POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Modernization of IoT Monitoring Architecture through Centralized Security, Dynamic Routing and Automation

Supervisor
Prof. Antonio SERVETTI

Candidate
Mathieu TREVES

MARCH 2026

Abstract

The evolution of modern web application architectures has shifted towards decentralized, microservice-oriented patterns that prioritize scalability, security, and maintainability. This thesis details the architectural re-engineering of the PROMET&O platform, a sensor data and survey management system used within the Politecnico di Torino. The pre-existing infrastructure, characterized by redundant localized authentication services and static routing configurations, suffered from significant scalability limitations, security vulnerabilities, and operational inefficiencies.

The primary contribution of this work is the migration from a fragmented deployment to a centralized architecture. This was achieved by decoupling the authentication logic from business applications through the implementation of a centralized Identity and Access Management (IAM) system using **Keycloak**. This centralization eliminated resource redundancy and enabled Single Sign-On (SSO) capabilities, further enhanced by a federation with the institutional Shibboleth Identity Provider (IdP) via SAML 2.0.

Simultaneously, the routing layer was modernized by replacing static NGINX reverse proxies with **Apache APISIX**, a dynamic, high-performance API Gateway. This transition facilitated the move to a "serverless" authentication model, where custom Lua code handle OpenID Connect (OIDC) sessions and authorization at the network edge, effectively removing the need for intermediate authentication servers.

Furthermore, the project introduced a robust automation pipeline. This includes a custom CI/CD workflow using GitHub Actions for consistent deployment and a Python-based orchestration tool for the dynamic provisioning of survey campaigns and QR code generation.

The resulting architecture demonstrates a significant improvement in security, resource utilization, and administrative flexibility, providing a scalable foundation for future development.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, for offering me the opportunity to work on such an interesting and well-structured project. I am deeply grateful for your guidance and for your availability at every stage of this work.

I would like to thank my parents for always being there for me. Thank you for supporting me through a challenging journey and for respecting choices that were not always easy to share. To my mother: thank you for instilling in me a love for knowledge and science from a young age; you guided my path and shaped the person I am today. To my father: you continue to be the example of the person I aspire to be, showing me the true meaning of fatherly love and unconditional support.

To my sisters: thank you for everything you continue to teach me. Despite being the older brother, I still find myself learning from you.

I also want to thank my grandmothers, whose expectations motivated me to see this journey through to the end. A special thought goes to my grandfather, with whom I shared many moments of "real life" over these last years, precious time spent away from the pressure of exams and deadlines.

To my friends, both past and present: in one way or another, you have all shaped who I am today.

A very special thank you goes to Beatrice. You have supported and endured me through every moment. You encouraged me, challenged me, comforted me, and helped me in ways I cannot fully express. This work would likely not exist without you.

Finally, I would like to thank myself, for the resilience to never give up.

"Shoot for the moon. Even if you miss, you'll land among the stars."

Table of Contents

Listings

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In the landscape of modern software engineering, the architecture of a web platform is as critical as its functional code. Legacy systems often suffer from tight coupling between services, where security, routing, and business logic are intertwined. This monolithic approach, while simple to bootstrap, becomes a bottleneck for scalability and a liability for security as the system grows.

The PROMET&O platform, designed to manage sensor networks and student surveys, was originally built using a localized deployment strategy. While functional, the architecture relied on redundant instances of resource-heavy services and static configurations that required manual intervention for every deployment change. Furthermore, an audit of the existing security mechanisms revealed critical vulnerabilities in session management and token handling.

The motivation behind this thesis is to modernize the PROMET&O infrastructure by adopting a Cloud-Native approach. By shifting towards a centralized Identity and Access Management (IAM) framework and a dynamic API Gateway, the project aims to decouple security concerns from application logic. This transition is not merely a technical upgrade but a necessary evolution to ensure the platform's reliability, data integrity, and operational efficiency within an institutional context.

1.2 Objectives

The primary goal of this thesis is to design and implement a scalable, secure, and automated architecture for the PROMET&O platform. This goal is decomposed into the following specific objectives:

- **Architectural Analysis and Remediation:** Conduct a thorough review of

the legacy architecture to identify performance bottlenecks and security risks, particularly regarding session management and credential storage.

- **Centralization of Identity Management:** Replace the fragmented authentication model with a unified Keycloak realm. This involves migrating to a single global instance, externalizing the database for persistence, and implementing Single Sign-On (SSO).
- **Institutional Integration:** Federate the local authentication system with the Politecnico di Torino's Shibboleth Identity Provider (IdP) using SAML 2.0, allowing users to authenticate with their official university credentials.
- **Implementation of an API Gateway:** Replace static reverse proxies with Apache APISIX to enable dynamic routing and offload cross-cutting concerns (authentication, rate limiting) to the network edge.
- **Development of Custom Extensions:** Design and implement custom plugins for both Keycloak (Java SPIs) and APISIX (Lua) to satisfy specific business requirements, such as regex-based role mapping and automated anonymous login flows.
- **Automation of Workflows:** Establish a Continuous Integration/Continuous Deployment (CI/CD) pipeline and develop tools for the dynamic provisioning of survey resources and QR codes.

1.3 Thesis Structure

The remainder of this thesis is organized as follows:

- **Chapter 2: State of the Art** provides an overview of the theoretical concepts and technologies underpinning the new architecture, including Infrastructure as Code (IaC), Identity and Access Management (IAM), and the role of modern API Gateways.
- **Chapter 3: Analysis of the Previous Architecture** details the legacy implementation of the PROMET&O platform. It identifies specific limitations in the NGINX ingress, the decentralized Keycloak setup, and the custom Node.js authentication server, setting the stage for the migration plan.
- **Chapter 4: Design and Implementation: Identity and Security** focuses on the configuration and customization of Keycloak. It details the transition to a global realm, the integration with the Politecnico di Torino IdP, and the development of custom Java SPIs for email validation and dynamic role assignment.

- **Chapter 5: Design and Implementation: Routing and API Gateway** describes the adoption of Apache APISIX. It contrasts the new dynamic control plane with the previous static approach and details the development of custom Lua plugins for handling OIDC sessions and Grafana integration.
- **Chapter 6: Design and Implementation: Refactoring the Repository** covers the modernization of the codebase, the enforcement of security standards, and the implementation of automated CI/CD pipelines using GitHub Actions.
- **Chapter 7: Automated Survey Creation** presents the design of the automation tool used to dynamically provision APISIX resources and generate signed QR codes for survey campaigns.
- **Chapter 8: Conclusion** summarizes the achievements of the project and outlines potential future developments.

Chapter 2

State of the Art

2.1 Identity and Access Management

Identity and Access Management (IAM) is a framework of policies and technologies designed to ensure that the right individuals have the appropriate access to technology resources. In modern distributed systems, IAM has shifted from a periphery concern to a core architectural pillar. Centralized IAM solutions decouple the complex logic of authentication (verifying who a user is) and authorization (verifying what a user can do) from individual business applications.

This centralization provides a "Single Source of Truth" for identities, enabling features like Single Sign-On (SSO) and centralized audit logging, which are critical for both security compliance and user experience. [1]

2.1.1 Keycloak

Keycloak is an open-source identity and access management server developed by Red Hat. It functions as an intermediary server, providing centralized authentication, authorization, and user management for web applications and APIs. By delegating security logic to Keycloak, developers can focus on business functionality rather than implementing bespoke security mechanisms.

Key Capabilities and Technical Features

- **Single Sign-On (SSO):** Enables users to authenticate once and gain access to multiple diverse applications.
- **Identity Brokering and User Federation:** Supports integration with external Identity Providers (IdPs) such as Google or GitHub, and synchronizes with existing user directories like LDAP and Active Directory.

- **Security Protocols:** Implements industry-standard protocols including OAuth 2.0, OpenID Connect (OIDC), and SAML 2.0.
- **Admin Console:** Intuitive administration interface to manage users, roles, clients, and settings.
- **Custom Theme Support:** Customization of login pages, emails, and other UI components.
- **Adapters for Different Technologies:** Easy integration with various technologies like Java, JavaScript, Node.js.
- **Multitenancy:** Supports the creation of isolated "realms," allowing a single instance to manage distinct domains of users and applications.

Authentication Flows

Keycloak supports various OAuth 2.0 flows to accommodate different application scenarios.

Authorization Code Flow [2]: Considered the most secure standard for server-side applications, this flow involves the exchange of a temporary authorization code for access tokens.

- The user accesses the client application and is redirected to Keycloak
- Upon successful authentication, Keycloak redirects back to the application with an authorization code.
- The application's backend exchanges this code for access and refresh tokens.
- The application uses the access token to access protected resources

Resource Owner Password Credentials (Direct Grant) [3]: In this flow, the user provides credentials directly to the client application, which then exchanges them for a token.

- The user provides credentials directly to the client application
- The application sends these credentials to Keycloak
- Keycloak validates the credentials and returns tokens
- The application uses the token to access protected resources

2.1.2 Comparative Analysis of IAM Solutions

While Keycloak is a comprehensive enterprise solution, other open-source alternatives cater to different architectural needs.

Authentik

Authentik is an open-source Identity Provider focused on flexibility and versatility. Unlike Keycloak's strict realm-based multitenancy, Authentik offers a more fluid policy engine and includes a built-in proxy provider. It is often regarded as "lighter" regarding resource consumption compared to Keycloak, though it maintains a rich feature set including detailed event logging and flow-based authentication stages. [4]

Authelia

Authelia is a lightweight authentication server written in Go. It is designed primarily to work in conjunction with reverse proxies (such as Nginx, Traefik, or HAProxy) to provide Multi-Factor Authentication (MFA) and SSO for internal applications. Unlike Keycloak, Authelia traditionally relies on configuration files rather than a comprehensive GUI for management, making it highly suitable for "GitOps" workflows but less extensible regarding complex enterprise federation scenarios. [5]

Feature	**Keycloak**	**Authentik**	**Authelia**
Primary Focus	Enterprise-grade IAM & SSO.	Versatility and Policy Engine.	Lightweight MFA/SSO Proxy.
Persistence	Relational DB (PostgreSQL, MariaDB).	PostgreSQL & Redis.	File-based or SQL.
Configuration	Primarily GUI/Admin Console.	Fluid Policy/Flow Engine.	Configuration-heavy (YAML).
Ideal Use Case	Complex federated architectures.	Resource-constrained flexible setups.	Securing homelabs or internal tools.

Table 2.1: Comparison of Open-Source IAM Solutions

2.2 Modern API Gateways

An API Gateway is a software architectural pattern that serves as the single entry point for an application programming interface (API) or a group of microservices.

It acts as a reverse proxy, accepting all application programming interface calls, aggregating the various services required to fulfill them, and returning the appropriate result. [6]

Beyond simple routing, API Gateways perform cross-cutting concerns that would otherwise need to be implemented in every microservice, including:

- **Traffic Management:** Load balancing, routing based on headers/paths, and rate-limiting.
- **Security:** Authentication offloading, authorization, and protection against vectors such as SQL injection and DDoS attacks.
- **Observability:** Centralized logging, metrics, and distributed tracing.

2.2.1 Apache APISIX

Apache APISIX is a dynamic, real-time, high-performance API Gateway. Built on top of Nginx and OpenResty (Lua), it distinguishes itself from traditional gateways through its architecture.

The primary differentiator of APISIX is its use of **etcd** for configuration management rather than a traditional relational database (like PostgreSQL).

- **Real-time Synchronization:** etcd provides a watch mechanism that allows APISIX nodes to sync configuration changes in real-time (sub-millisecond latency) without reloading.
- **High Availability:** The distributed nature of etcd prevents single points of failure in the configuration plane.
- **Hot-Pluggable Architecture:** Plugins can be dynamically mounted and unmounted without restarting the service.

2.2.2 Other API Gateways

Kong

Kong is one of the most mature open-source API gateways, also built on Nginx and OpenResty. While historically similar to APISIX in its data plane foundation, Kong traditionally relies on PostgreSQL or Cassandra for its control plane data. While robust, this architecture can sometimes introduce higher latency during configuration propagation compared to the etcd-based approach of APISIX.

Traefik

Traefik differs significantly as it is a cloud-native "Edge Router" written in Go. Its primary strength lies in seamless integration with container orchestrators like Kubernetes and Docker. Traefik automatically discovers services and updates routing rules dynamically. While APISIX and Kong are often favored for complex API management policies, Traefik is frequently chosen for its operational simplicity and ease of deployment in containerized environments.

2.3 CI/CD and DevOps Methodology

The implementation of an automated pipeline is the focal point of the **DevOps** methodology, a technical shift aimed at bridging the traditional gap between software development (Dev) and IT operations (Ops). At the heart of this methodology lies the **CI/CD pipeline**, a series of automated steps, designed to transition code from a developer's workspace to a production environment with minimal manual intervention and maximum reliability.

2.3.1 Continuous Integration (CI)

Continuous Integration focuses on the early stages of the software development life-cycle. It requires developers to frequently merge their code changes into a central repository, where each merge triggers an automated sequence of:

- **Building:** Compiling the code and managing dependencies,
- **Testing:** Running unit tests and integration tests to identify bugs or regressions early.
- **Static Analysis:** Inspecting code quality and security vulnerabilities (e.g., linting and vulnerability scanning).

By identifying issues immediately after the code is written, CI reduces the technical debt and integration challenges that typically occur when large volumes of code are merged at the end of the development cycle. [7]

2.3.2 Continuous Deployment (CD)

The CD acronym can carry a dual meaning, representing two levels of the automation process:

- **Continuous Delivery:** This ensure that the code is always in a "deployable" state. After passing the CI stage, the software is automatically built into an artifact (such as a Docker image) and pushed to the repository. However, the final push to the production environment requires a manual trigger.
- **Continuous Deployment:** This is the most advanced stage, where every change that passes the automated testing suite is automatically deployed to the production environment without human intervention.

It follows the steps of continuos delivery by automating the next stage in the pipeline. [8]

2.3.3 Impact on Lifecycle Management

The primary value of CI/CD pipelines lies in the automation of the *feedback loop*. By removing manual execution from the building, packaging, and deployment phases, organizations can minimize human error and ensure a consistent, repeatable process. In containerized environments, such as the one developed in this work, these pipelines are essential for packaging code into immutable images, ensuring that the software behaves identically in development and production.

2.3.4 Workflow Orchestration: GitHub Actions vs. Traditional Solutions

While the principles of CI/CD are tool-agnostic, the selection of an orchestration platform significantly impacts development velocity and infrastructure overhead. For this thesis, **GitHub Actions** was selected as the primary CI/CD engine.

GitHub Actions Overview

GitHub Actions is a modern, cloud-native automation platform that allows developers to create workflows triggered by specific repository events (e.g., `push`, `pull_request`, or `workflow_dispatch`). Workflows are defined using YAML syntax, promoting "Pipeline as Code" practices [9]. Its architecture is built around:

- **Actions:** Reusable units of code (e.g. `actions/checkout`) that simplify complex tasks.
- **Runners:** The execution environments for jobs. GitHub provides hosted runners, but self-hosted ones can be utilized to facilitate direct deployment to the target remote server.

GitHub Actions vs. Jenkins

To justify the adoption of GitHub Actions, it is useful to compare it against Jenkins, a long-standing leader in the CI/CD space.

Feature	GitHub Actions	Jenkins
Integration	Native to GitHub; workflows are triggered directly by repository events.	Requires external plugins and webhooks to sync with version control.
Infrastructure	Cloud-native (managed) with support for local self-hosted runners.	Requires a dedicated server to manage the Master/Node architecture.
Configuration	YAML-based files stored in the `.github/workflows` directory.	Traditionally configured via a Web UI; supports Groovy-based Jenkinsfiles.
Maintenance	Minimal; security and updates are handled by the platform.	High; requires manual management of plugins and security patches.
Learning Curve	Gentle; uses standard YAML and a vast community Marketplace.	Steep; requires familiarity with Groovy scripting for advanced pipelines.

Table 2.2: Comparative Analysis: GitHub Actions vs. Jenkins

Rationale for Selection

The decision to utilize GitHub Actions over alternatives like Jenkins or GitLab CI was driven by three primary factors:

- **Unified Ecosystem:** By using GitHub Actions, the entire development lifecycle, from source code management and issue tracking to automated deployment, resides within a single platform. This eliminates the "context switching" and integration overhead associated with managing an external CI server.
- **Ease of Use and Maintenance:** Unlike Jenkins, which requires significant effort to maintain the underlying infrastructure and plugin compatibility, GitHub Actions provides a serverless experience. This allowed the focus of the thesis to remain on architectural improvements rather than DevOps maintenance.
- **Path-Based Triggering:** GitHub Actions provides native, high-performance support for path-filtering. In a project involving multiple services (Ingress,

Keycloak, NS), this allows for an efficient "monorepo-style" management within the organization, ensuring that only modified services are rebuilt, thereby saving execution time and computational resources.

Chapter 3

Analysis of the previous architecture

3.1 PROMET&O Platform

The architecture of the PROMET&O platform has evolved through multiple iterations, both in the deployment environment and in the service composition.

Initially, the platform utilized a aws-centric deployment, hosted entirely on the cloud using Amazon services. It was then migrated to a on-premise environment hosted on a Politecnico di Torino server.

The primary rationale behind this migration was to regain granular control over infrastructure management. Developing on aws required complex authorizations flows, user management, other than having the students know about the specific software utilized on this platform. Furthermore, the cloud deployment incurred a variable monthly subscription fee based on resource utilization, whereas the local deployment eliminated these operational costs.

Following the migration to the Politecnico di Torino server, the software architecture was redesigned by previous thesis students, resulting in the architecture illustrated in Figure 3.1.

The entire architecture is containerized using Docker. For ease of development and clarity when using Git versioning software, docker compose is used to allow the definition of the containers in yaml files.

The entry point of the system is an NGINX reverse proxy, responsible for routing requests to the appropriate PROMET&O instance. The architecture supports multi-tenancy through multiple replicas of the web application, referred to as namespaces (**NS-XX**), each operating within its own isolated subnet with dedicated supporting services.

Each namespace is composed of the following containers:

1. **NGINX:** An internal reverse proxy routing requests to the Web App, Grafana and Keycloak.
2. **Keycloak:** A dedicated Identity Manager instance managing user identities for that specific namespace.
3. **Web App:** The user-facing frontend application.
4. **Authentication Server:** A custom Node.js service facilitating the connection between the Web App and Keycloak.
5. **Database:** A MySQL instance storing user data and survey responses.
6. **Grafana:** A visualization tool fetching sensor data from external containers.

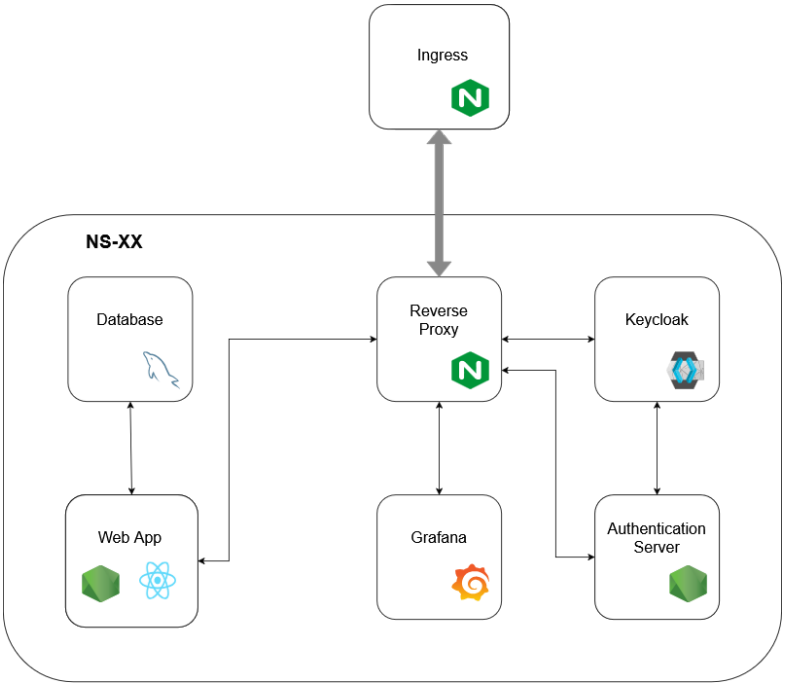

Figure 3.1: Old Architecture Diagram

3.2 Component Analysis and Limitations

3.2.1 Ingress

An NGINX instance served as the global ingress point. This configuration presented several critical limitations regarding scalability and maintainability:

1. **DNS Resolutions Issues:** The container frequently failed to initialize if upstream hosts were unavailable. This was caused by the resolver defaulting to the systemd-resolved address (`127.0.0.53`). Reconfiguring the resolver to the Docker internal DNS (`127.0.0.11`) was necessary to decouple the ingress state from the health of individual namespace containers.

2. **Network Coupling:** The ingress server was required to explicitly join the docker network of the different namespace it served. While this is not directly wrong, it is not ideal. Firstly, all the networks had to be defined in the docker compose of the NGINX instance, as showcased in Listing 3.2.1.

   Listing 3.2.1: Old nsXX networks definition

   ```
   networks:
     nsXX:
       name: nsXX_default
       external: true
     nsXY:
       name: nsXY_default
       external: true
     nsXZ:
       ...

   ```

   And it is clear that, when scaling the architecture and having more than a few namespaces, it would become unmanageable very fast.

   This coupling created two operational risks. First, the ingress container had to be stopped and reconfigured every time a new namespace was added. Second, it violated the principle of least privilege, as the ingress container gained access to internal namespace services (such as databases) that it did not need to reach.

   A superior approach, implemented in the new architecture, utilizes a dedicated `ingress_net`. Only the namespace's internal reverse proxy connects to this network, effectively isolating internal components from the global ingress.

   The Docker Compose configuration illustrated in Listing 3.2.2 is all that is needed to define in the new architecture.

Listing 3.2.2: New ingress network definition

```
networks:
  ingress_net:
    driver: bridge
    name: ingress_net
```

3. **Monolithic Configuration:** Routing logic for all namespaces was centralized in a single `default.conf` file. This lack of modularity made configuration management error-prone. Splitting configuration into per-namespace files (e.g., `nsXX.conf`) allows for better organization and easier maintenance. Repeating configurations can then be condensed in generic files and imported where needed.

3.2.2 Internal NGINX

Within each namespace, a secondary NGINX instance acted as a local reverse proxy. While this component functioned correctly, it was replaced in the new architecture by APISIX, to leverage advanced dynamic configuration capabilities.

3.2.3 Keycloak

The legacy architecture implemented a decentralized identity manager strategy, deploying a standalone Keycloak instance per for each namespace. This approach resulted in several inefficiencies:

- **Resource Overhead:** Keycloak is a Java-based application with a significant memory and CPU footprint. Running multiple idle instances scales resource usage linearly, whereas a single instance can manage multiple clients with negligible marginal overhead.
- **Operational Complexity:** Updates, patches, and configuration changes had to be applied to every instance individually. This often led to configuration drift, where different environments ran slightly different versions or security policies.
- **Security Fragmentation:** Security policies (e.g., password complexity, OTP requirements) were not centrally enforced. A misconfiguration in a single instance could compromise that specific namespace, increasing the overall attack surface.
- **Poor User Experience:** The lack of a centralized realm prevented Single Sign-On (SSO). Users operating across multiple namespaces were forced to maintain separate accounts and re-authenticate when switching environments.

3.2.4 Main web application

This web app is the main component of the platform, containing the graphical interface of PROMET&O. It is composed by a React frontend, and an Express backend, the former that holds all the pages that a user can view, like surveys and a dashboard for all the sensor's data, and the latter that allows to interact with the database, managing the surveys and the user's information.

This part of the platform has remained essentially the same, with some changes mainly in the backed, to integrate it with the changes done to the rest of the architecture.

3.2.5 Authentication Server

Each namespace included a Node.js server, using `Passport.Js`, an authentication middleware, to connect the frontend and Keycloak. This server managed oidc sessions and enforced route permissions. While functional, an audit of the implementation revealed critical security vulnerabilities:

- **Weak Session Signing:** The Express session middleware utilized a hardcoded secret (`my_secret`). This trivial key strength makes the session signatures susceptible to brute-force attacks, potentially facilitating session hijacking.
- **Unsigned Cookies:** The server relied on unsigned cookies for authorization checks. Consequently, a malicious user could modify the cookie payload in the browser to elevate their privileges or spoof roles.
- **Privilege Escalation Risk:** The `/auth/key` endpoint performed a login using hardcoded Administrator credentials and leaked the Admin Access Token via the `user_info` cookie. This exposed the system to total compromise, as any user could extract this token to delete users or modify the realm configuration.
- **Token Exposure (XSS):** The `user_info` cookie containing the full OIDC token set lacked the `httpOnly` flag. This made the tokens accessible to client-side JavaScript, leaving the application vulnerable to Cross-Site Scripting (XSS) attacks where an attacker could exfiltrate the user's session.

3.2.6 MySQL Database

The database is a container using a MySQL image. It presented no obvious problems, so it has remain unchanged.

3.2.7 Grafana

Grafana is a platform for data visualization that can fetch data from multiple data sources and where the user can build various dashboards to present, in a clear and simplified way, information about multiple services. In this project, it is used to retrieve data from an external container that aggregates the main sensor data. It was not something modified in this thesis's work.

3.3 Migration plan

Given that three active namespaces, `nso1` to `nso3`, were already in production, a phased migration strategy was adopted. A new development namespace, `nso5`, was established to prototype the architecture, which will subsequently be promoted to a production environment, `nso4`.

The legacy namespaces remain unchanged to ensure service continuity. The new global Keycloak instance operates in parallel with the legacy environments, currently serving only the new architecture. The ingress server was updated to route traffic to both legacy and modern namespaces simultaneously, enabling a hybrid operation mode until a full migration can be executed.

3.4 New Architecture

In redesigning the architecture, the overall structure was maintained, utilizing a single ingress point that routes traffic between distinct, identical namespaces. The primary architectural modifications were:

- **Externalizing Keycloak:** Utilizing a single, global instance to centrally manage authentication across all services.
- **Adopting APISIX:** Replacing the internal NGINX server with APISIX, which allows for more granular configuration and direct authentication management.
- **Removing the authentication server:** Its role has been fully assumed by the APISIX gateway.
- **Survey Pipeline:** Adding a pipeline to streamline survey creation, automatically managing resources, routes, and Keycloak configuration

Besides these primary changes, significant work was undertaken on the client and ingress server to resolve existing issues and adapt them to the updated service architecture.

Figure 3.2 illustrates the reworked architecture, while also highlighting the different namespaces and docker networks.

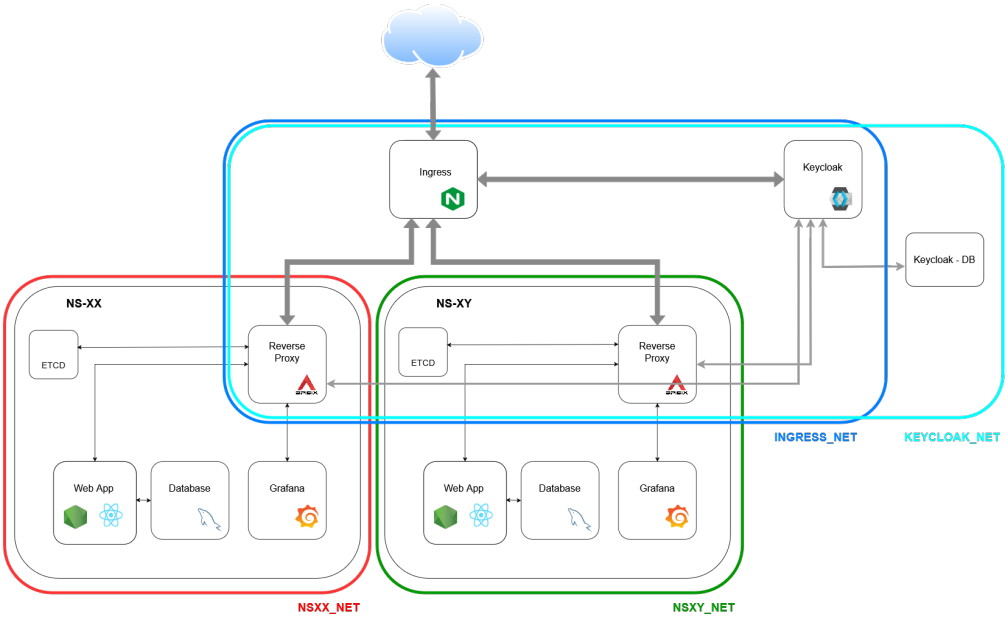

Figure 3.2: New Architecture Diagram

Chapter 4

Design and Implementation: Identity and Security

As previously stated, the old architecture relied on Keycloak for user authentication, a highly extensible open-source Identity and Access Management solution. While this technology choice was maintained, the deployment strategy underwent a significant redesign to resolve various issues present in the previous iteration.

4.1 From multiple instances to a single realm

A pivotal shift in the new architecture is the consolidation of a identity management in a single, global Keycloak instance, configured with a unified realm named *Prometeo*. In this topology, each namespace is registered as a distinct oidc **Client** within the realm, rather than a separate realm.

This consolidation offers two distinct advantages:

- **Centralized Administration:** Administrators can manage users, groups, and global policies in one location, rather than maintaining disparate configurations across isolated environments.
- **Single Sign-On (SSO) Capability:** By sharing a parent realm, users can authenticate only once against *Prometeo* and gain access to all authorized namespaces. This was later improved with the implementation of the Politecnico di Torino as Identity Provider, to much enhance the SSO options.

4.2 Network Configuration

The updated deployment required precise network configuration to function correctly behind a reverse proxy. Keycloak utilizes a dual-path communication model involving both **front-channel** (User Agent → Keycloak) and **back-channel** (Service → Keycloak) requests. [10]

- **Front-Channel:** The user's browser interacts with Keycloak via the public internet. This requires the instance to be aware of its public frontend URL to generate valid redirect URIs and satisfy Cross-Origin Resource Sharing (cors) policies.
- **Back-Channel:** Internal services communicate with Keycloak directly. To minimize latency and avoid hair-pinning traffic through the external internet, these services utilize the internal Docker DNS (container names) rather than the public URL.

This configuration aligns with the architectural decision to implement **TLS Termination** at the ingress point. By offloading SSL/TLS encryption to the ingress container, internal traffic between services is transmitted via standard HTTP. This reduces the computational overhead of encryption for internal services. [11] Security is maintained through strict **network segmentation**, where services are isolated in dedicated Docker networks and are inaccessible from the outside world except through the guarded ingress routes. In particular, Keycloak and its DB are inside a dedicated *keycloak_net* network, to which only the ingress server and the various APISIX instances can connect.

4.3 Database Externalization and Persistence

The redesign began at the orchestration level. A critical architectural correction was the decoupling of the database from the application container. The previous setup relied on Keycloak's embedded H2 database also for production data that, while not explicitly discouraged by the developers, is suggested to be replaced by a more production-ready database [12].

To ensure data integrity, **PostgreSQL** was selected as the external database backend. This transition to a mature, acid-compliant relational database offers two primary advantages:

- **Persistence and Durability:** By mapping the database to a persistent Docker volume, user data and configurations survive container recreation and updates.

- **Operational Management**: An external database simplifies backup and recovery procedures, allowing independent maintenance fo the state layer without disrupting the application layer.

4.4 Version Upgrades and Compatibility

Given that the legacy architecture utilized an outdated version of Keycloak (v. 23.0.7), the system was also upgraded to the latest available stable release (v. 26.4). Regular updates are essential for patching Common Vulnerabilities Exposures (CVEs), and adding new functionalities; however, major version updates often introduce breaking changes that can lead to problems.

The primary challenge during this migration was **plugin deprecation**. Two custom plugins used for theme customization and email whitelisting were incompatible with the new architecture.

- **Theming:** The customization issue was resolved by upgrading the already used theming engine, **Keycloakify**, to its latest version, ensuring compatibility with the updated Keycloak templates.
- **Email Whitelisting:** The legacy email whitelisting plugin was incompatible with the new architecture. While Keycloak's native "Declarative User Profile" feature provided an initial solution for regex-based validation, it lacked the specific business logic required for the final implementation. Consequently, a custom plugin was developed to bridge this gap, as detailed in the following subsection.

4.5 Custom Functionality and Extensions

While Keycloak offers extensive configuration options out of the box, the specific business rules of the *Prometeo* platform required custom logic for added capabilities, specifically regarding strict domain validation and automated role assignment on user registration, based on email patterns.

To address this, a custom extension module was developed using Keycloak's Service Provider Interface (SPI) framework. The extensions were implemented in Java 21 and packaged as a single artifact `prometeo-keycloak-extensions.jar`, deployed in the container provider directory.

The relevant source code for these extensions is available in Appendix A.1.1 and A.1.2.

4.5.1 Keycloak Authentication Flows

To understand the placement of the custom validation logic, it is first necessary to define the **Authentication Flow** architecture. In Keycloak, a flow is a customizable state machine that governs the sequence of interactions required for specific events, such as login or registration.

These flows are composed of multiple **Executions**, atomic steps that perform tasks like credential verification or CAPTCHA validation. By leveraging this pipeline-based architecture, it is possible to inject custom logic into the lifecycle without modifying the core server code. The custom validator described below is specifically plugged into the *Registration Flow*, ensuring it acts as a mandatory gatekeeper before any user data is committed. [13]

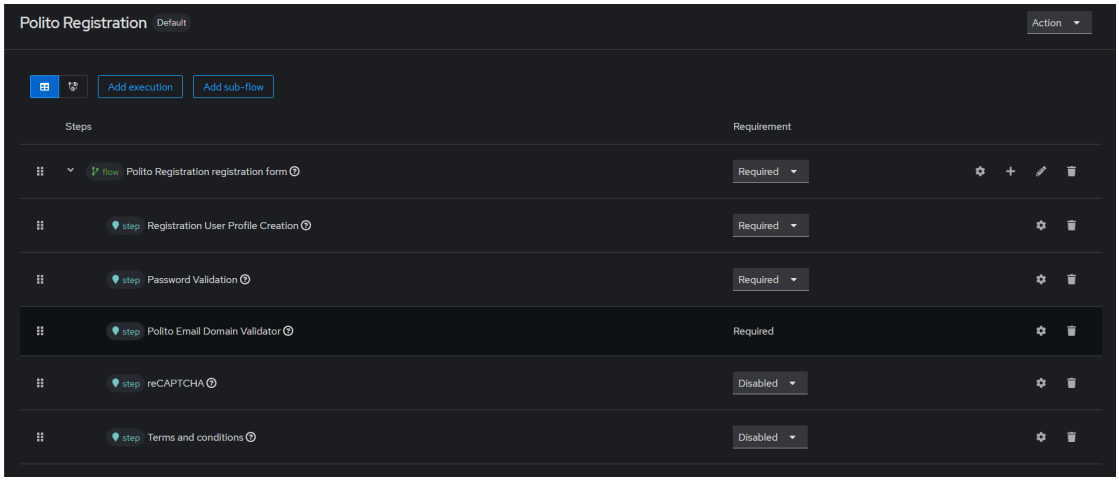

Figure 4.1: Keycloak registration flow page

4.5.2 Domain-specific Email Validation

To enforce strict access control at the system's entry point, it was necessary to restrict registration exclusively to users possessing a valid institutional account. While Keycloak offers basic validation capabilities, they are often global or limited in scope. To address this, a custom **FormAction** SPI implementation, named `PolitoEmailValidator`, was developed. Unlike standard validators, this component is integrated directly into the Registration Flow, allowing administrators to enforce constraints specifically on new user creation without affecting existing users or other authentication flows.

This component provides several key architectural advantages:

- **Pre-persistence Validation:** By implementing the `validate` method of the

`FormAction` interface, the code intercepts the form submission context. It validates the email address against the defined rules before the user entity is persisted to the database, effectively preventing "dirty data" from entering the system.

- **Dynamic Configuration:** The class implements the `Configurable` interface and overrides `getConfigProperties`. This exposes the configuration parameters—specifically the validation Regex and the error message—directly to the Keycloak Admin GUI, as seen in figure 4.2. This design allows administrators to update validation logic (e.g., adding a new subdomain) at runtime without the need to recompile or redeploy the Java artifact.

By default, the validator enforces the pattern `.*@(studenti.)?polito.it$`, ensuring that only official `@polito.it` or `@studenti.polito.it` addresses are accepted.

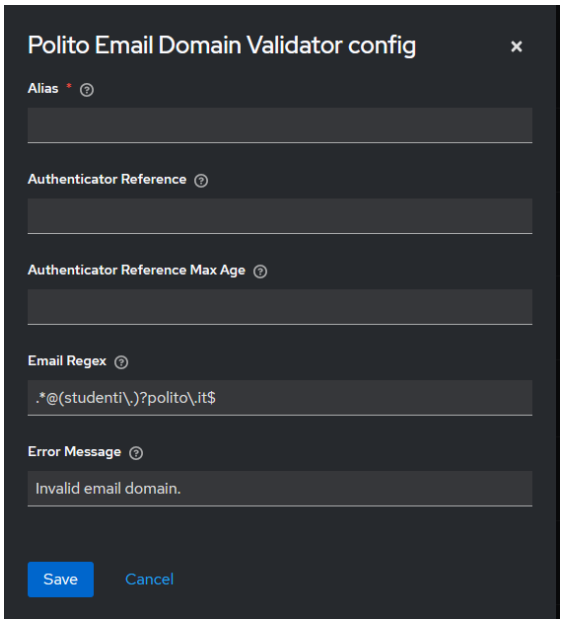

Figure 4.2: Email validator configuration

4.5.3 Dynamic Role Assignment via Event Listener

A core requirement of the platform was the automation of authorization. Specifically, users required immediate assignment of roles (e.g., *Viewer* or *Editor*) based on the pattern of their email address at the moment of registration. While Keycloak supports attribute mapping for external Identity Providers (e.g., SAML/OIDC), it lacks native support for regex-based role mapping during standard user registration.

To bridge this gap, a **EventListenerProvider**, named `PolitoRoleListener`, was implemented. This component adopts an event-driven architecture, subscribing specifically to the `REGISTER` event type.

The implementation utilizes a "configuration-as-data" approach to decouple the code from the business rules:

- **Rule Definition:** Mappings are not hardcoded in Java. Instead, they are stored as **Realm Attributes** within Keycloak's core configuration. The listener scans for attributes carrying the reserved prefix `role-mapping:`.
- **Pattern Matching Mechanism:** The key suffix defines the *Target Role* (e.g., `role-mapping:Editor`), while the attribute's value defines the *Regular Expression* used for matching.
- **Execution Logic:** Upon a user registration event, the listener iterates through the realm attributes. If the user's email matches a configured regex, the corresponding role is retrieved and assigned. The listener includes error handling to log warnings if a configured role does not exist in the realm, preventing silent failures.

This architecture allows the authorization logic to evolve dynamically; an administrator can introduce complex new mapping rules simply by modifying realm attributes, without requiring a server restart.

4.5.4 Automated Configuration Management for role mappings

While the `PolitoRoleListener` enforces the logic, managing raw Realm Attributes via the general Keycloak API is error-prone and cumbersome for operators. To streamline this process, a dedicated Command Line Interface (CLI) tool was developed to act as a control plane for the role mapping rules.

Implemented in Python using the `python-keycloak` library, this tool abstracts the underlying storage mechanism (the `role-mapping:` prefix) behind a user-friendly interface. It operates as an ephemeral Docker container, designed to run within the internal network alongside the Keycloak instance.

The tool provides three primary functions:

- **list:** Queries the Keycloak Admin API for realm attributes, filters strictly for those matching the configuration prefix, and presents the active rules in a readable tabular format.

Listing 4.5.1: Keycloak role configurator list command

```
$ docker compose run --rm keycloak_configurator list

Current Role Mappings for realm 'prometeo':
       ------------------------------------------------
* Role: Editor       Regex: .*@polito.it$
* Role: Viewer       Regex: .*@(studenti.)?polito.it$
       ------------------------------------------------
```

- **add:** Accepts a `–role` and `–regex` argument, automatically formatting the attribute key and updating the realm definition via a PATCH request. This ensures that the syntax expected by the Java listener is always respected.

Listing 4.5.2: Keycloak role configurator add command

```
$ docker compose run --rm keycloak_configurator add --role "Editor" --regex ".*@polito.it$"

Successfully  mapper Role "Editor" to Regex ".*@polito.it$"
```

- **delete:** safely removes a mapping rule by targeting the specific role name.

This tooling completes the loop, transforming the custom Java SPI from a backend logic component into a fully manageable feature of the platform. The source code for this utility is available in Appendix A.1.3.

4.6 Integration with the PoliTo Identity Provider

A requirement for the new architecture was seamless federation with the existing institutional infrastructure. The platform has been designed to be able to use the authentication means provided by the Politecnico di Torino Identity Provider (IdP). The IdP authenticates users against institutional credentials and provides the required identity attributes to the configured Service Provider via SAML assertions, containing user authentication and authorization information.

4.6.1 Infrastructure Overview

The integration connects to a robust infrastructure based on **Shibboleth IdP 5**, an open-source platform standard in academic federations. [14] The architecture is designed for resilience and security, consisting of:

- **High Availability Topology:** The system is composed of a main node, responsible for managing the primary database, the Shibboleth SP, and a Apache HTTPD-based load balancer with AJP (Apache Jserv Protocol) to distribute requests between two IdP nodes. This ensures that the authentications service remains available even during high-traffic periods.

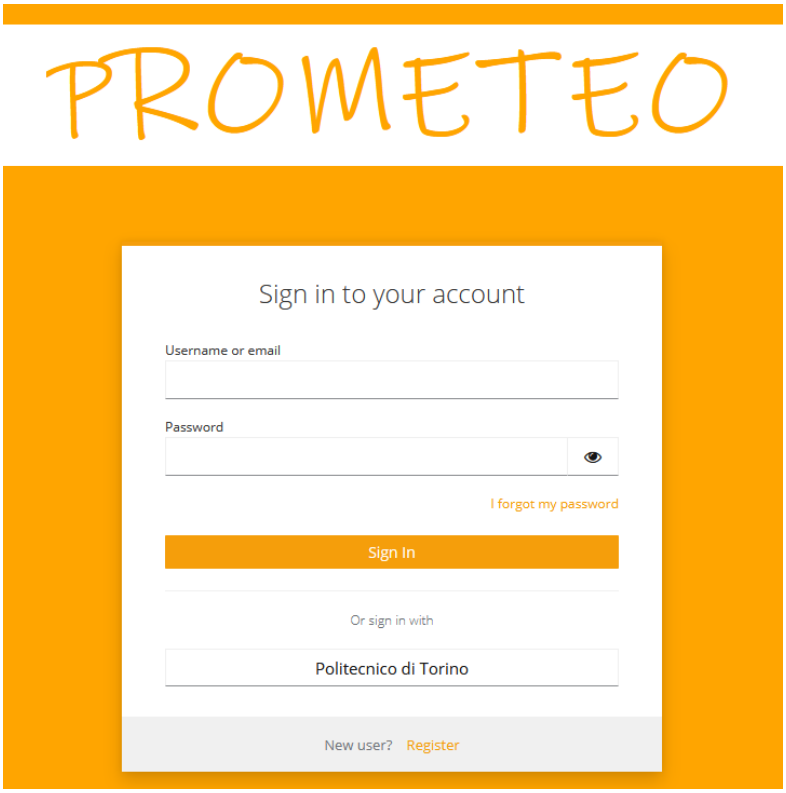

Figure 4.3: Prometeo Polito IdP login

- **Integrated Multi-Factor Authentication (MFA):** The main node also hosts a subsystem, **eduMFA**, that manages MFA (via TOTP, SMS or app-based push notifications). By federating with this system, the PROMET&O platform automatically inherits these security controls without requiring a custom MFA implementation.
- **Multiple Authentication Methods:** The IdP supports various authentication methods:
  - **Username Password Login:** Traditional method based on institutional credentials
  - **Passkey Login:** Passwordless authentication based on trusted devices
  - **Certificate Login:** Legacy method based on client certificates
  - **SPID, CIE, eIDAS Login:** Federated authentication through national and European digital identity systems
  - **IDEM/eduGAIN Login:** Federated authentication via academic and research identity federations

4.6.2 SAML 2.0

The integration utilizes the **SAML 2.0** protocol. In this context, Keycloak functions as the Service Provider, acting as an "OIDC Proxy", to bridge the gap between the XML-based institutional IdP and the JSON-based OIDC services used via APISIX.

The authentication flow follows the standard SP-initiated pattern [15]:

1. **Request:** When a user attempts to login to the platform, Keycloak generates a SAML Authentication Request and redirects the User to the PoliTo IdP.
2. **Challenge:** The IdP authenticates the user using institutional credentials.
3. **Response:** Upon success, the IdP issues a digitally signed SAML Response, containing the user identity information, which is sent back to Keycloak.
4. **Verification:** Keycloak verifies the XML signature, maps the assertion to a local OIDC token and the needed attributes to internal user attributes.

4.6.3 Configuration and Attribute Mapping

By importing the IdP's metadata XML, Keycloak was allowed to dinamically update SSO endpoints and signing certificates. To ensure data consistency between the institutional profile and the local one, specific **Attribute Importers** were configured to map the incoming SAML attributes to the Keycloak user model.

Keycloak Field	**SAML Attribute**	**Description**
Principal ID	`uniqueIdentifier`	Immutable, persistent user ID
Email	`epuid`	Canonical institutional email address
First Name	`givenName`	User's registered first name
Last Name	`sn`	User's registered surname

Table 4.1: SAML to Keyclok Attribute Mapping

Chapter 5

Design and Implementation: Routing and API Gateway

5.1 Introduction and Architectural Evolution

To enhance the scalability and maintainability of the architecture, the local reverse proxy within the namespaces was replaced by an instance of APISIX, to leverage its advanced capabilities both as a high-performance reverse proxy and a feature-rich API gateway.

A critical driver for this architectural migration was the necessity to decouple the authentication logic from the backend services. In the previous iteration, a dedicated Node.js server handled authentication state and redirection. This centralized point of failure introduced latency (an extra network hop) and operational complexity (maintenance of custom authentication code). By adopting APISIX, the system moves towards a decentralized, *serverless* architecture where authentication logic is embedded directly into the network edge via **Lua** plugins, running within the high-performance OpenResty runtime.

With this migration, the system achieves the following improvements:

- **Dynamic Configuration via Control Plane:** Unlike traditional reverse proxies that rely on static configuration files, APISIX exposes a RESTful Admin API. In this implementation, a custom Python automation script utilizes this API to dynamically provision routes and consumers, ensuring the gateway state remains synchronized with the cluster state without service interruption.
- **Modular Plugin Architecture:** The gateway abstracts the complexity of raw Lua scripting into a structured plugin framework. This provided the

necessary hooks to implement custom logic in a reusable manner, decoupling the authentication code from the underlying proxy mechanism.

- **Resource Optimization:** By terminating invalid requests at the proxy level, upstream traffic is significantly reduced, decreasing the computational overhead on backend services.
- **Performance Optimization:** By eliminating the network hop to the dedicated Node.js service, there could be a reduction in the request latency and increased throughput.
- **Unified Observability:** The gateway acts as a centralized telemetry point, providing standardized logging and metrics for all traffic.
- **Easier Maintenance:** Eliminating the dedicated authentication container reduces the maintenance burden, minimizing the work needed for software updates and dependency management.

5.2 Comparison with Traditional Reverse Proxies

While APISIX is built upon the Nginx core (specifically OpenResty), it introduces a dynamic control plane that distinguishes it from a standard Nginx deployment. This distinction is crucial for the automated routing requirements of this project, as summarized in Table 5.1.

Feature	**Standard Nginx / OpenResty**	**Apache APISIX**
Configuration	Static Files (`nginx.conf`)	Dynamic (Admin API + ETCD)
Updates	Process Reload Required	Real-time (Hot-reloading)
Extensibility	Raw Lua Scripts	Structured Plugin Framework

Table 5.1: Comparison between Nginx and APISIX Capabilities

5.3 System Architecture and Configuration

5.3.1 Data Plane vs. Control Plane Topology

APISIX supports multiple deployment modes. For this implementation, the Traditional/Clustered Mode was selected over Standalone mode to support the requirement for dynamic API-driven updates.

In this mode, we can separate two planes:

- **Data Plane (APISIX):** The APISIX instance listens on port `9080` for user traffic and port `9180` for Admin API requests. It handles the actual processing of requests, executing the plugins and forwarding traffic to upstreams.
- **Control Plane (ETCD):** This acts as a single source of truth. When the Admin API is called to generate a new route or service, it is immediately saved in etcd and loaded in APISIX.

While the Standalone mode exists, were APISIX loads configuration from a local yaml file, it was deemed unsuitable because it lacks the ability to interface with the Admin API, needed by the external scripts created in this thesis work to push state changes programmatically.

5.3.2 State Persistence and Disaster Recovery

Since the gateway is stateless, all configuration resides in the etcd container. To ensure system reliability and recoverability, a backup strategy using the etcdctl utility was implemented.

The content of the etcd database can be periodically exported to a snapshot file. This allows the entire routing configuration to be restored in the event of a data loss.

- **Backup Command:** `etcdctl snapshot save /backup/snapshot.db`
- **Restoration Process:** Restoring involves stopping the container, deleting the data volume, and re-initializing the database from the snapshot using `etcdutl snapshot restore`

This separation of state (ETCD) and processing (APISIX) ensures that the gateway nodes can be destroyed and recreated at will without data loss, provided the etcd container or snapshot remains intact.

5.3.3 Dynamic Provisioning

A unique requirement of this thesis is the dynamic configuration of infrastructure. Each survey campaign requires unique routing rules, and manually configuring these is infeasible. To solve this, a specialized container was developed, of which will be talked in a future chapter.

5.4 Authentication Logic Implementation

Normal Login

Routes under `/api/*` are protected by APISIX. When someone contacts them, APISIX checks if an OIDC session is active; this session is saved as an access token within cookies.

If there is no session, a redirect is performed to the Keycloak login page. Keycloak validates the user credentials and returns an access token to APISIX, which inserts it into the session cookies (along with user-related information, the refresh token, etc.).

The Login button points to `/api/login`. Thus, clicking on it initiates the authentication flow described above.

Anonymous Login

Since submitting a questionnaire involves a POST request to `/api/prv/survey`, this route is also protected by APISIX, which must verify the presence of a valid session. Therefore, even for access via an "anonymous user," a valid OIDC session must be established with Keycloak (which, incidentally, must be signed with the same secret, otherwise they appear as different sessions).

The "normal" authentication flow expects the user to enter credentials, so a custom plugin that automatically creates a session without user intervention is used:

- The credentials of an "anonymous" user in Keycloak are used to obtain an access token from Keycloak via a password-grant.
- The token is used to retrieve user information from Keycloak (roles, permissions, info).
- An OIDC session is manually created and saved in the cookies.

The client recognizes whether the user is anonymous or not via an `is_anonymous` attribute from Keycloak. In both cases, the session is valid for navigating the entire site, but for the anonymous user, the Profile button is hidden (as they don't have one). Since they are effectively a full user on Keycloak, you can assign different permissions if desired (e.g., if you want to allow access to specific charts, dashboards, or a "partial" profile).

Furthermore, if logged in as an anonymous user, the client still displays a Login button. Clicking this first performs a logout from the current session and automatically redirects to the Keycloak page where the user can log in with their personal credentials. In this way, the fact that they were logged in anonymously is completely transparent to the user (if that is the desired behavior).

5.5 Security Architecture and API Gateway Configuration

The security architecture of the platform is built on a *Zero Trust* model, where every request is authenticated and authorized at the edge. This is achieved by using the Apache APISIX API Gateway as a Policy Enforcement Point (PEP), integrated with Keycloak as the Identity Provider (IdP).

5.5.1 Global Authentication: OpenID Connect (OIDC)

The `openid-connect` plugin is the primary authentication layer used across all protected routes. It manages the user lifecycle from login to session maintenance.

The plugin has the following main configuration parameters [16]:

- **discovery**: Points to the Keycloak `.well-known/openid-configuration` endpoint. This allows APISIX to automatically retrieve the necessary URLs for login, logout, and token exchange.
- **client_id**: The unique identifier for the application (Relying Party) registered in Keycloak. This ID allows Keycloak to apply the correct client-side configurations, such as allowed redirect URIs and specialized protocol mappers.
- **client_secret**: A confidential string used by the gateway to authenticate itself to the Keycloak token endpoint. By using `$ENV://`, this sensitive credential is kept out of the configuration files, adhering to the principle of least exposure.
- **realm**: Defines the isolation boundary within Keycloak. In the OIDC protocol response, this value is used in the `WWW-Authenticate` header (e.g., `Bearer realm="your-realm"`) to inform the client which security domain is challenging the request.
- **scope**: Specifies the level of access requested from the user.
  - The `openid` scope is mandatory for OIDC to ensure an ID Token is returned.
  - Additional scopes (e.g., `profile`, `email`) allow the gateway to retrieve specific user claims which can then be propagated to upstream services.
- **bearer_only (false)**: By setting this to false, APISIX acts as a *Relying Party* that can initiate the browser-based login flow. If set to true, it would only validate pre-existing tokens.

- **session.secret**: A cryptographic key used to encrypt the session cookie stored in the user's browser, preventing session hijacking or tampering.
- **redirect_uri**: The specific path where Keycloak sends the authorization code after a successful login. This must match the *Valid Redirect URIs* configured in the Keycloak client settings.

Listing 5.5.1: Global OIDC Configuration for Protected Routes

```
"openid-connect": {
  "redirect_uri": "$ENV://KEYCLOAK_LOGIN_REDIRECT_PATH",
  "client_secret": "$ENV://KEYCLOAK_CLIENT_SECRET",
  "set_access_token_header": true,
  "set_id_token_header": true,
  "set_userinfo_header": true,
  "discovery": "$ENV://KEYCLOAK_DISCOVERY_URL",
  "post_logout_redirect_uri": "$ENV://
   KEYCLOAK_LOGOUT_REDIRECT_PATH",
  "scope": "$ENV://KEYCLOAK_OIDC_SCOPE",
  "realm": "$ENV://KEYCLOAK_REALM",
  "client_id": "$ENV://KEYCLOAK_CLIENT_ID",
  "access_token_header": "X-Access-Token",
  "session": {
    "secret": "$ENV://APISIX_OIDC_SESSION_SECRET"
  },
  "access_token_in_authorization_header": true,
  "id_token_header": "X-Id-Token",
  "logout_path": "$ENV://KEYCLOAK_LOGOUT_PATH",
  "bearer_only": false
}
```

Technical Operational Flow

The implementation follows the **OIDC Authorization Code Flow**. [17] When an unauthenticated user attempts to access a protected resource:

1. **Redirection:** APISIX identifies the lack of a session and redirects the user to the Keycloak login page.
2. **Token Exchange:** After successful authentication, Keycloak returns an authorization code. APISIX exchanges this code for an Access Token and ID Token.
3. **Identity Propagation:** The gateway injects the `X-Access-Token` and `X-Id-Token` headers into the upstream request. This allows backend services to identify the user without performing their own OAuth handshake.

4. **Session Persistence:** By setting `bearer_only` to `false`, APISIX maintains a stateful session via an encrypted cookie, providing a seamless User Experience (UX).

5.5.2 Fine-Grained Authorization (RBAC/ABAC)

The `authz-keycloak` plugin [18] enforces fine-grained authorization (Resource-Based Access Control):

- **permissions**: An array defining the required resource and scope. The gateway will deny access if the user's token does not contain a specific "RPT" (Requesting Party Token) issued by Keycloak for that resource.

Grafana Visualization Access (/chart/*)

Access to the Grafana configuration page and web app dashboards is restricted to users with the `View` permission for the Grafana resource.

Listing 5.5.2: RBAC Configuration for Grafana

```
"authz-keycloak": {
  "discovery": "$ENV://KEYCLOAK_DISCOVERY_URL",
  "client_id": "$ENV://KEYCLOAK_CLIENT_ID",
  "permissions": [
    "Grafana_Dashboard#View"
  ],
  "client_secret": "$ENV://KEYCLOAK_CLIENT_SECRET"
}
```

APISIX Dashboard (/ui/*)

Access to the infrastructure management interface, to modify the gateway configuration, is restricted to users with administrative privileges.

Listing 5.5.3: APISIX Configuration for Administrative UI

```
{
  "authz-keycloak": {
      "discovery": "$ENV://KEYCLOAK_DISCOVERY_URL",
      "client_secret": "$ENV://KEYCLOAK_CLIENT_SECRET",
      "permissions": [
        "APISIX_Dashboard#Edit"
      ],
      "client_id": "$ENV://KEYCLOAK_CLIENT_ID"
    }
}
```

To provide a seamless experience for the admin UI, a `serverless-post-function` was implemented. Since the route is already protected by `openid-connect` and `authz-keycloak`, we can safely assume the requester is an authorized administrator.

Listing 5.5.4: Serverless Post-Function for Admin Key Injection

```
"serverless-post-function": {
  "phase": "access",
  "functions": [
    "return function()
       local admin_key = os.getenv('APISIX_ADMIN_KEY')
       if admin_key then
         ngx.req.set_header('X-API-KEY', admin_key)
       end
     end"
  ]
}
```

5.6 Custom plugins

To address specific integration requirements that could not be met by the standard APISIX plugin ecosystem, two custom Lua plugins were developed. These plugins are deployed to the `/opt/apisix/plugins` directory and registered within the APISIX `config.yaml`. They leverage the `apisix.core` library to interact with the request lifecycle and modify internal state.

5.6.1 Automatic Keycloak Authentication

The standard OpenID Connect (OIDC) workflow typically requires user interaction (redirection to a login page). However, for the QR code survey system, the requirements dictated a "headless" authentication flow where an anonymous user is transparently logged in using a pre-configured service account.

The `keycloak-auto-auth` plugin automates this process by implementing the OAuth2 **Resource Owner Password Credentials Grant** flow [19] directly within the gateway. The implementation relies on `resty.http` for upstream communication and `resty.session` for state management.

The plugin's execution logic follows these steps:

1. **Session Check:** Upon receiving a request, the plugin first checks the existing session storage. If a valid, authenticated session exists, the execution is skipped to prevent redundant authentication calls.

2. **Token Negotiation:** If no session exists, the plugin acts as an HTTP client. It constructs a `POST` request to the configured Keycloak token endpoint, transmitting the `client_id`, `client_secret`, and the specific `username/password` credentials defined in the configuration.

Listing 5.6.1: Token negotiation

```
local res, err = httpc:request_uri(conf.token_endpoint, {
    method = "POST",
    body = ngx.encode_args({
        grant_type = "password",
        client_id = conf.client_id,
        client_secret = conf.client_secret,
        username = conf.username,
        password = conf.password,
        scope = conf.scope
    }),
    headers = {
        ["Content-Type"] = "application/x-www-form-urlencoded"
    },
    ssl_verify = conf.ssl_verify
})
```

3. **User Info Retrieval:** Upon successfully receiving an access token, the plugin immediately queries the UserInfo endpoint to fetch the user's profile data (e.g., sub, preferred_username).

4. **Session Hydration:** A critical aspect of this plugin is its compatibility with the standard APISIX OIDC ecosystem. Rather than just storing the token, it manually constructs a session object that mirrors the structure used by the official `openid-connect` plugin. It populates fields such as `id_token`, `access_token`, and `user`, and marks the session as `authenticated = true`.

Listing 5.6.2: Session info creation

```
local current_time = ngx.time()
sess.data = {}
sess.data.authenticated = true
sess.data.id_token = id_token_payload
sess.data.enc_id_token = token_data.id_token
sess.data.access_token = token_data.access_token
sess.data.access_token_expiration = current_time + (
    token_data.expires_in or 3600) - 1
if token_data.refresh_token then
    sess.data.refresh_token = token_data.refresh_token
end
sess.data.user = userinfo
```

```
sess.data.last_authenticated = current_time
```

5. **Cookie Injection:** Finally, the populated session is saved, which triggers the injection of a session cookie into the response headers. This allows the client to maintain the session for subsequent requests without re-authenticating.

The `keycloak-auto-auth` plugin allows the service to act on behalf of a system user, while `jwt-auth` provides a secondary mechanism to validate tokens passed directly via query parameters, facilitating mobile integrations.

The following are the parameters for the keycloak-auto-auth plugin:

- **username / password**: Credentials for a dedicated *Service Account*. This allows the QR service to authenticate itself to Keycloak programmatically without a physical user being present.
- **token_endpoint**: The direct URL used to exchange credentials for an access token via the *Resource Owner Password Credentials* or *Client Credentials* flow.

And for the jwt-auth one:

- **query (token)**: Configures the gateway to look for the JWT in a URL parameter named `token`. This is useful for "one-click" access links generated for mobile devices.
- **key_claim_name (key)**: Specifies which field inside the JWT payload identifies the "Consumer" (user or service) configured in APISIX.

Listing 5.6.3: Automated Authentication for QR Service

```
{
  "keycloak-auto-auth": {
    "username": "$ENV://KEYCLOAK_ANON_USER",
    "password": "$ENV://KEYCLOAK_ANON_PASSWORD",
    "token_endpoint": "$ENV://KEYCLOAK_TOKEN_ENDPOINT"
  },
  "jwt-auth": {
    "query": "token",
    "key_claim_name": "key"
  }
}
```

5.6.2 Automatic User-Info Extraction

Grafana's "Auth Proxy" authentication method relies on specific HTTP headers to identify the incoming user and their permissions. While the upstream OIDC plugin injects a JSON object into the `X-Userinfo` header, Grafana cannot natively parse this JSON to determine roles. The `extract-userinfo-grafana` plugin serves as a translation layer between the OIDC provider and Grafana.

This plugin is configured with a `role_priority` list (e.g., `["Admin", "Editor", "Viewer"]`) and executes the following logic:

1. **Decoding:** It intercepts the request and retrieves the `X-Userinfo` header. Since the header is Base64 encoded by the upstream OIDC plugin, this plugin decodes it and parses the underlying JSON payload using the `cjson` library.
2. **Role Resolution:** A user in Keycloak may possess multiple roles, but Grafana typically expects a single, determinate role for mapping. The plugin iterates through the user's roles and matches them against the configured `role_priority` list. The highest-priority match is selected as the user's effective role.

Listing 5.6.4: Session info creation

```
local username = userinfo.username or ""
local roles = userinfo.roles or {}

-- Determine highest priority role
local rolePriority = plugin_conf.role_priority or { "Admin", "Editor", "Viewer" }
local selectedRole = nil

for _, priorityRole in ipairs(rolePriority) do
    for _, r in ipairs(roles) do
        if r == priorityRole then
            selectedRole = priorityRole
            break
        end
    end
    if selectedRole then break end
end
```

3. **Header Injection:** The plugin injects two new headers required by Grafana:
   - `X-WEBAUTH-USER`: Set to the extracted `username`.
   - `X-WEBAUTH-ROLE`: Set to the resolved, highest-priority role.
4. **Sanitization:** Crucially, the plugin removes the `Authorization` header from the request. If this header remains present, Grafana attempts to validate it as an API Key or Bearer token, causing the Auth Proxy flow to fail. Removing it ensures Grafana relies solely on the `X-WEBAUTH` headers.

5.7 Dashboard and Operational Management

Operational visibility is maintained through the official APISIX Dashboard, which provides a GUI for monitoring traffic, managing routes, and inspecting plugin configurations.

In previous versions, the dashboard was a separate container, but from APISIX 3.13 onward, it is embedded in the main image. To secure the dashboard, a recursive security model was applied: the route to the dashboard (`/ui/`) is itself protected by the APISIX OIDC plugin.

This means that to access the control plane, one must first successfully authenticate against that very same control plane. The dashboard is configured to automatically load the `APISIX_ADMIN_KEY` internally, but only for requests that have passed the OIDC gatekeeper. This ensures that only users with the

specific "Admin" role in Keycloak can access the gateway configuration, providing a robust layer of role-based access control (RBAC) over the infrastructure itself.

To facilitate maintenance without physical server access, the dashboard is exposed via a sub path on the same public domain as the Prometeo instance. This allows administrators to perform routing updates, debug authentication issues, protected by the same OIDC session used for the rest of the application.

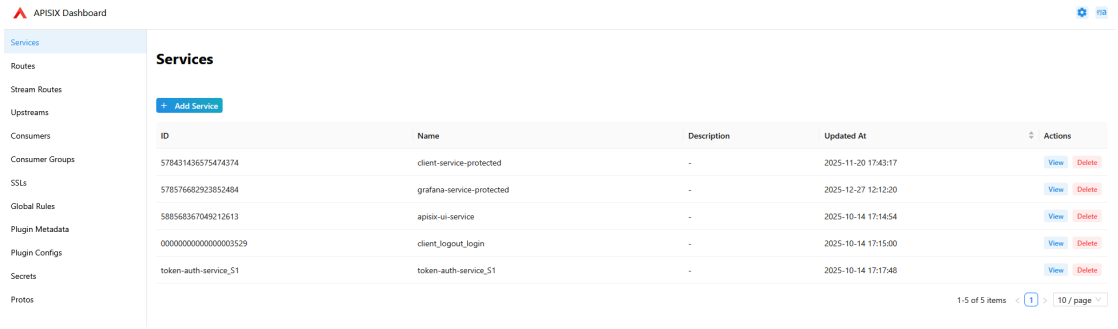

Figure 5.1: APISIX dashboard

Chapter 6

Design and Implementation: Refactoring the Repository

6.1 Legacy vs New Code

This phase of this thesis focused on transitioning the project from a monolithic repository to a modular multi-repo architecture. The previous architecture structure presented several challenges:

- **Access Control:** All contributors had access to the entire codebase, violating the principle of *least privilege*. For instance, a frontend developer does not require access to the Keycloak configuration logic.
- **History Management:** Git logs became cluttered with unrelated changes from different domains.

To resolve these issues, the system was refactored into three distinct repositories, under a centralized GitHub Organization:

- **Ingress:** Manages the NGINX Ingress controller and routing logic.
- **Keycloak:** Dedicated to the global Identity and Access Management instance.
- **NS:** Contains the logic for the various *NS* components.

This separation allows for granular access control, independent versioning, and specialized CI/CD workflows for each service.

6.2 Security Standards and Repository Integrity

A primary objective of the refactor was to eliminate security vulnerabilities inherent in the legacy system.

6.2.1 Secret Management

In the old repository, secrets were stored in `.env` files and committed to version control. This practice posed a critical security risk: if the repository was compromised, secrets and api keys would be compromised as well, giving access to the platform to malicious users. The new architecture implements a dual-layer security approach:

- **Local Development:** `.env` files are strictly included in `.gitignore` to prevent accidental leaks.
- **Production:** GitHub Secrets are utilized to store encrypted variables. During the deployment phase, these secrets are injected into the environment, having an action dynamically generating a `.env` file on the remote server using the secrets stored in the context.

6.2.2 Branching strategy

To prevent code divergence and ensure stability, a strict branching model was implemented. The repository now follows a structured flow:

- **`main` branch:** Represents the production-ready state. It is a **protected branch**, requiring manual approval and successful status checks from the development branch before merging. Direct commits are prohibited.
- **`develop` branch:** Acts a the primary integration branch, where all new features and bug fixes are merged first to undergo automated testing and image rebuilding.
- **feature branches:** Developers branch off `develop` and merge back via Pull Requests (PRs).

To enforce code quality at the developer level, **Husky** was integrated into the frontend workflow. It executes pre-commit hooks (`npm run lint`) and pre-push hooks (`npm run build`) to ensure that only syntactically correct and buildable code reaches the remote repository.

6.3 CI/CD with GitHub Actions

The automation layer was redesigned to support the modular nature of the multi-repo architecture. By utilizing GitHub Actions, the project achieves a clear separation between **Continuous Integration (CI)**, focused on validation and artifact creation, and **Continuous Deployment (CD)**, focused on state synchronization on the production server.

For this work, on the develop branch were created actions for testing and building the images, and on the main branch for deploying on the remote server. An important point was to not rebuild code not touched/modified, so all actions check the folders that contain some changes, and rebuild the container images based on that. Moreover, changes not tied to code or container images, do not trigger a rebuild or redeploy.

6.3.1 Environment-Specific Configuration

To maintain a "Single Source of Truth" while allowing for environment variations, a multi-file Docker Compose strategy was implemented:

- `docker-compose.yml`: Contains the base service definitions, networks, and volumes common to all environments.
- `docker-compose.dev.yml`: Overrides settings for local development, such as mounting local source code for hot-reloading and using local SSL certificates.
- `docker-compose.prod.yml`: Optimizes containers for production, utilizing pre-built images from the **GitHub Container Registry (GHCR)** and defining restart policies (`unless-stopped`).

6.3.2 Continuous Integration (CI) and Validation

The CI pipeline is triggered on every push or Pull Request to the `develop` branch. Its primary goal is to ensure that proposed changes do not break the build or configuration integrity.

- **Nginx Configuration Testing:** In the Ingress repository, the CI workflow performs a "dry run" validation. It spins up a temporary Nginx container and executes `nginx -t`. To simulate the production environment accurately, `–add-host` flags are used to map internal service names (such as `keycloak` or `nso5-apisix`) to the local loopback, preventing configuration errors due to unreachable upstream servers.

Listing 6.3.1: Nginx validation

```
docker run --rm \
    --add-host nso1-app-1:127.0.0.1 \
    --add-host nso2-app-1:127.0.0.1 \
    --add-host nso3-app-1:127.0.0.1 \
    --add-host nso5-apisix:127.0.0.1 \
    --add-host keycloak:127.0.0.1 \
    -v ${{ github.workspace }}/nginx.conf:/etc/nginx/nginx.conf:ro \
    -v ${{ github.workspace }}/conf.d:/etc/nginx/conf.d:ro \
    -v ${{ github.workspace }}/includes:/etc/nginx/includes:ro \
    -v ${{ github.workspace }}/certs/dhparam.pem:/etc/ssl/certs/dhparam.pem:ro \
    -v ${{ github.workspace }}/certs/fullchain.pem:/etc/letsencrypt/live/nso1-prometeo.polito.it/fullchain.pem:ro \
    -v ${{ github.workspace }}/certs/privkey.pem:/etc/letsencrypt/live/nso1-prometeo.polito.it/privkey.pem:ro \
    nginx:alpine nginx -t
```

- **Compose Schema Validation:** For all repositories, the pipeline executes `docker compose config -q`. This ensures that environment variables are correctly mapped and that the YAML syntax is valid before any deployment attempt.
- **Conditional Image Building:** To optimize runner time and storage, the NS repositories utilize the `dorny/paths-filter` action. This allows the pipeline to detect which specific component (Client, Database, or QR-Generator) was modified. Images are rebuilt and pushed to GHCR only for the modified components, significantly reducing the CI overhead.

Listing 6.3.2: Example of conditional image building

```
- name: Build and push Client
    if: steps.filter.outputs.client == 'true'
    uses: docker/build-push-action@v5
    with:
      context: nso5/client
      push: true
      tags: ${{ env.REGISTRY }}/${{ env.IMAGE_OWNER }}/nso5-client:latest
      build-args: VITE_APP_DOMAIN=${{ secrets.DOMAIN }}
      cache-from: type=gha,scope=nso5-client
      cache-to: type=gha,mode=max,scope=nso5-client
```

6.3.3 Continuous Deployment (CD) and Secret Injection

The deployment phase is handled by a dedicated workflow targeting the `main` branch, executed on a **self-hosted runner** located within the production infrastructure. This setup allows the runner to interact directly with the local Docker daemon and internal networks.

Secret Management and .env Reconstitution: Since `.env` files are no longer tracked by Git, the CD pipeline must securely reconstruct the environment. This is achieved by:

1. Storing the entire production environment configuration in **GitHub Secrets** (populated via the `gh secret set -f .env` CLI tool).
2. During the workflow execution, the pipeline dynamically generates a physical `.env` file on the runner using the secrets stored in the repository context.
3. This file is then used by Docker Compose to inject sensitive credentials (e.g., KEYCLOAK_DB_PASSWORD, APISIX_ADMIN_KEY) into the containers at runtime.

Listing 6.3.3: Deployment of the ns05 namespace

```
- name: Log in to the Container registry
    uses: docker/login-action@v4
    with:
      registry: ${{ env.REGISTRY }}
      username: ${{ github.actor }}
      password: ${{ secrets.GITHUB_TOKEN }}

- name: Create .env file
    working-directory: ns05
    run: |
      echo "${{ secrets.NS_05_SECRETS }}" > .env

- name: Pull and Deploy Changes
    working-directory: ns05
    run: |
      docker compose -f docker-compose.yml -f docker-compose.
    prod.yml pull
      docker compose -f docker-compose.yml -f docker-compose.
    prod.yml up -d
      docker image prune -f
```

6.3.4 Repository-Specific Workflow Summary

The following table summarizes the specialized logic applied to each repository within the organization:

Repository	CI Key Feature	CD Key Feature
Ingress	Nginx syntax validation with simulated upstreams.	Zero-downtime reload using `nginx -s reload`.
Keycloak	Automatic building of the custom Java Configurator.	Dynamic `.env` generation for IAM credentials.
NS (Apps)	Path-aware filtering for Client, DB, and QR builds.	Automated pruning of untagged images from GHCR.

Table 6.1: Summary of CI/CD Workflow Responsibilities

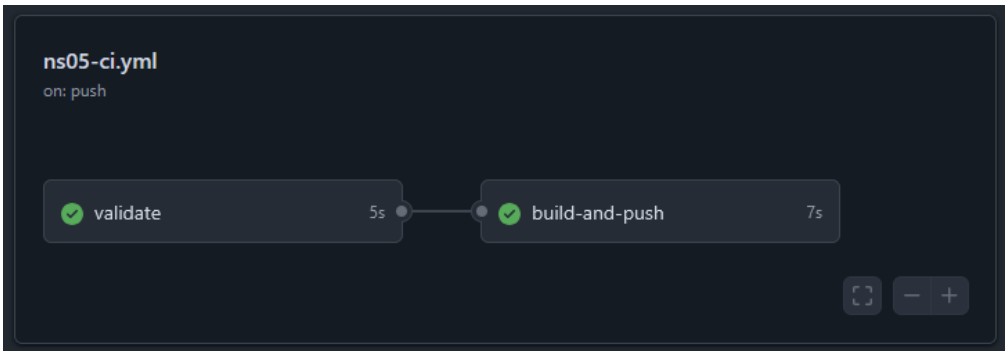

Figure 6.1: CI Workflow page example

Chapter 7

Automated Survey Creation

As distinct survey campaigns often require unique configurations—varying by location, time, or specific access permissions—the authentication infrastructure within APISIX cannot remain static. Instead, it must be dynamically provisioned on a case-by-case basis. To streamline this process, the system employs a containerized Python automation script. This component is responsible for orchestrating the generation of all necessary resources, utilizing a combination of environment variables for static configuration and runtime parameters for campaign-specific data.

The automation tool operates in three primary modes, accessible via the Docker command line interface:

`docker compose run –rm qr-generator list|delete|generate`

The supported operations are defined as follows:

- **list:** Retrieves and displays an enumeration of all currently active `surveyID`s managed by the system.

  Listing 7.0.1: List command for QR code and survey routes

```
$ docker compose run --rm qr-generator list

Surveys found:
- s1
```

- **delete <surveyID>:** Performs a comprehensive cleanup operation. This command removes local artifacts (such as generated QR code images and manifests) and revokes the corresponding configuration resources on the APISIX gateway (Consumers, Services, and Routes).
- **generate:** Initiates the provisioning process. This command accepts several mandatory arguments:

- –buildingID and –roomID: Identifiers for the physical location.
- –seats: The integer count of distinct access tokens to generate.
- –surveyID and –surveyVersion: Metadata linking the access tokens to a specific questionnaire.

Upon execution, this mode generates the requisite QR codes and creates the associated APISIX resources, including protected routes, JWT validation plugins, and upstream authentication configurations.

Listing 7.0.2: Generate command for QR code and survey routes

```
$ docker compose run --rm qr-generator generate --buildingID B01 --roomID R01 --seats 10 --surveyID
    s1 --surveyVersion v1

Created QR for seat1: qr_codes/s1/s1_B01_R01_seat1.png
Created QR for seat2: qr_codes/s1/s1_B01_R01_seat2.png
Created QR for seat3: qr_codes/s1/s1_B01_R01_seat3.png
Created QR for seat4: qr_codes/s1/s1_B01_R01_seat4.png
Created QR for seat5: qr_codes/s1/s1_B01_R01_seat5.png
Created QR for seat6: qr_codes/s1/s1_B01_R01_seat6.png
Created QR for seat7: qr_codes/s1/s1_B01_R01_seat7.png
Created QR for seat8: qr_codes/s1/s1_B01_R01_seat8.png
Created QR for seat9: qr_codes/s1/s1_B01_R01_seat9.png
Created QR for seat10: qr_codes/s1/s1_B01_R01_seat10.png
Manifest written: qr_codes/s1/tokens_s1_B01_R01.json
Saved full manifest for surveyID s1 at qr_codes/manifest_s1.json
```

The script also supports the -h or –help flag to display usage instructions and parameter details.

7.1 Resource State and Manifests

To maintain state without an external database, the implementation utilizes a file-system-based "manifest" approach. When a survey is generated, the script serializes the metadata and resource identifiers into JSON files stored in a persistent volume (qr_codes/).

Below is an example of the generated manifest structure, which serves as the "source of truth" for the lifecycle management of a survey:

Listing 7.1.1: Example of a generated Survey Manifest

```
{
  "surveyID": "s1",
  "created_at": "2026-03-13T22:27:08.920796",
  "expires": "2027-03-13T22:27:08",
  "tokens_manifest": "tokens_s1_B01_R01.json",
  "apisix_consumer_username": "qrcodes_s1",
```

```
  "apisix_service_name": "token-auth-service_s1",
  "apisix_route_name": "token-auth_s1"
}
```

This file allows the `delete` command to identify exactly which APISIX resources need to be purged and which local token files (`tokens_manifest`) are associated with the campaign.

7.2 Practical Implementation

The automation logic is implemented as a Python script designed to run within an ephemeral Docker container. This approach ensures that dependencies—such as cryptographic libraries and image generation tools—are isolated from the host system. The implementation relies on standard libraries for system interaction (`os`, `sys`, `argparse`) and external packages for core functionality: `requests` for HTTP communication with the APISIX Admin API, `PyJWT` for token creation, and `qrcode` for image synthesis.

The script's execution flow is governed by the `argparse` library, which handles sub-command parsing (`generate`, `list`, `delete`) and validation. Configuration is injected into the script through two vectors:

1. **Environment Variables:** Sensitive or infrastructure-level constants (e.g., ADMIN_KEY, CLIENT_ID) are retrieved via `os.getenv`. This separation of concerns prevents hard-coding secrets into the source code.
2. **Runtime Arguments:** Campaign-specific variables (e.g., number of seats, building ID) are passed as CLI arguments, allowing the same container image to serve multiple distinct survey deployments.

7.3 Dynamic Resource Management

The core complexity of the automation lies in the dynamic provisioning of security resources. The script acts as an orchestrator that synchronizes the generation of client-side credentials (QR codes) with server-side access control (APISIX configuration). This process allows each survey campaign to possess its own isolated security context.

7.3.1 Token and QR Code Generation

The generation phase begins by establishing a unique cryptographic context for the survey. The function `generate_jwt_secret` creates a high-entropy, URL-safe string that serves as the signing key for the JWTs.

For every seat specified in the arguments, the script constructs a payload containing specific metadata: `buildingID`, `roomID`, `seatID`, and `surveyID`. This payload is signed using the HMAC-SHA256 algorithm (`HS256`) to produce a JSON Web Token (JWT). Simultaneously, the script generates a corresponding QR code image that encodes a URL in the format:

`https://<host>/<route>?<query_param>=<token>`

This URL structure ensures that when a user scans the code, the token is passed immediately to the API gateway via a query parameter for validation.

7.3.2 APISIX Configuration Hierarchy

Once the tokens are generated, the script contacts the APISIX Admin API to configure the gateway. To ensure modularity and security, the script creates three distinct, linked resources for each survey:

- **Consumer:** A unique consumer entity (e.g., `qrcodes_<surveyID>`) is created to represent the user group for this specific survey. The `jwt-auth` plugin is enabled on this consumer, configured with the specific `jwt_secret` generated earlier. This ensures that only tokens signed with this specific secret are accepted.
- **Service:** An APISIX Service is created to act as an abstraction layer for plugin configuration. This service enables:
  - `jwt-auth`: Configured to extract the token from the URL query string.
  - `keycloak-auto-auth`: A custom plugin configured with environment variables to handle the downstream authentication with Keycloak using a generic anonymous user account.
  - `proxy-rewrite`: Ensures that headers such as `X-Forwarded-For` and `Host` are correctly propagated to the upstream client.
- **Route:** Finally, a specific route (e.g., `token-auth_<surveyID>`) is created. This route maps a specific URI to the Service created above. By decoupling the Route from the Service, the system maintains flexibility in how endpoints are exposed while keeping the authentication logic centralized in the Service object.

7.3.3 Lifecycle and Cleanup

To prevent resource exhaustion, the script includes a robust cleanup mechanism via the `delete` command. By reading the JSON manifest associated with a `surveyID`,

the script identifies the exact names of the APISIX resources (Consumer, Service, Route) created during generation. It then issues HTTP `DELETE` requests to the APISIX Admin API to remove these configurations and deletes the local QR code images and JSON manifests. This ensures that the system returns to a clean state and that old access tokens become immediately invalid once a campaign is concluded.

Chapter 8

Conclusion

The primary objective of this thesis was the architectural modernization and securing of the PROMET&O platform. By transitioning from a fragmented, legacy infrastructure to a centralized, Cloud-Native architecture, this work has established a robust foundation for the platform's future scalability and operational reliability within the Politecnico di Torino ecosystem.

8.1 Summary of Achievements

The project successfully addressed the critical bottlenecks and security vulnerabilities identified during the initial analysis phase. The core achievements can be summarized as follows:

- **Consolidation of Identity Management:** The migration from multiple, resource-heavy Keycloak instances to a single, global realm has significantly reduced the infrastructure's memory footprint and simplified administrative overhead. The implementation of Single Sign-On (SSO) has vastly improved the user experience for students and staff operating across different namespaces.
- **Institutional Integration:** By federating Keycloak with the Politecnico di Torino's Shibboleth Identity Provider via SAML 2.0, the platform now leverages official university credentials. This not only enhances security through institutional Multi-Factor Authentication (MFA) but also ensures that user data is synchronized with the university's "Single Source of Truth."
- **Dynamic Gateway Orchestration:** The adoption of Apache APISIX marked a shift from static, error-prone configurations to a dynamic, API-driven control plane. By offloading authentication logic to custom Lua plugins at the network edge, the backend services were decoupled from security concerns, resulting in a more modular and performant architecture.

- **Security Hardening:** Through a comprehensive audit and subsequent refactoring, critical vulnerabilities—such as hardcoded session secrets and exposed OIDC tokens—were eliminated. The enforcement of "Security by Design" through GitHub Secrets and protected branching strategies has significantly reduced the platform's attack surface.
- **Operational Automation:** The development of the Python-based QR-Generator tool and the implementation of GitHub Actions CI/CD pipelines have transformed the deployment process. What previously required manual configuration of reverse proxies and identity realms is now an automated, repeatable workflow, enabling the rapid provisioning of survey campaigns.

8.2 Final Reflections

This work demonstrates that the modernization of legacy systems is not merely a task of "containerizing" existing code, but a fundamental redesign of how services interact and how trust is managed. The transition to a modular multi-repo structure and the use of Infrastructure as Code (IaC) principles have moved PROMET&O away from a "craftsman" model of deployment toward a professional, industrial-standard DevOps methodology.

The integration of custom Java SPIs for Keycloak and Lua plugins for APISIX showcases the extensibility of open-source tools when faced with niche institutional requirements. It proves that centralized security does not have to come at the cost of flexibility; rather, it provides a stable framework upon which complex business logic (such as regex-based role mapping or anonymous QR-code flows) can be built securely.

8.3 Future Work

While the current architecture represents a significant leap forward, several areas remain for further optimization and expansion:

- **Template-driven Scaffolding:** To improve scalability, future iterations should implement base Docker images and GitHub Template Repositories. This would allow new namespaces to be provisioned instantly with standardized CI/CD and security configurations.
- **Centralized Multi-tenant Observability:** Currently, each namespace runs a dedicated Grafana instance. A more efficient approach would involve a single, multi-tenant Grafana deployment using "Organizations" to silo data, reducing resource consumption while maintaining privacy.

- **Automated Security Auditing (DevSecOps):** To strengthen the "Security by Design" principle, the CI pipelines should be expanded to include Static Application Security Testing (SAST). Tools like Trivy or Snyk could be integrated to automatically scan images for vulnerabilities before deployment.
- **Orchestration with Kubernetes:** As the number of namespaces grows, moving from Docker Compose to Kubernetes would provide better high-availability, automated self-healing, and finer control over resource allocation.

In conclusion, the reworked architecture provides PROMET&O with the resilience and security necessary to serve as a reliable tool for institutional research, ensuring data integrity and user privacy for the university community.

Appendix A

Source Code

A.1 Keycloak

A.1.1 Email whitelisting plugin

Listing A.1.1: Email Whitelisting

```
@Override
public void validate(ValidationContext context) {
    String email = context.getHttpRequest().getDecodedFormParameters().getFirst
    ("email");

    // Get the configuration model
    AuthenticatorConfigModel configModel = context.getAuthenticatorConfig();

    String regexString = ".*@(studenti\\.)?polito\\.it$";
    String errorMsg = "You must use a Politecnico email address.";

    // Overrride with configured values if present
    if (configModel != null && configModel.getConfig() != null) {
        regexString = configModel.getConfig().getOrDefault(CONF_REGEX,
    regexString);
        errorMsg = configModel.getConfig().getOrDefault(CONF_ERROR_MSG,
    errorMsg);
    }

    Pattern pattern = Pattern.compile(regexString);

    // Validate the email with the regex
    if (email == null || !pattern.matcher(email).matches()) {
        context.error(Errors.INVALID_REGISTRATION);
        context.validationError(
            context.getHttpRequest().getDecodedFormParameters(),
            List.of(new org.keycloak.models.utils.FormMessage("email", errorMsg
    ))
        );
    } else {
        context.success();
    }
}
```

```
30
31  @Override
32  public List<ProviderConfigProperty> getConfigProperties() {
33      List<ProviderConfigProperty> config = new ArrayList<>();
34
35      // Regex Field
36      ProviderConfigProperty regex = new ProviderConfigProperty();
37      regex.setName(CONF_REGEX);
38      regex.setLabel("Email Regex");
39      regex.setType(ProviderConfigProperty.STRING_TYPE);
40      regex.setHelpText("The regular expression the email must match.");
41      regex.setDefaultValue(".*@(studenti\\.)?polito\\.it$");
42      config.add(regex);
43
44      // Error Message Field
45      ProviderConfigProperty msg = new ProviderConfigProperty();
46      msg.setName(CONF_ERROR_MSG);
47      msg.setLabel("Error Message");
48      msg.setType(ProviderConfigProperty.STRING_TYPE);
49      msg.setHelpText("Message to show if validation fails.");
50      msg.setDefaultValue("Invalid email domain.");
51      config.add(msg);
52
53      return config;
54  }
```

A.1.2 Role mapping plugin

Listing A.1.2: Role Mapping

```
@Override
public void onEvent(Event event) {
    // Run only on Registration
    if (EventType.REGISTER.equals(event.getType())) {

        RealmModel realm = session.getContext().getRealm();
        UserModel user = session.users().getUserById(realm, event.getUserId());

        if (user != null && user.getEmail() != null) {
            applyDynamicRoles(realm, user);
        }
    }
}

private void applyDynamicRoles(RealmModel realm, UserModel user) {
    String email = user.getEmail();
    Map<String, String> attributes = realm.getAttributes();

    // Iterate over ALL realm attributes
    for (Map.Entry<String, String> entry : attributes.entrySet()) {
        String key = entry.getKey();
        String regexPattern = entry.getValue();

        // Check if this attribute is a rule (starts with the prefix)
        if (key.startsWith(CONFIG_PREFIX)) {

            // Extract Role Name from the key (e.g. "role-mapping:Editor" -> "
    Editor")
            String targetRoleName = key.substring(CONFIG_PREFIX.length());

            try {
                // Check if Email matches the Regex
                if (email.matches(regexPattern)) {

                    // Find and Grant the Role
                    RoleModel role = realm.getRole(targetRoleName);
                    if (role != null) {
                        if (!user.hasRole(role)) {
                            user.grantRole(role);
                            logger.infof("Rule Matched! Granted role '%s' to %s
     (Pattern: %s)", targetRoleName, email, regexPattern);
                        }
                    } else {
                        logger.warnf("Configuration Error: Rule found for role
    '%s', but role does not exist in Realm.", targetRoleName);
                    }
                }
            } catch (Exception e) {
                logger.errorf("Invalid Regex pattern for role '%s': %s",
    targetRoleName, regexPattern);
            }
        }
    }
}
```

A.1.3 Role Mapping Manager

Listing A.1.3: Role Mapping Manager

```
import os
import sys
import argparse
from keycloak import KeycloakAdmin

SERVER_URL = os.getenv("KEYCLOAK_URL", "http://keycloak:8080/")
ADMIN_USER = os.getenv("KEYCLOAK_ADMIN", "admin")
ADMIN_PASS = os.getenv("KEYCLOAK_ADMIN_PASSWORD", "admin")
TARGET_REALM = os.getenv("TARGET_REALM", "prometeo")

# Prefix used by the Java Listener
PREFIX = "role-mapping:"

# Establish Keycloak Admin Client
def get_admin_client():
    try:
        keycloak_admin = KeycloakAdmin(server_url=SERVER_URL,
                                       username=ADMIN_USER,
                                       password=ADMIN_PASS,
                                       realm_name="master",
                                       verify=True)
        return keycloak_admin
    except Exception as e:
        print(f"Error connecting to Keycloak at {SERVER_URL}: {e}")
        sys.exit(1)

# Retrieve realm attributes (ALL)
def get_realm_attributes(admin):
    realm = admin.get_realm(TARGET_REALM)
    if 'attributes' not in realm:
        return {}
    return realm['attributes']

# List current mappings (those with PREFIX)
def list_mappings(admin):
    attrs = get_realm_attributes(admin)
    print(f"\nCurrent Role Mappings for realm '{TARGET_REALM}':")
    print("-" * 60)
    found = False
    for key, val in attrs.items():
        if key.startswith(PREFIX):
            role_name = key.replace(PREFIX, "")
            print(f"  Role: {role_name:<15} Regex: {val}")
            found = True

    if not found:
        print("  (No mappings found)")
    print("-" * 60 + "\n")

# Add or Update a mapping
def add_mapping(admin, role, regex):
    attrs = get_realm_attributes(admin)
    key = f"{PREFIX}{role}"

    # Update local dict
    attrs[key] = regex
```

```
57
58      # Push update
59      admin.update_realm(TARGET_REALM, {"attributes": attrs})
60      print(f"Successfully mapped Role '{role}' to Regex '{regex}'")
61
62  # Delete a mapping
63  def delete_mapping(admin, role):
64      attrs = get_realm_attributes(admin)
65      key = f"{PREFIX}{role}"
66
67      if key in attrs:
68          del attrs[key]
69          admin.update_realm(TARGET_REALM, {"attributes": attrs})
70          print(f" Successfully deleted mapping for Role '{role}'")
71      else:
72          print(f" Mapping for role '{role}' not found.")
73
74  if __name__ == "__main__":
75      parser = argparse.ArgumentParser(description="Manage Keycloak Role Mappings
        ")
76      subparsers = parser.add_subparsers(dest="command", help="Command to run")
77
78      # List Command
79      subparsers.add_parser("list", help="List all current mappings")
80
81      # Add Command
82      add_parser = subparsers.add_parser("add", help="Add or Update a mapping")
83      add_parser.add_argument("--role", required=True, help="The Role Name (e.g.
        Editor)")
84      add_parser.add_argument("--regex", required=True, help="The Regex Pattern")
85
86      # Delete Command
87      del_parser = subparsers.add_parser("delete", help="Delete a mapping")
88      del_parser.add_argument("--role", required=True, help="The Role Name to
        remove")
89
90      args = parser.parse_args()
91
92      # Connect
93      admin = get_admin_client()
94
95      if args.command == "list":
96          list_mappings(admin)
97      elif args.command == "add":
98          add_mapping(admin, args.role, args.regex)
99      elif args.command == "delete":
100         delete_mapping(admin, args.role)
101     else:
102         parser.print_help()
```

A.2 APISIX

A.2.1 Automatic User-Info Extraction

Listing A.2.1: Automatic User-Info Extraction

```
function _M.access(plugin_conf, ctx)
    -- Get the X-Userinfo header
    local userinfo_header = ngx.req.get_headers()["X-Userinfo"]
    if not userinfo_header then
        ngx.log(ngx.ERR, "X-Userinfo header not found")
        return
    end

    -- Base64 decode
    local decoded = ngx.decode_base64(userinfo_header)
    if not decoded then
        ngx.log(ngx.ERR, "Failed to base64 decode X-Userinfo")
        return
    end

    -- Parse JSON
    local userinfo, err = cjson.decode(decoded)
    if not userinfo then
        ngx.log(ngx.ERR, "Failed to decode JSON from X-Userinfo: ", err)
        return
    end

    -- Extract username and roles
    local username = userinfo.username or ""
    local roles = userinfo.roles or {}

    -- Determine highest priority role
    local rolePriority = { "Admin", "Editor", "Viewer" }
    local selectedRole = nil

    for _, priorityRole in ipairs(rolePriority) do
        for _, r in ipairs(roles) do
            if r == priorityRole then
                selectedRole = priorityRole
                break
            end
        end
        if selectedRole then break end
    end

    -- Set headers
    if username ~= "" then
        ngx.req.set_header("X-WEBAUTH-USER", username)
    end

    if selectedRole then
        ngx.req.set_header("X-WEBAUTH-ROLE", selectedRole)
    end

    ngx.req.clear_header('Authorization')
end
```

A.2.2 Automatic Keycloak Authentication

Listing A.2.2: Automatic Keycloak Authentication

```
function _M.access(plugin_conf, ctx)
    -- Create a clone of the config and fetch env variables
    local conf_clone = core.table.clone(plugin_conf)
    local conf = fetch_secrets(conf_clone, true, plugin_conf, "")

    -- Check if a valid session already exists. If so, do nothing.
    -- This is to avoid re-authenticating on every request.
    local session_opts = {
        secret = conf.session_secret,
        cookie = conf.cookie,
    }

    local sess, sess_err = session.start(session_opts)
    if not sess then
        core.log.error("Failed to start session: ", sess_err)
        return 500, core.json.encode({ error = "Failed to initialize session"
    })
    end

    if sess.data and sess.data.authenticated then
        core.log.info("User already has a valid session. Skipping grant flow.")
        return
    end

    core.log.info("No active session found. Starting automatic Keycloak
    password grant.")

    -- Create an HTTP client
    local httpc, err = http.new()
    if not httpc then
        core.log.error("Failed to create http client: ", err)
        return 500, core.json.encode({ error = "Internal server error" })
    end

    httpc:set_timeout(conf.timeout)

    -- Perform the password grant flow with the token endpoint of keycloak
    local res, err = httpc:request_uri(conf.token_endpoint, {
        method = "POST",
        body = ngx.encode_args({
            grant_type = "password",
            client_id = conf.client_id,
            client_secret = conf.client_secret,
            username = conf.username,
            password = conf.password,
            scope = conf.scope
        }),
        headers = {
            ["Content-Type"] = "application/x-www-form-urlencoded"
        },
        ssl_verify = conf.ssl_verify
    })

    if not res then
        core.log.error("Failed to request token from Keycloak: ", err)
```

```
54          return 500, core.json.encode({ error = "Authentication provider request
     failed" })
55      end
56
57      if res.status ~= 200 then
58          core.log.error("Keycloak token endpoint returned status ", res.status,
    ": ", res.body)
59          return res.status, core.json.encode({ error = "Authentication failed",
    details = res.body })
60      end
61
62      -- Parse token response
63      local token_data, decode_err = core.json.decode(res.body)
64      if not token_data or not token_data.access_token or not token_data.id_token
     then
65          core.log.error("Failed to decode token response or token is invalid: ",
     decode_err or "missing tokens")
66          return 500, core.json.encode({ error = "Invalid token response from
    provider" })
67      end
68
69      core.log.info("Successfully obtained tokens from Keycloak.")
70
71      -- Fetch userinfo from the userinfo endpoint
72      local userinfo_res, userinfo_err = httpc:request_uri(conf.userinfo_endpoint
    , {
73          method = "GET",
74          headers = {
75              ["Authorization"] = "Bearer " .. token_data.access_token
76          },
77          ssl_verify = conf.ssl_verify
78      })
79
80      -- If we cannot obtain the userinfo information
81      local userinfo
82      if not userinfo_res or userinfo_res.status ~= 200 then
83          core.log.warn("Failed to fetch userinfo: ", userinfo_err or "status "
    .. (userinfo_res and userinfo_res.status or "unknown"))
84          -- Create a minimal userinfo object as a fallback
85          userinfo = {
86              sub = conf.username,
87              preferred_username = conf.username,
88              name = conf.username
89          }
90      else
91          userinfo = core.json.decode(userinfo_res.body)
92          core.log.info("Successfully fetched userinfo for sub: ", userinfo.sub)
93      end
94
95      -- Decode ID token payload (no verification needed, just extraction)
96      -- The openid-connect plugin expects the decoded payload in the session.
97      local _, payload_b64 = string.match(token_data.id_token, "([^%.]+)%.([^%.
    ]+)%.([^%.]+)")
98      if not payload_b64 then
99          core.log.error("ID token is not a valid JWT format.")
100         return 500, core.json.encode({ error = "Invalid ID token format" })
101     end
102     local payload_json = ngx.decode_base64(payload_b64)
103     local id_token_payload = core.json.decode(payload_json)
104
105     if not id_token_payload then
```

```
106             core.log.error("Failed to decode id_token payload.")
107             return 500, core.json.encode({ error = "Failed to process ID token" })
108         end
109
110         -- Store data in the session in the format openid-connect expects
111         local current_time = ngx.time()
112         sess.data = {}
113         sess.data.authenticated = true
114         sess.data.id_token = id_token_payload
115         sess.data.enc_id_token = token_data.id_token
116         sess.data.access_token = token_data.access_token
117         sess.data.access_token_expiration = current_time + (token_data.expires_in
    or 3600) - 1
118         if token_data.refresh_token then
119             sess.data.refresh_token = token_data.refresh_token
120         end
121         sess.data.user = userinfo
122         sess.data.last_authenticated = current_time
123
124         -- Save the session, which sets the cookie for the client
125         local ok, save_err = sess:save()
126         if not ok then
127             core.log.error("Failed to save session: ", save_err)
128             return 500, core.json.encode({ error = "Failed to create session",
    details = save_err })
129         end
130
131         core.log.info("Session created and cookie set for user: ", userinfo.sub)
132 end
```

A.3 Surveys Generator

Listing A.3.1: Surveys Generator

```
PREFIX = os.getenv("PREFIX") # Prefix of the nsXX
ROUTE = os.getenv("QR_CODE_ROUTE") # Name of the route (eg /tokenAuth)
QUERY_PARAM = os.getenv("QR_CODE_QUERY_PARAM")
APISIX_ADMIN_URL = f"http://{PREFIX}-apisix:9180/apisix/admin"
UPSTREAM_ID = os.getenv("APISIX_CLIENT_ID") # Client upstream ID
HEADERS = {"X-API-KEY": os.getenv("APISIX_ADMIN_KEY")}

QR_CODES_DIR = Path("qr_codes")
QR_CODES_DIR.mkdir(exist_ok=True)

# Generates a URL-safe random string of the specified length
def generate_jwt_secret(length=32):
    return secrets.token_urlsafe(length)

# JWT & QR code generation function
def generate_jwt_and_qr(base_url, building_id, room_id, seats, survey_id,
    survey_version, jwt_key, jwt_secret, expires=None):
    tokens = []

    # Parse the expiry time, if present, or set a 1-year default
    if expires:
        try:
            exp_timestamp = int(datetime.fromisoformat(expires).timestamp())
        except Exception:
            raise ValueError("Invalid expiration date format; use ISO format
    like 2026-12-31T23:59:59")
    else:
        exp_timestamp = int((datetime.now() + timedelta(days=365)).timestamp())

    # Create informations for every seat number
    for seat_num in range(1, seats + 1):
        seat_id = f"seat{seat_num}"
        payload = {
            "key": jwt_key,
            "exp": exp_timestamp,
            "buildingID": building_id,
            "roomID": room_id,
            "seatID": seat_id,
            "surveyID": survey_id,
            "surveyVersion": survey_version,
            "createdAt": int(datetime.now().timestamp()),
        }

        # Create the tokens, add them to the URLs and generate the QRs, saving
    them in the correct folders
        token = jwt.encode(payload, jwt_secret, algorithm="HS256")
        qr_url = f"{base_url}{requests.utils.quote(token)}"

        survey_dir = Path(os.path.join(QR_CODES_DIR, survey_id))
        survey_dir.mkdir(exist_ok=True)

        img_path = os.path.join(survey_dir, f"{survey_id}_{building_id}_{
    room_id}_{seat_id}.png")

        qr_img = qrcode.make(qr_url)
        qr_img.save(img_path)
```

```
53
54          tokens.append({
55              "seatID": seat_id,
56              "token": token,
57              "qrURL": qr_url,
58              "qrImagePath": img_path
59          })
60
61          print(f"Created QR for {seat_id}: {img_path}")
62
63      manifest_file = os.path.join(survey_dir, f"tokens_{survey_id}_{building_id}
        _{room_id}.json")
64      with open(manifest_file, "w") as f:
65          json.dump(tokens, f, indent=2)
66
67      print(f"Manifest written: {manifest_file}")
68      return tokens, manifest_file, exp_timestamp
69
70  # Function to create a consumer in APISIX. The consumer is needed to validate
71  # the JWT extracted from the query param, and contains the secret with which
        the
72  # JWT has been signed
73  def create_consumer(consumer_username, jwt_key, jwt_secret):
74      url = f"{APISIX_ADMIN_URL}/consumers/{consumer_username}"
75      payload = {
76          "username": consumer_username,
77          "plugins": {
78              "jwt-auth": {
79                  "_meta": {"disable": False},
80                  "algorithm": "HS256",
81                  "exp": 86400,
82                  "key": jwt_key,
83                  "secret": jwt_secret
84              }
85          }
86      }
87
88      resp = requests.put(url, json=payload, headers=HEADERS)
89      resp.raise_for_status()
90      return resp.json()
91
92  # Function to create a service in APISIX. The service contains the plugins
93  # configuration, in this case for jwt validation and the "automatic" keycloak
94  # session creation
95  def create_service(service_name, key_claim_name):
96      url = f"{APISIX_ADMIN_URL}/services/{service_name}"
97      payload = {
98          "name": service_name,
99          "upstream_id": UPSTREAM_ID,
100         "plugins": {
101             "jwt-auth": {
102                 "key_claim_name": key_claim_name,
103                 "query": "token"
104             },
105             "keycloak-auto-auth": {
106                 "token_endpoint": "$ENV://KEYCLOAK_TOKEN_ENDPOINT",
107                 "userinfo_endpoint": "$ENV://KEYCLOAK_USERINFO_ENDPOINT",
108                 "client_id":"$ENV://KEYCLOAK_CLIENT_ID",
109                 "client_secret": "$ENV://KEYCLOAK_CLIENT_SECRET",
110                 "username": "$ENV://KEYCLOAK_ANON_USER",
111                 "password": "$ENV://KEYCLOAK_ANON_PASSWORD",
```

```
                "scope": "$ENV://KEYCLOAK_OIDC_SCOPE",
                "session_secret": "$ENV://APISIX_OIDC_SESSION_SECRET"
            },
            "proxy-rewrite": {
                "headers": {
                    "Cookie": "$http_cookie",
                    "Host": "$host",
                    "X-Forwarded-For": "$proxy_add_x_forwarded_for",
                    "X-Forwarded-Host": "$host",
                    "X-Forwarded-Proto": "$scheme",
                    "X-Real-IP": "$remote_addr"
                }
            }
        }
    }

    resp = requests.put(url, json=payload, headers=HEADERS)
    resp.raise_for_status()
    return resp.json()

# Function to create a route in APISIX, the endpoint that is protected by the
# defined service
def create_route(route_name, uri, methods, service_id):
    url = f"{APISIX_ADMIN_URL}/routes/{route_name}"
    payload = {
        "uri": uri,
        "name": route_name,
        "methods": methods,
        "service_id": service_id,
        "status": 1
    }

    resp = requests.put(url, json=payload, headers=HEADERS)
    resp.raise_for_status()
    return resp.json()

# Function to delete a consumer in APISIX
def delete_consumer(consumer_username):
    url = f"{APISIX_ADMIN_URL}/consumers/{consumer_username}"
    resp = requests.delete(url, headers=HEADERS)
    if resp.status_code == 404:
        print(f"Consumer {consumer_username} not found, skipping delete.")
    else:
        resp.raise_for_status()
        print(f"Deleted consumer {consumer_username}")

# Function to delete a service in APISIX
def delete_service(service_name):
    url = f"{APISIX_ADMIN_URL}/services/{service_name}"
    resp = requests.delete(url, headers=HEADERS)
    if resp.status_code == 404:
        print(f"Service {service_name} not found, skipping delete.")
    else:
        resp.raise_for_status()
        print(f"Deleted service {service_name}")

# Function to delete a route in APISIX
def delete_route(route_name):
    url = f"{APISIX_ADMIN_URL}/routes/{route_name}"
    resp = requests.delete(url, headers=HEADERS)
    if resp.status_code == 404:
```

```
173             print(f"Route {route_name} not found, skipping delete.")
174         else:
175             resp.raise_for_status()
176             print(f"Deleted route {route_name}")
177 
178 # Function to list all the surveys created, providing their ID for ease of
        management
179 def list_surveys():
180     manifests = list(QR_CODES_DIR.glob("manifest_*.json"))
181     survey_ids = set()
182     for mf in manifests:
183         try:
184             with open(mf, "r") as f:
185                 j = json.load(f)
186                 if "surveyID" in j:
187                     survey_ids.add(j["surveyID"])
188         except Exception as e:
189             print(f"Skipping malformed manifest {mf}: {e}")
190     return sorted(survey_ids)
191 
192 def get_manifest_path_by_survey(survey_id):
193     for mf in QR_CODES_DIR.glob("manifest_*.json"):
194         try:
195             with open(mf) as f:
196                 j = json.load(f)
197                 if j.get("surveyID") == survey_id:
198                     return mf
199         except Exception:
200             continue
201     return None
202 
203 # Function to delete all the surveys created, provided their ID. It deletes
204 # all the local resources, the QR codes generated and contacts APISIX to delete
205 # its resources too (consumer, service and route)
206 def delete_survey_resources(survey_id):
207     manifest_path = get_manifest_path_by_survey(survey_id)
208     if not manifest_path:
209         print(f"No manifest found for surveyID {survey_id}")
210         return
211 
212     with open(manifest_path) as f:
213         manifest = json.load(f)
214 
215     # Delete QR code files
216     survey_dir = Path(os.path.join(QR_CODES_DIR, survey_id))
217     if survey_dir.exists():
218         tokens_manifest_path = survey_dir / manifest["tokens_manifest"]
219         if tokens_manifest_path.exists():
220             try:
221                 with open(tokens_manifest_path) as tf:
222                     tokens = json.load(tf)
223                 for token_entry in tokens:
224                     qr_path = Path(token_entry["qrImagePath"])
225                     if qr_path.exists():
226                         qr_path.unlink()
227                         print(f"Deleted QR image {qr_path}")
228                 tokens_manifest_path.unlink()
229                 print(f"Deleted tokens manifest {tokens_manifest_path}")
230             except Exception as e:
231                 print(f"Failed deleting QR tokens: {e}")
232 
```

```
        survey_dir.rmdir()

    # Delete APISIX resources
    try:
        delete_route(manifest["apisix_route_name"])
        delete_service(manifest["apisix_service_name"])
        delete_consumer(manifest["apisix_consumer_username"])
    except Exception as e:
        print(f"Error deleting APISIX resources: {e}")

    # Delete full manifest
    manifest_path.unlink()
    print(f"Deleted full manifest {manifest_path}")


def generate_command(args):
    jwt_secret = generate_jwt_secret()
    jwt_key = "anon_user"

    base_url_token_param = f"https://{PREFIX}-prometeo.polito.it/{ROUTE}?{QUERY_PARAM}="
    tokens, tokens_manifest_file, exp_timestamp = generate_jwt_and_qr(
        base_url_token_param,
        args.buildingID,
        args.roomID,
        args.seats,
        args.surveyID,
        args.surveyVersion,
        jwt_key,
        jwt_secret,
        args.expires
    )

    consumer_username = f"qrcodes_{args.surveyID}"
    service_name = f"token-auth-service_{args.surveyID}"
    route_name = f"token-auth_{args.surveyID}"

    consumer_resp = create_consumer(consumer_username, jwt_key, jwt_secret)
    service_resp = create_service(service_name, "key")

    service_id = service_resp.get("value", {}).get("id")
    if not service_id:
        print("Failed to get service ID from service creation response")
        sys.exit(1)

    route_resp = create_route(route_name, f"/{ROUTE}", ["GET", "POST"], service_id)

    # Create full manifest linking all info
    full_manifest = {
        "surveyID": args.surveyID,
        "created_at": datetime.now().isoformat(),
        "expires": datetime.fromtimestamp(exp_timestamp).isoformat(),
        "tokens_manifest": os.path.basename(tokens_manifest_file),
        "apisix_consumer_username": consumer_username,
        "apisix_service_name": service_name,
        "apisix_route_name": route_name
    }
    saved_manifest_path = QR_CODES_DIR / f"manifest_{args.surveyID}.json"
    with open(saved_manifest_path, "w") as f:
        json.dump(full_manifest, f, indent=2)
```

```
292     print(f"Saved full manifest for surveyID {args.surveyID} at {
        saved_manifest_path}")
293
294
295 def list_command(args):
296     surveys = list_surveys()
297     if not surveys:
298         print("No surveys found.")
299     else:
300         print("Surveys found:")
301         for s in surveys:
302             print(f"- {s}")
303
304
305 def delete_command(args):
306     delete_survey_resources(args.surveyID)
307
308 def main():
309     parser = argparse.ArgumentParser(description="JWT QR + APISIX management")
310     subparsers = parser.add_subparsers(dest="command", required=True)
311
312     gen_parser = subparsers.add_parser("generate", help="Generate QR, create
        APISIX resources")
313     gen_parser.add_argument("--buildingID", required=True, help="ID of the
        building")
314     gen_parser.add_argument("--roomID", required=True, help="ID of the room")
315     gen_parser.add_argument("--seats", type=int, required=True, help="Number of
         seats")
316     gen_parser.add_argument("--surveyID", required=True, help="ID of the survey
        ")
317     gen_parser.add_argument("--surveyVersion", required=True, help="Version of
        the survey")
318     gen_parser.add_argument("--expires", required=False, help="Expiration date
        (ISO, default 1 year)")
319     gen_parser.add_argument("--consumerUsername", default=None, help="Username
        of the APISIX consumer to be created")
320
321     list_parser = subparsers.add_parser("list", help="List existing surveyIDs")
322
323     del_parser = subparsers.add_parser("delete", help="Delete resources by
        surveyID")
324     del_parser.add_argument("surveyID", help="Survey ID to delete")
325
326     args = parser.parse_args()
327
328     if args.command == "generate":
329         generate_command(args)
330     elif args.command == "list":
331         list_command(args)
332     elif args.command == "delete":
333         delete_command(args)
334     else:
335         parser.print_help()
336
337
338 if __name__ == "__main__":
339     main()
```

Bibliography

[1] Cloudflare. *Identity and Access Management*. URL: https://www.cloudflare.com/learning/access-management/what-is-identity-and-access-management/ (cit. on p. 4).

[2] Keycloak. *Keycloak Authorization Code Flow*. URL: https://www.keycloak.org/docs/latest/server_admin/index.html#_oidc-auth-flows-authorization (cit. on p. 5).

[3] Keycloak. *Keycloak Resource Owner password credentials grant*. URL: https://www.keycloak.org/docs/latest/server_admin/index.html#_oidc-auth-flows-direct (cit. on p. 5).

[4] Authentik. *Authentik Comparison*. URL: https://goauthentik.io/#comparison (cit. on p. 6).

[5] Authelia. *Authelia Info*. URL: https://www.authelia.com/ (cit. on p. 6).

[6] Red Hat. *What does an API gateway do*. URL: https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do (cit. on p. 7).

[7] Read Hat. *Continuos Integration*. URL: https://www.redhat.com/en/topics/devops/what-is-ci-cd#continuous-integration (cit. on p. 8).

[8] Read Hat. *Continuos Deployment*. URL: https://www.redhat.com/en/topics/devops/what-is-ci-cd#continuous-deployment (cit. on p. 9).

[9] Github. *Undesrtand Github Actions*. URL: https://docs.github.com/en/actions/get-started/understand-github-actions (cit. on p. 9).

[10] Keycloak. *Configuring the hostname*. URL: https://www.keycloak.org/server/hostname#_utilizing_an_internal_url_for_communication_among_clients (cit. on p. 20).

[11] Wikipedia. *TLS termination proxy*. URL: https://en.wikipedia.org/wiki/TLS_termination_proxy (cit. on p. 20).

[12] Keycloak. *Configuring the DB*. URL: https://www.keycloak.org/server/db (cit. on p. 20).

[13] Keycloak. *Authentication Flows*. URL: https://wjw465150.gitbooks.io/keycloak-documentation/content/server_admin/topics/authentication/flows.html (cit. on p. 22).

[14] Politecnico di Torino. *Polito IdP Technical documentation*. URL: https://idp.polito.it/docs/#/info (cit. on p. 25).

[15] Okta Developer. *SAML SP-initiated sign-in flow*. URL: https://developer.okta.com/docs/concepts/saml/#understand-sp-initiated-sign-in-flow (cit. on p. 27).

[16] Apache APISIX. *OIDC-connect plugin description*. URL: https://apisix.apache.org/docs/apisix/plugins/openid-connect/#description (cit. on p. 32).

[17] Autho. *Authorization Code Flow*. URL: https://autho.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow (cit. on p. 33).

[18] Apache APISIX. *Authz-keycloak plugin description*. URL: https://apisix.apache.org/docs/apisix/plugins/authz-keycloak/#description (cit. on p. 34).

[19] Autho. *Resource Owner Password Flow*. URL: https://autho.com/docs/get-started/authentication-and-authorization-flow/resource-owner-password-flow (cit. on p. 35).