    - action: "read-Dataset"
      resourceSpec:
        - org: "foo-organization"
          capacity: "mainDataset"
```

Listing 3.7: Example of User Organization Membership Resource Specification

The permission presented in listing 3.8, allows a user with the capacity *mainDataset* to read all datasets belonging to the group *foo-group*.

```
permissions:
    - action: "read-Dataset"
      resourceSpec:
        - group: "foo-group"
          capacity: "mainDataset"
```

Listing 3.8: Example of User Group Membership Resource Specification

3.4.2 Permissions on Storage layer's resources

This type of permission governs access to resources stored in the platform's storage layer. Specifically, it allows for the definition of access rules based on actions and resource targets within MinIO, the system's underlying object storage. Permissions expressed in YAML are automatically translated into corresponding S3-compatible policies, as presented in the example in figure 2.6 at subsection 2.6.3, which are then assigned to user roles. An example of how such permissions are represented in YAML format is shown in Figure 3.8.

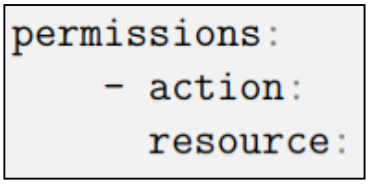

Figure 3.8: Storage Layer Permission Specification

Actions

Actions in storage-layer permissions are based on the S3-compatible API exposed by MinIO. As such, STELAR leverages the same set of operations defined by AWS S3 policies. These include actions such as `s3:GetObject`, `s3:PutObject`, and `s3:DeleteObject`, which correspond to read, write, and delete operations on objects stored in MinIO buckets.

To enhance usability and reduce verbosity in policy definitions, administrators can take advantage of the **action aliases** feature. These aliases map to one or more low-level S3 operations. For example:

- `read` → `s3:GetObject`
- `write` → `s3:PutObject`
- `delete` → `s3:DeleteObject`
- `list` → `s3:ListBucket`

A full list of supported S3 actions can be found in the MinIO documentation[1].

An example of a permission that allows a role to read objects from the bucket *foo-bucket* is shown in listing 3.9:

```
permissions:
    - action:"s3:GetObject"
      resource: "foo-bucket/*"
```

Listing 3.9: Storage layer's permission example

[1] `https://min.io/docs/minio/linux/administration/identity-access-management/policy-based-access-control.html`

3.4.3 Special Permissions in STELAR

Some permissions in STELAR are considered *special* because they are not implemented using the standard YAML-based permission model. Unlike the configurable access rules described in previous sections, these permissions are hardcoded into the platform's logic and are not exposed for user customization. They primarily concern operations related to reading and searching resources, which are fundamental to the usability and navigability of the platform.

Read Permission

The semantics of the read permission vary depending on the resource type:

- **Groups and Organizations**: All users are granted read access by default. This allows them to view basic metadata such as names, descriptions, etc.
- **Package-based resources** (e.g., Datasets, Workflows, Processes, Tools): Read access is granted if the package is marked as *public*. If the package is *private*, the user must be a member of the owning group or organization to view its contents.
- **Resources and Tasks**: Read access is granted only if the underlying package entity to which the resource or task belongs is accessible by the user.

Search/List Permission

Search and List permissions follow a similar model to read permissions:

- **Package-based resources**: These operations returns all packages that are either *public* or belong to a group or organization the user is a member of.
- **Resources and Tasks**: These are included in search results only if their associated package entities are accessible to the user, either by being *public* or through *group/organization* membership.

These special permissions are implemented outside the YAML-based policy specification framework for performance reasons and to maintain compatibility with internal logic already present in various STELAR components. We will revisit these design choices in more detail in the next chapter, where the underlying implementation and architectural considerations of STELAR KLMS are discussed.

3.4.4 Access Control in STELAR's Knowledge Graph (Ontop)

As discussed in Section 2.8, knowledge graphs offer a robust framework for representing complex data relationships and semantics. However, this expressiveness also introduces specific challenges related to access control, particularly in dynamic environments where fine-grained permissions are essential. In the context of the STELAR, it was crucial to ensure that users could only retrieve data they were authorized to access when querying the knowledge graph.

To achieve this, we integrated STELAR's permission model—specifically the read and search permissions—directly into the knowledge graph layer. Our goal was to enforce

these permissions transparently during query execution, so that only authorized data would be returned in response to a user's SPARQL query.

As outlined in Section 2.8.2, Ontop plays a central role in this architecture by mapping the ontology to the underlying relational database. These mappings enable Ontop to translate SPARQL queries into SQL, which is then executed against the original database. Therefore, to incorporate access control, we needed to ensure that these translated SQL queries would inherently respect the user's permissions.

Ontop, however, is not designed to handle arbitrarily complex permission logic directly within its mapping layer—especially when such logic involves multiple joins or conditional access rules. To address this, we adopted a two-step approach:

1. **Pre-filtering via Permission View**: We created a statically defined SQL view that encapsulates the logic of STELAR's read and search permissions. This view does not change per user session; instead, it represents all users and the packages they are authorized to access. It is built using an SQL query that implements the *read* permission model logic and serves as a central access control filter.

2. **Query Execution via Ontop Mappings**: Ontop mappings are then defined to use this permission-aware view as their data source. As a result, when a user issues a SPARQL query, Ontop translates it into SQL that automatically references the pre-filtered view. This ensures that only packages associated with the querying user (as defined in the view) are considered during execution and no unauthorized data is exposed.

This solution ensures that access control is enforced efficiently and consistently, while preserving the benefits of semantic querying offered by the knowledge graph. It also avoids the complexity of embedding permission logic directly into Ontop mappings or dynamically adjusting them at runtime. An overview of this process is illustated in Figure 3.9.

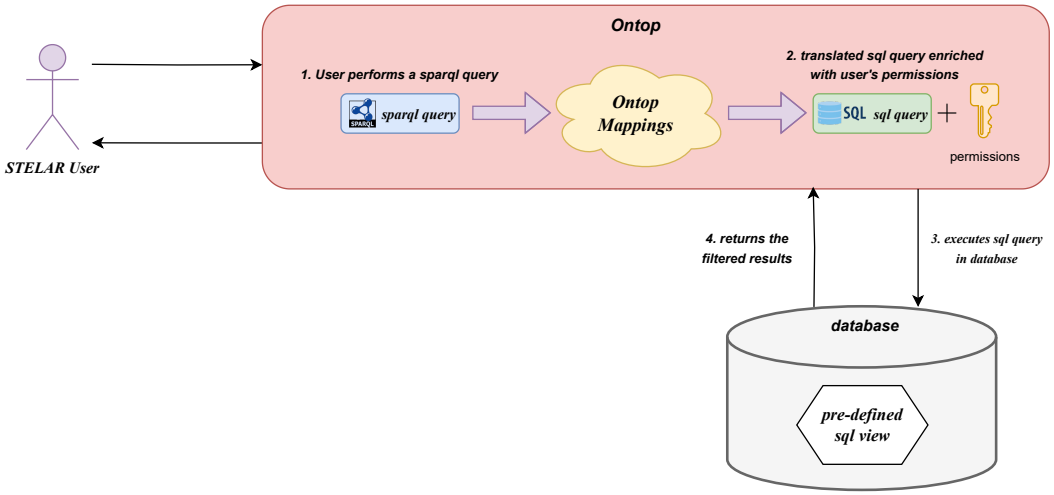

Figure 3.9: Access Control in STELAR's Knowledge Graph

3.5 Policy Reconciliation

In STELAR KLMS the access policy is dynamic and can be updated to adapt to evolving requirements of the knowledge lake and its applications. To facilitate this, STELAR employs a reconciliation-based approach through the Access Policy Controller. This controller continuously monitors the actual access configurations of each STELAR component, it compares them against the desired configurations specified by the current Access Policy Model and afterwards implements necessary updates to achieve consistency and fidelity. Each component in the STELAR ecosystem features its own dedicated driver, consisting of a Monitor and an Update component instance. When the Access Policy Model is updated through a new Policy Document, the Access Policy Controller triggers an evaluation process, retrieves the current configuration for each component, identifies discrepancies and applies the required conversions. These updates might include invalidating existing JWT tokens, removing outdated permissions and introducing new permissions to align with the updated policy. This automated reconciliation process ensures that our access policies remain consistent, secure, and reflective of current operational requirements, aligning with principles of policy reconciliation outlined in prior work ([20], [21]). In Figure 3.10 we illustrate the reconciliation process when a new policy is applied to the system.

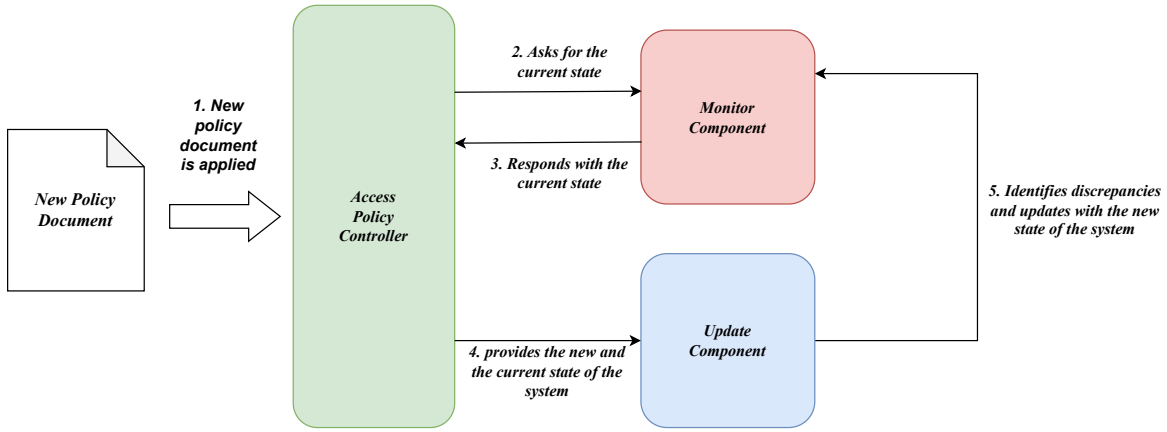

Figure 3.10: Reconciliation process in STELAR KLMS

Chapter 4

Implementation and Evaluation

4.1 Integration with STELAR KLMS

The Authorization Policy Management system 3.3.2 is an integral component of the STELAR KLMS API. It provides a set of endpoints dedicated to managing the lifecycle of access control policies, encompassing their creation, modification and monitoring. Each policy is treated as a discrete entity, characterized by a unique identifier, a descriptive name, a creation timestamp and the policy document itself.

This structured approach facilitates efficient management, retrieval and tracking of policies over time, enabling detailed auditing and historical analysis. All policy-related data—including metadata and the policy definitions—are persistently stored in the platform's database.

Interaction with the policy management system can occur through multiple interfaces: direct API calls, the STELAR Python client or the STELAR console. These options provide flexibility depending on the integration needs and the technical preferences of the user.

Policy evaluation is decoupled from policy management. While policies are stored and maintained within the policy management system, their evaluation is carried out within STELAR's service endpoints. During evaluation, the relevant policy is retrieved from the system and applied to the incoming request context to determine access decisions.

The following sections examine this process in greater detail, including the internal mechanics of policy parsing and the runtime evaluation logic applied during request handling.

4.2 Policy Document parsing and validation

In this section, the parsing and processing of the policy document in STELAR KLMS is described. After a policy document is received by the Policy Controller, a series of operations are performed to apply the defined permissions to the system. These operations differ depending on the type of permissions encountered during parsing. As described in Chapter 3.4, two major categories of permissions exist: permissions related to the storage layer and permissions related to the Data Catalog and STELAR entities. The following sections detail the processing steps for each type of permission.

4.2.1 Policy Document Parsing

The parsing phase begins when a policy document is received through a REST API call to the Policy Controller. The controller first converts the YAML-based policy document into JSON format, which is more suitable for internal processing. Next, two Python class instances are created: one responsible for handling storage layer permissions and another for handling Data Catalog and STELAR permissions. The controller then iterates over the permissions defined in the document. Depending on the presence of a specific field in each permission entry, it invokes the appropriate method on the corresponding class instance. Once parsing is complete, the entire policy document is stored in the database along with several metadata fields, including creation date, creator identity, policy name and a status flag indicating whether the policy is currently active.

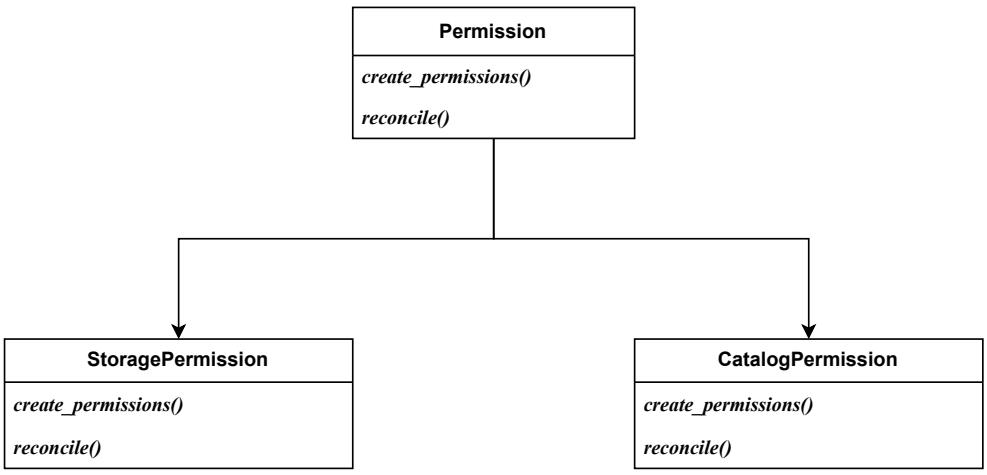

Figure 4.1: Class diagram of the Permission types and their methods

4.2.2 Storage Layer Permissions

Storage layer permissions are identified by the presence of the `resource` field in the policy document. When this field is detected, the `create_permissions()` method of the `StoragePermission` class is triggered. This method communicates with both Keycloak and MinIO to configure the system according to the permission definitions.

Underlying Configurations

During initialization, the `StoragePermission` class establishes a connection to Keycloak using the `keycloak-python` library. A Keycloak client is created with administrative privileges, allowing interaction with Keycloak's REST API. The first step is to create a new realm role in Keycloak, using the value of the name field from the policy document. This role is defined as composite, enabling it to include other roles—functionality that facilitates later association with MinIO policies.

Once the realm role is created, the system proceeds to define policies in MinIO. This begins with iteration over each *action-resource* pair listed under the `permissions` field of the policy document. For every such pair, a JSON policy is constructed using the specified action (e.g., `s3:GetObject`, `s3:PutObject`, `s3:DeleteObject`) and the target resource

(e.g., bucket or object in MinIO). To ensure unique and conflict-free policy naming in MinIO, a hash of the policy content is computed and used as the policy name.

For each policy created in MinIO, a corresponding Keycloak client role is generated with the same name. This client role is then added to the previously created realm role, effectively linking the MinIO policy to the Keycloak role structure. During the evaluation phase, this linkage allows MinIO to correctly resolve which permissions are associated with the authenticated user.

Example of Storage Permission Parsing and Processing

In this section, an example of how a Storage permission is parsed and processed in STELAR KLMS is presented. Let's assume we have a policy document that defines a permission for a specific bucket in MinIO, allowing users to read or get objects from it. The policy document might look like listing 4.1:

```yaml
roles:
  - name: "data-reader"
    permissions:
      - action: "s3:GetObject"
        resource: "my-bucket/*"

  - name: "data-writer"
    permissions:
      - action: "s3:PutObject"
        resource: "my-bucket/*"
```

Listing 4.1: Example of a Storage permission policy document

This document specifies two roles: `data-reader` and `data-writer`, each with associated actions and resources. Whoever has the `data-reader` role can read objects from the `my-bucket` bucket, while those with the `data-writer` role can write objects to the same bucket. When the Policy Controller receives this `document`, it begins parsing it. It starts with the first role `data-reader`. The controller identifies the presence of the resource field and invokes the `create_permissions()` method of the StoragePermission class. Then a composite realm role named `data-reader` is created in Keycloak. After that, the controller iterates through the permissions defined for this role. For the action `s3:GetObject` and resource `my-bucket/*`, a JSON policy is constructed as illustrated in Figure 4.2.

The policy is hashed to generate a unique name, which is then used to create the corresponding policy in MinIO. After the policy is successfully created, a Keycloak client role with the same name is generated and associated with the `data-reader` realm role. The process is repeated for the `data-writer` role, creating a new realm role and MinIO policy for the `s3:PutObject` action on the same resource. The final result is that two roles are created in Keycloak, each with its own **MinIO policy**. The steps of this example are summarized in Figure 4.3.

```
"Version": "2012-10-17",
"Statement": [
    {
        "Effect": "Allow",
        "Action": [
            "s3:GetBucketLocation",
            "s3:ListAllMyBuckets"
        ],
        "Resource": [
            "arn:aws:s3:::*"
        ]
    },
    {
        "Effect": "Allow",
        "Action": [
            "s3:ListBucket"
        ],
        "Resource": [
            "arn:aws:s3:::my-bucket"
        ]
    },
    {
        "Effect": "Allow",
        "Action": [
            "s3:GetObject"
        ],
        "Resource": [
            "arn:aws:s3:::my-bucket/*"
        ]
    }
]
```

Figure 4.2: Example of a JSON policy created in MinIO for the "data-reader" role

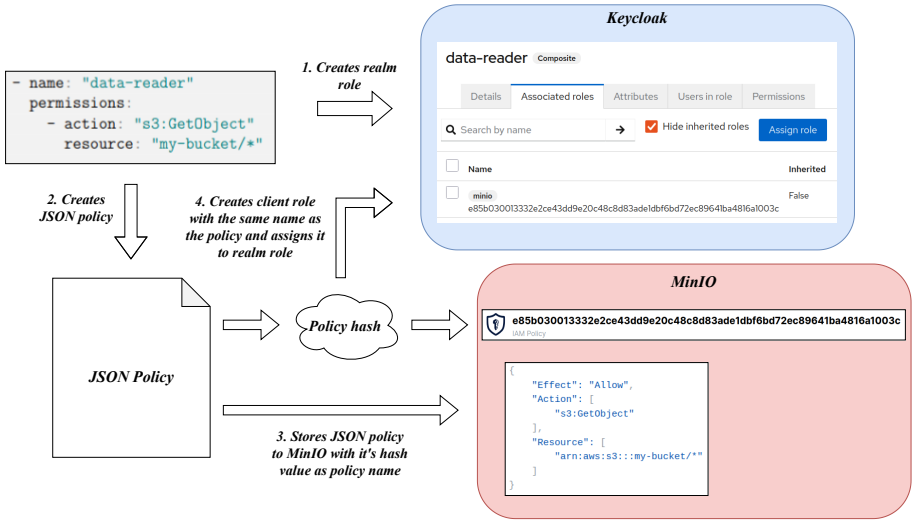

Figure 4.3: Storage permission parsing and processing example

4.2.3 Data Catalog and STELAR Permissions

Data Catalog and STELAR permissions are identified by the presence of the `resourceSpec` field in the policy document. The `DataCatalogPermission` class provides a dedicated method to construct permissions of this type.

Underlying Configurations

The DataCatalogPermission class is initialized with a connection to Keycloak, similarly to the `StoragePermission` class. The first step is to create a new realm role in Keycloak, if it does not already exist, using the **name field** from the policy document. Unlike storage permissions, which result in policies being stored and enforced in MinIO, Data Catalog and STELAR permissions are kept in a **global in-memory structure**. This structure maintains a mapping of actions to roles and corresponding resource specifications. The permission construction method then iterates through each `action-resourceSpec` pair. For every resource specification encountered—as described in Chapter 3.4.1—a corresponding Python class instance is created. The parsing logic is encapsulated in the `parse_resource_spec()` method, which uses structural pattern matching to identify the appropriate class to instantiate. These include `GMspec` for group-based specifications, `UMspec` for user membership constraints, and `AttrSpec` for attribute-based rules. Each instance stores the relevant parameters extracted from the policy document, such as group identifiers, organization names, resource types, capacity requirements or attribute conditions. These instantiated objects are then added to the global permissions dictionary, indexed by action and role name. This dictionary preserves the structure and semantics of the original policy and serves as the source for permission evaluation in later stages.

Example of Data Catalog and STELAR Permission Parsing and Processing

In this section, an example of how a Data Catalog and STELAR permission is parsed and processed in STELAR KLMS is presented. We have a policy document that defines a permission for a specific set of datasets in the Data Catalog. The policy document might look like listing 4.2.

```yaml
roles:
  - name: "data-analyst"
    permissions:
      - action: "delete_dataset"
        resourceSpec:
          type: "dataset"
          org: "stelar-klms"
          capacity: "mainDataset"

  - name: "data-manager"
    permissions:
      - action: "update_dataset"
        resourceSpec:
          attr: "name"
          operation: "like"
          value: "stelar-dataset-*"
```

Listing 4.2: Example of a Data Catalog permission policy document

This document specifies two roles: `data-analyst` and `data-manager`, each with associated actions and resource specifications. The role `data-analyst` is allowed to delete datasets in the `stelar-klms` organization with a capacity of `mainDataset`, while the `data-manager` role can update datasets whose name matches the pattern `stelar-dataset-*`.

Upon receiving the policy document, the Policy Controller initiates the parsing process, beginning with the first role, `data-analyst`. It detects the presence of the `resourceSpec` field and subsequently invokes the `create_permissions()` method of the `DataCatalogPermission` class. A corresponding realm role named `data-analyst` is then created in Keycloak.

Following this, the controller iterates over the permissions specified for the role. For a resource specification defined by the type `dataset`, organization `stelar-klms`, and capacity `mainDataset`, an instance of `GMspec` is constructed, as depicted in Figure 4.4.

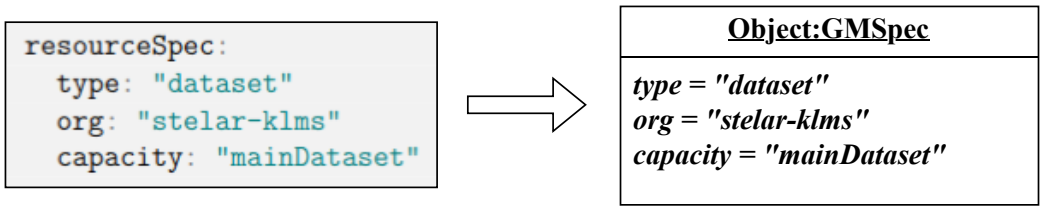

Figure 4.4: Example of GMspec instantiation for the "data-analyst" role

After the GMspec instance is created, it is added to the global permissions dictionary under the action `delete` and role `data-analyst`. The process is repeated for the `data-manager` role, where a `AttrSpec` instance is created for the action **update** with the specified attributes. The final result is that two roles are created in Keycloak, each with its own resource specifications stored in the global permissions dictionary. The global permission dictionary will look like listing 4.3.

```
{
    "delete_dataset": {
        "data-analyst": [
            Object<GMSpec>(type="dataset", org="stelar-klms", capacity="
    mainDataset")
        ]
    },
    "update_dataset": {
        "data-manager": [
            Object<AttrSpec>(attr="name", operation="like", value="stelar-
    dataset-*")
        ]
    }
}
```

Listing 4.3: Example of the global permissions dictionary after parsing

4.3 Policy Evaluation

In this section, the evaluation phase of the policy enforcement mechanism in STELAR KLMS is described. Once permissions are parsed and configured, the system must eval-

uate incoming resource access requests against the defined policies to determine whether access should be granted or denied. Depending on the type of permission, the evaluation process interacts with either the MinIO storage layer or the internal STELAR authorization logic.

4.3.1 Evaluation Entry Point

The evaluation phase begins when a user makes a request to access a resource. The system retrieves the active policy associated with the user and determines the appropriate evaluation logic based on the type of resource and permission. Two evaluation paths exist: one for storage layer resources and one for Data Catalog and STELAR entities.

4.3.2 Storage Layer Permission Evaluation

Requests targeting MinIO resources are evaluated based on policies configured directly in the MinIO system and the user roles resolved via Keycloak. When an access request is made, the Keycloak realm role associated with the user is used to determine which MinIO policies are applicable. The evaluation is delegated to MinIO, which enforces the defined policies based on action and resource matching.

Role and Policy Resolution

When a user attempts to access a MinIO resource, the JWT token issued by Keycloak after successful authentication is presented to MinIO. MinIO uses this token to determine whether access should be granted and which actions are permitted. It validates the JWT and extracts authorization information from specific claims. The claim used for policy assignment is configurable; by default, MinIO looks for a claim named `policy`. This claim should contain a list of policy names corresponding to predefined access policies configured in MinIO. For example, a token containing `"policy": ["read-only", "project-a"]` results in MinIO applying both the `read-only` and `project-a` policies, which define the user's allowed actions on the object store.

This is where the client roles created during the configuration phase come into play. As previously mentioned, each client role is created with a name that matches a specific MinIO policy. A client role mapper is configured in Keycloak to project these roles into the JWT under the custom `policy` claim. When the user authenticates, these roles are embedded into the token and subsequently interpreted by MinIO as policy names. This enables MinIO to enforce role-based access control by dynamically resolving the user's permissions based on their assigned Keycloak roles.

Example of Storage Permission Evaluation

Let's assume we assign a user the role of "data-reader" we created in the example 4.2.2. The JWT issued to this user after authentication contains the following claim:

```
{
  "policy": ["e85b030013332e2ce43dd9e20c48c8d83ade1dbf6bd72ec89641ba4816a1003c
   "]
}
```

MinIO reads this claim and then associates the user with it's stored policy that matches the hash `e85b030013332e2ce43dd9e20c48c8d83ade1dbf6bd72ec89641ba4816a1003c`. When the user attempts to read an object inside "my-bucket", the system checks this policy against the requested action (e.g., `s3:GetObject`) and the resource (e.g., "mybucket/testfile"). If the policy allows the `s3:GetObject` action on the specified resource, MinIO grants access.

4.3.3 Data Catalog and STELAR Permission Evaluation

Requests targeting Data Catalog and STELAR entities are evaluated using the internal permission evaluation framework defined in the Policy Controller. Each permission is represented as a Python object instantiated during the parsing phase.

Authorization Workflow

The evaluation process begins when a user initiates a request to perform an operation on a specific resource, such as deleting a dataset. This request is routed to the corresponding API endpoint, which invokes an `authorize()` function. This function receives three parameters: the requested action, the target resource object, and its type.

If the user is a system administrator, access is immediately granted. Otherwise, the function retrieves the user's roles and looks up the relevant permission rules in the global action-permissions dictionary. For each matching rule, the corresponding resource specification object is invoked through its `auth()` method to determine whether the request should be allowed.

Each resource specification encapsulates fine-grained conditions—such as group or organization membership, resource attributes or user capacity—that are checked against the actual resource. If all conditions are satisfied, access is granted; otherwise, it is denied.

Resource Specification Evaluation

Each resolved resource specification instance implements an `auth()` method, which encapsulates the logic for evaluating whether a user is authorized to perform a specific action on a given resource. The method receives the target resource object as input and compares its attributes against the conditions defined in the active policy during parsing.

The exact evaluation logic depends on the type of resource specification. These may include checks such as group or organization membership, metadata attributes or user capacity within a particular context. Each specification acts as a filter: only if all defined conditions are met does the `auth()` method return `True`, granting access. Otherwise, it returns `False` and the request is denied.

This mechanism ensures that access control decisions are not left to individual components such as CKAN but are fully enforced within the centralized STELAR permission framework. CKAN operates only as a metadata backend, while STELAR policies dictate which actions are allowed at runtime—guaranteeing consistency across the platform's services.

CKAN Data Catalog Integration and Internal User Mapping

All Data Catalog operations are executed via CKAN but controlled exclusively by STELAR. Users do not interact with CKAN directly. Instead, for each STELAR user, an internal "twin" CKAN account is automatically maintained. When a request is authorized—such as creating or deleting a dataset—STELAR invokes the corresponding CKAN API operation using this twin account.

This mechanism ensures proper attribution of actions (e.g., who created a dataset) and enables logging and auditing within CKAN, while fully offloading the access control logic to STELAR. Since CKAN lacks awareness of the full system context, including federated identity and complex policy rules, all CKAN service accounts are assigned administrative privileges. This guarantees that once STELAR authorizes a request, it can be executed in CKAN without triggering internal permission conflicts.

This design encapsulates CKAN as a passive metadata backend, while STELAR serves as the single point of truth for authorization.

Example Evaluation

Suppose a user with the **data-analyst** role attempts to delete a dataset with ID **dataset-1234**. The request is routed to the dataset deletion endpoint, triggering the `authorize()` function. The system retrieves applicable permissions from the global dictionary and matches the action **delete_dataset** against those assigned to the data-analyst role.

It then invokes the `auth()` method of the corresponding **GMspec** object, which checks if the dataset is of type **dataset**, belongs to the organization **stelar-klms**, and has the capacity **mainDataset**. If all conditions match, the action is approved. The deletion operation is then carried out in CKAN using the twin CKAN user linked to the STELAR identity. The example is illustrated in Figure 4.5.

4.3.4 Special Permissions Evaluation

As described in Section 3.4.3, read and search permissions in STELAR are handled outside the standard policy evaluation framework. The primary reason for this design choice is performance scalability.

While the YAML-based model allows the definition of fine-grained access control policies for read operations, enforcing such logic—particularly in high-volume operations like listing or searching resources—introduces considerable overhead. Evaluating multiple conditions per resource (e.g., attributes, group/organization memberships, capacities) becomes computationally expensive when querying large datasets, making it unsuitable for real-time filtering.

To address this, STELAR leverages **Apache Solr**, the full-text search engine integrated with CKAN, to perform efficient metadata-based filtering during read and search operations. Solr provides powerful indexing, faceting, filtering and query optimization capabilities over large collections of structured and unstructured data. CKAN uses Solr to

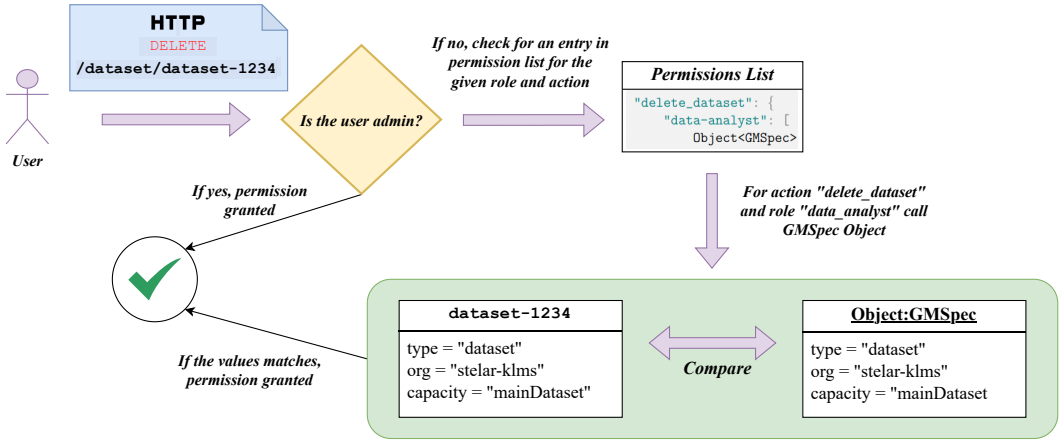

Figure 4.5: Example of policy evaluation in STELAR KLMS. The user attempts to delete a dataset, and the system evaluates the request against the defined policies.

maintain an index of all dataset metadata, including visibility flags (e.g., public/private) and organizational context.

STELAR extends this capability by dynamically constructing Solr queries based on the user's group and organization memberships:

- Public datasets are matched using a visibility flag.
- Private datasets are filtered using the user's group or organization identifiers.

This approach allows scalable and performant enforcement of access rules at the search layer, avoiding costly per-resource evaluations while ensuring consistency with the overall access control model.

Read Permission Evaluation

When a user attempts to read a resource, the `authorize()` function is invoked with the action `read`, the resource type and the target resource object. The function includes logic that distinguishes read operations from standard policy-based actions. Rather than consulting the action-permissions dictionary, it delegates to a dedicated method for evaluating read permissions.

This method first determines whether the resource is public. If so, access is granted immediately. If the resource is private, it retrieves the user's group or organization memberships and verifies whether the user is a member of the organization that holds the requested resource. Access is granted only if such a membership exists. This logic ensures efficient evaluation while maintaining access integrity for private content.

Search Permission Evaluation

When a user performs a search operation, STELAR invokes a dedicated function to construct a Solr query tailored to the user's access scope. This function retrieves the user's group and organization memberships and builds a filter clause using their identifiers:

```
permission_labels:("capacity:public" OR
              "member-organization_id1" OR
              "member-organization_id2")
```

To include public datasets in the results, the filter also incorporates the condition `capacity:public`. If the user is a member of one or more organizations, the query includes clauses for each organization ID, such as `member-organization_id1` and `member-organization_id2`. The final query is executed against the Solr index, ensuring that only datasets the user is authorized to access—either public or private via membership—are returned in the search results.

4.4 Task Execution and Resource Access

In STELAR, access to resources is not limited to direct user interactions. As discussed in Section 2.9.3, one of the platform's core features is the definition and execution of multi-step workflows. Each workflow step corresponds to an automated task that must be executed in a defined sequence.

Tasks are designed to automate various operations, including tool execution, metadata collection for tracking execution context, dataset lineage recording (for outputs produced by tools) and the automated publication of datasets and associated resources in both the Data Catalog and the Storage Layer (MinIO).

During task execution, the system must access input resources required for successful completion. These may include datasets, metadata and files referenced by the workflow. Additionally, upon task completion, the system may need to publish newly generated datasets or files, making them available in the appropriate platform components.

Task execution is asynchronous and can be initiated either by the user or by the system itself as part of a scheduled or chained workflow. Therefore, a secure mechanism is required to grant the executing system components the necessary permissions to access resources on behalf of the user.

To enable this, STELAR passes the initiating user's access token as part of the task execution context. This token encapsulates the user's identity and access rights, allowing the system to perform all operations within the authorized scope of that user.

When accessing resources in the Data Catalog, the system uses the access token to authenticate against the STELAR API, retrieving metadata and permissions associated with the requested entities. For operations in the Storage Layer, the system uses the access token to generate a short-lived, scoped credential that allows temporary access to the required MinIO objects. This credential is generated by resolving the roles associated with the user and deriving the necessary MinIO policies dynamically.

This approach ensures that all task executions are performed securely and traceably, fully respecting the user's access control boundaries and eliminating the need for privileged service-level access.

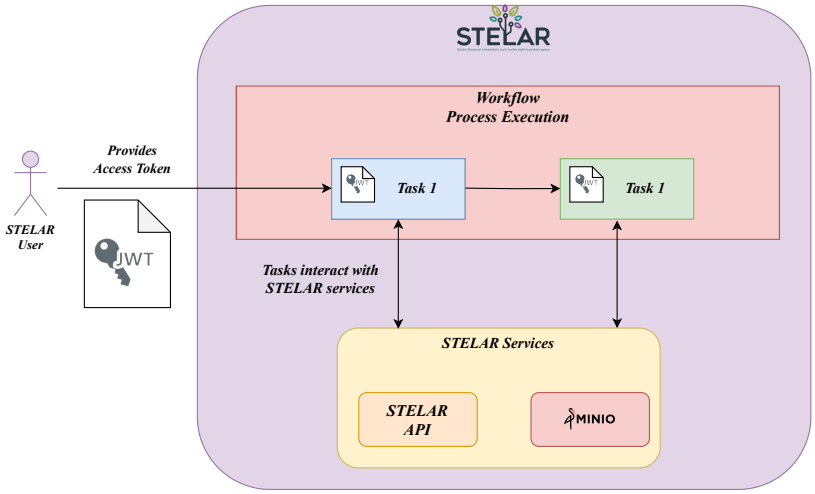

Figure 4.6: Task Execution Process in STELAR

4.5 Implementing Access-Control in STELAR Knowledge Graph (Ontop)

To enforce fine-grained access control in the STELAR knowledge graph interface, we implemented a pipeline that leverages JWT-based user identification and Ontop's dynamic mapping capabilities. The process begins when a user sends a request to a dedicated API endpoint for executing SPARQL queries. This request includes a valid JWT token, which contains the user's unique identifier (UUID) in the `sub` claim. Upon receiving the request, the backend extracts this UUID and assigns it to the `X-User` header parameter. This header is then forwarded to Ontop as part of the query request.

Ontop is configured to retrieve this user identifier through the internal function `ontop_user()`, which resolves the value of the `X-User` header and makes it available within SQL mappings. The mappings are written to filter query results based on the user's permissions. A representative mapping is illustrated in Listing 4.4.

The mapping's source query references a predefined SQL view, which encodes STELAR's access control model. This view unifies both private and public package access logic, associating each user with the packages they are permitted to access. The view is constructed using a SQL query that combines two main components: one for private packages (accessible only to members of the organization) and another for public packages (accessible to all users). The SQL view is defined in Listing 4.5.

When the user's SPARQL query is received by Ontop, it is rewritten and translated into an SQL query based on this mapping. The `ontop_user()` function injects the UUID (from the `X-User` header) into the SQL, filtering the data at the database level. As a result, the final SQL query retrieves only the packages the user is authorized to access, as determined by their membership and the visibility (private or public) of the package. This design ensures secure, efficient, and semantically transparent access control directly within the knowledge graph querying process.

```
mappingId Dataset-basic

target klms:dataset/{id} a dcat:Dataset ;
dct:identifier {id}^^xsd:string ;
dct:title {title}^^xsd:string ;
dct:description {notes}^^xsd:string ;
owl:versionInfo {version}^^xsd:string ;
dct:type {type} ;
dct:license {license_id} ;
dct:creator klms:user/{creator_user_id} ;
dct:publisher klms:organization/{owner_org} ;
dcat:landingPage {url} .

source SELECT id, title, notes, version, type, license_id,
owner_org, private, creator_user_id, url
FROM klms.accessible_packages_per_user
WHERE user_access_id = ontop_user()
AND type = 'dataset'
AND state = 'active'
```

Listing 4.4: Ontop Mapping for Dataset Access Control

```
CREATE OR REPLACE VIEW klms.accessible_packages_per_user AS

-- PRIVATE PACKAGES (only accessible to members)
SELECT
p.*,
m.table_id AS user_access_id
FROM public.package p
JOIN public.member m
ON p.owner_org = m.group_id
WHERE p.private = true
AND m.table_name = 'user'
AND m.state = 'active'

UNION

-- PUBLIC PACKAGES (accessible to everyone, so join with all users)
SELECT
p.*,
u.id AS user_access_id
FROM public.package p
CROSS JOIN public."user" u
WHERE p.private = false;
```

Listing 4.5: SQL View for Package Access Control

4.6 Policy Reconciliation Implementation

The reconciliation process, as described in Section 3.5, is implemented in STELAR KLMS as a structured sequence of operations that ensures consistency and correctness of access control policies across system components. Each permission type class implements a `reconcile()` method, which is responsible for reconciling the permissions of that specific type. The reconciliation logic differs based on the nature of the permission type and is triggered immediately after the parsing of a new policy document. In the following sections, we describe the reconciliation process for each permission category.

4.6.1 Reconciliation of Storage Type Permissions

When a new policy document is parsed, Storage permissions are configured as described in Section 4.2.2. Following the parsing phase, the `reconcile()` method of the `StoragePermission` class is invoked to align the system state with the new policy definition. This method performs the following operations, as presented in Figure 4.7:

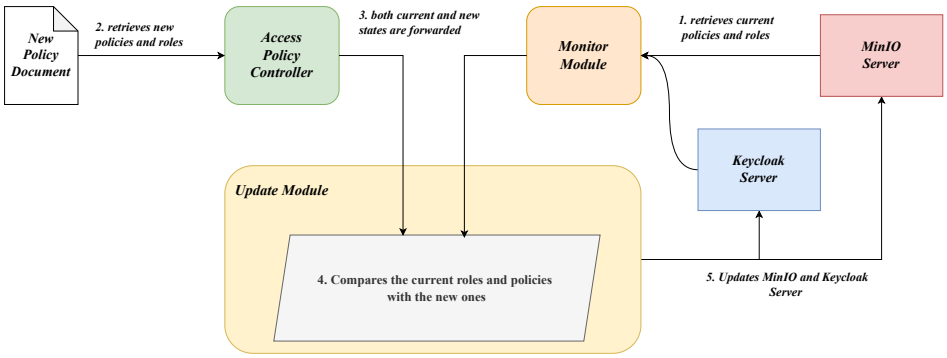

Figure 4.7: Reconciliation of Storage Permissions

1. The Monitor module retrieves the current state of realm roles, client roles and MinIO access policies from the Keycloak and MinIO servers.

2. The Policy Controller extracts the desired state (roles and policies) from the newly parsed policy document.

3. Both the current and desired states are forwarded to the Update module.

4. The Update module compares the two states and identifies discrepancies.

5. Roles and policies that exist in the new policy but are missing in the current system state are created in Keycloak and MinIO.

6. Roles and policies present in the current system state but absent from the new policy are deleted from Keycloak and MinIO.

4.6.2 Reconciliation of Catalog/STELAR Type Permissions

The reconciliation process for Catalog/STELAR permissions is initiated immediately after the parsing of the policy document, as described in Section 4.2.3. The `reconcile()` method of the `CatalogPermission` class is executed and carries out the following operations, as depicted in Figure 4.8:

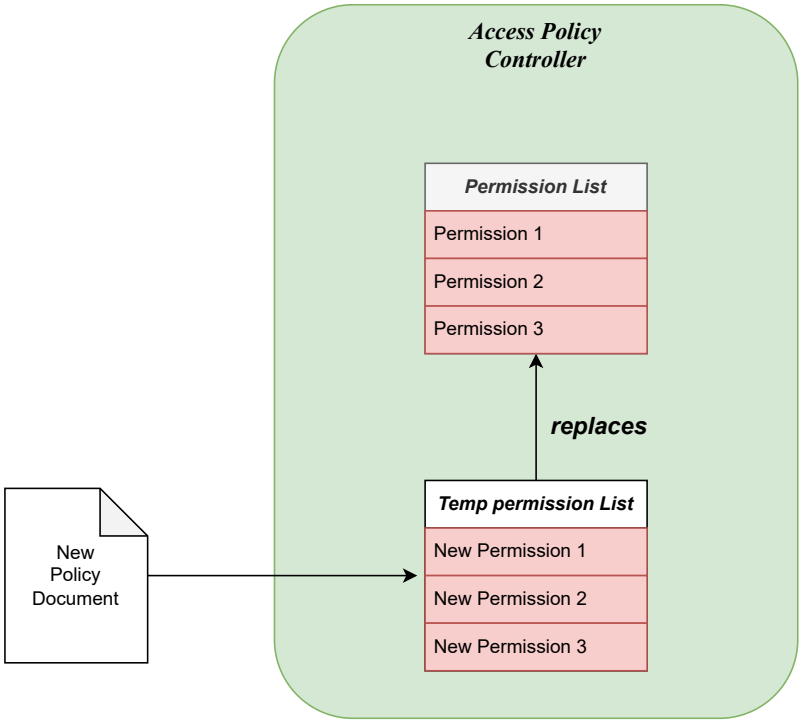

Figure 4.8: Reconciliation of Catalog/STELAR Permissions

1. The newly parsed permissions are stored in a temporary global dictionary, representing the desired state of Catalog/STELAR permissions.
2. Once the parsing and initialization are complete, this temporary dictionary replaces the existing global permissions dictionary (listing 4.3), effectively substituting the previous state with the new one.

This approach ensures atomic replacement of Catalog permissions, reducing the risk of intermediate inconsistencies and guaranteeing that only the most recent policy definitions are enforced across the system.

Chapter 5

Conclusions

In this thesis, we addressed the critical challenge of designing and implementing a unified, secure and flexible access control system for the STELAR Knowledge Lake Management System (KLMS). As data ecosystems grow increasingly complex and distributed, ensuring consistent and fine-grained access control across heterogeneous services becomes essential not only for operational efficiency but also for security and regulatory compliance.

We began by examining the evolution from traditional data lakes to knowledge lakes, highlighting how semantic enrichment and integrated governance mechanisms add significant value in data-intensive domains such as agri-food. Within this context, the limitations of fragmented authentication and authorization models became apparent—each system component managing its own access rules, resulting in configuration inconsistencies, increased administrative overhead and potential security vulnerabilities.

To address these concerns, we proposed and developed a centralized access control framework, built around widely accepted technologies such as Keycloak for identity and role management, MinIO for secure object storage and CKAN for metadata-driven data cataloging. A declarative, YAML-based policy specification language was introduced to empower administrators with a human-readable yet expressive mechanism to define access rules across the platform.

Our implementation supports Role-Based Access Control (RBAC) while also enabling dynamic permission resolution through structured resource specifications. We further integrated automated reconciliation logic to ensure that changes to access policies are propagated consistently and securely across all components.

The evaluation of the system demonstrated its effectiveness in achieving the goals of policy consistency, administrative simplicity and operational scalability. Through token-based authentication and the abstraction of authorization into a centralized controller, STELAR KLMS now offers a secure and maintainable access control model suitable for modern knowledge-driven data infrastructures.

Appendices

Software Specifications

STELAR KLMS Code Base

The source code of *STELAR Knowledge Lake Management System* may be found in the below repositories.

STELAR-EU Organization Repositories:

- STELAR KLMS Data API Repository
  `https://github.com/stelar-eu/data-api`
- STELAR KLMS Core Components Repository
  `https://github.com/stelar-eu/klms-core-components-setup`
- STELAR KLMS Deployment Repository
  `https://github.com/stelar-eu/klms-deploy`

Bibliography

[1] Elisa Bertino et al. "The Challenge of Access Control Policies Quality". In: *J. Data and Information Quality* 10.2 (Sept. 2018). ISSN: 1936-1955. DOI: 10.1145/3209668. URL: https://doi.org/10.1145/3209668.

[2] Dae-Kyoo Kim, Pooja Mehta, and Priya Gokhale. "Describing access control models as design patterns using roles". In: *Proceedings of the 2006 Conference on Pattern Languages of Programs*. PLoP '06. Portland, Oregon, USA: Association for Computing Machinery, 2006. ISBN: 9781605583723. DOI: 10.1145/1415472.1415485. URL: https://doi.org/10.1145/1415472.1415485.

[3] Vincent Hu, David Ferraiolo, and D. Kuhn. "Assessment of Access Control Systems". In: (Jan. 2007).

[4] Vincent C. Hu, D. Richard Kuhn, and David F. Ferraiolo. "Access Control for Emerging Distributed Systems". In: *Computer* 51.10 (2018), pp. 100–103. DOI: 10.1109/MC.2018.3971347.

[5] Dick Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. Oct. 2012. DOI: 10.17487/RFC6749. URL: https://www.rfc-editor.org/info/rfc6749.

[6] *Final: OpenID Connect Core 1.0 incorporating errata set 2*. URL: https://openid.net/specs/openid-connect-core-1_0.html#toc.

[7] *Open Policy Agent (OPA)*. 2024. URL: https://www.openpolicyagent.org/.

[8] *Keycloak IDP*. Version 25.0. 2024. URL: https://www.keycloak.org/.

[9] *MinIO Object Storage*. Version RELEASE.2025-01-20T14-49-07Z. 2025. URL: https://min.io/.

[10] *Amazon S3*. 2025. URL: https://aws.amazon.com/s3/.

[11] CKAN Association. *CKAN - Comprehensive Knowledge Archive Network*. Version 2.9.0. 2024. URL: https://ckan.org/.

[12] *Ontop - Ontology-Based Data Access*. Version 4.3.0. 2024. URL: https://ontop-vkg.org/.

[13] Pegdwendé N. Sawadogo and Jérôme Darmont. "On data lake architectures and metadata management". In: *CoRR* abs/2107.11152 (2021). arXiv: 2107.11152. URL: https://arxiv.org/abs/2107.11152.

[14] Ashish Thusoo and Ben Sharma. *Architecting Data Lakes*. 2016. URL: https://www.oreilly.com/library/view/architecting-data-lakes/9781492042518/.

[15] Fatemeh Nargesian et al. "Data lake management: challenges and opportunities". In: *Proc. VLDB Endow.* 12.12 (Aug. 2019), pp. 1986–1989. ISSN: 2150-8097. DOI: 10.14778/3352063.3352116. URL: https://doi.org/10.14778/3352063.3352116.

[16] Raghu Ramakrishnan et al. "Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 51–63. ISBN: 9781450341974. DOI: 10.1145/3035918.3056100. URL: https://doi.org/10.1145/3035918.3056100.

[17] Chris Olston et al. "Managing Google's data lake: an overview of the Goods system". In: *IEEE Engineering Bulletin* 39 (3) (2016), p. 5.

[18] *STELAR Project*. Funded by the European Commision, HORIZON Europe, GA No. 101070122. 2022. URL: https://stelar-project.eu/.

[19] *YAML Ain't Markup Language (YAML)*. Version 1.2. 2024. URL: https://yaml.org/.

[20] D. Preuveneers, W. Joosen, and E. Ilie-Zudor and. "Policy reconciliation for access control in dynamic cross-enterprise collaborations". In: *Enterprise Information Systems* 12.3 (2018), pp. 279–299. DOI: 10.1080/17517575.2017.1355985. eprint: https://doi.org/10.1080/17517575.2017.1355985. URL: https://doi.org/10.1080/17517575.2017.1355985.

[21] Patrick McDaniel and Atul Prakash. "Methods and limitations of security policy reconciliation". In: *ACM Trans. Inf. Syst. Secur.* 9.3 (Aug. 2006), pp. 259–291. ISSN: 1094-9224. DOI: 10.1145/1178618.1178620. URL: https://doi.org/10.1145/1178618.1178620.