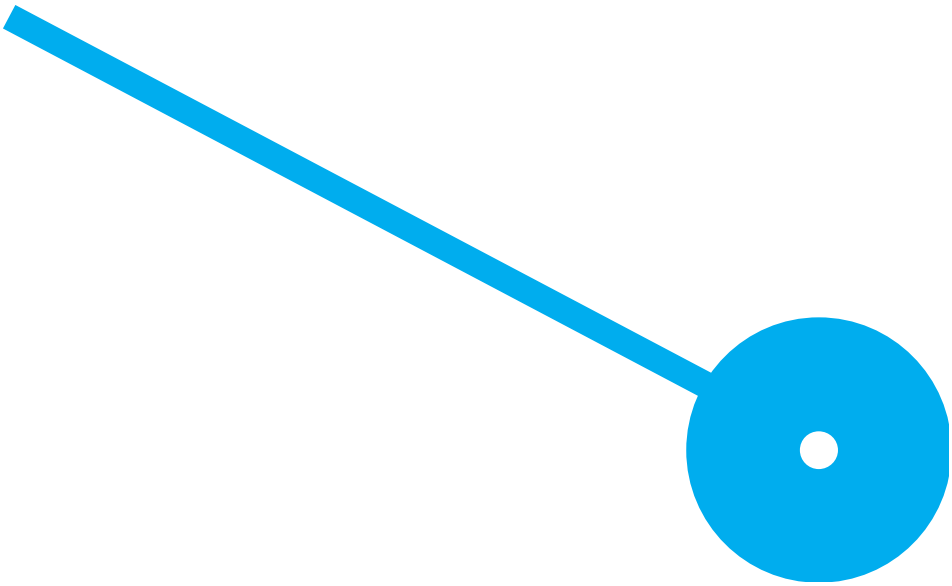
SCHOOL OF
MANAGEMENT AND
TECHNOLOGY
POLYTECHNIC OF
PORTO

MASTER
Engenharia Informática

Authentication API - A SSO Authentication and Authorisation Infrastructure for Web

José Pedro Fernandes

12/2024



SCHOOL OF
MANAGEMENT AND
TECNOLOGY
POLYTECHNIC OF
PORTO

MASTER
Engenharia Informática

Authentication API - A SSO Authentication and Authorisation Infrastructure for Web

José Pedro Fernandes
8190239

Advisor

Professor Doutor Fábio André Souto Da Silva

Project Report submitted in fulfilment of the requirements for the Master's degree in Engenharia Informática in the School of Management and Technology of the Polytechnic of Porto.

12/2024

Integrity Statement

I, **José Pedro Fernandes**, student nº **8190239**, of the Master's Degree in **Engenharia Informática** of the School of Management and Technology of the Polytechnic of Porto, declare that I have not plagiarised or self-plagiarised, therefore the work entitled "**Authentication API - A SSO Authentication and Authorisation Infrastructure for Web**" is original and of my own authorship, not having been used previously for any other purpose. I further declare that all sources used are cited, in the text and in the final bibliography, according to the referencing rules adopted in the institution.

Acknowledgements

I would like to extend my gratitude to Natixis for providing the opportunity to undertake this project and for supporting the creation of this thesis. Without their resources and environment, this work would not have been possible.

I am also grateful to Professor Fábio Silva for accepting the role of my advisor and offering valuable guidance and insights throughout the course of the project. His expertise and support were crucial to this thesis.

A special thanks to Nuno Rocha, whose technical guidance and brainstorming in the early stages of the project was essential in defining the architecture and selecting the most appropriate login strategies.

Abstract

Modern web applications leverage various login techniques, such as Single Sign-On (SSO), passkeys, and password-less authentication, to enhance user experience. Many SSO solutions exist, that enable users to log in once and be authenticated across multiple applications. In this project a custom web authentication system, tailored to the specific needs of a corporate team, was developed. In this team, the lack of web-based authentication infrastructure inhibited the transition from desktop to web applications. The primary objective was to develop a SSO authentication system that not only supports human users but also provides authentication for processes running without a browser, such as automated scripts which will not use SSO but Windows authentication instead. By utilising JSON Web Tokens (JWTs) and refresh tokens, the solution ensures authentication and fast re-authentication, while a distributed cache enables scalability allowing multiple instances to run concurrently. As a result, an Application Programming Interface (API) called AuthenticationApi was developed alongside three internal connection libraries to simplify integration for both web applications and services. A management console was also created to manage the whitelisting of clients, being them web applications or technical processes. The API was rigorously tested, achieving 96.1% code coverage through unit and integration tests, and successfully deployed in two geographical locations, New York and Paris. Structured logs were implemented, offering insights into API performance and usage patterns. Currently, the API is being used in production and serves as a key infrastructure component for the team.

Keywords: Authentication, Web, Single Sign On, JSON Web Tokens, PKCE

Resumo

As aplicações web modernas utilizam várias técnicas de autenticação, como Single Sign-On (SSO), passkeys e autenticação sem palavra-passe, para melhorar a experiência do utilizador. Muitas soluções baseadas em SSO existem, que permitem os utilizadores autenticar-se apenas uma vez e obter acesso a múltiplas aplicações. Neste projeto foi desenvolvido um sistema de autenticação web personalizado, adaptado às necessidades específicas de uma equipa corporativa. Nesta equipa, a falta de uma infraestrutura de autenticação web impedia a transição de aplicações desktop para aplicações web. O principal objetivo foi desenvolver um sistema de autenticação SSO que não só suporte utilizadores humanos, mas também forneça autenticação para processos que corram sem navegador, como scripts calendarizados que não utilizarão SSO mas sim autenticação Windows. Ao utilizar JSON Web Tokens (JWTs) e refresh tokens, a solução garante autenticação e reautenticação rápida, adicionalmente uma cache distribuída permite escalabilidade, possibilitando que várias instâncias funcionem em simultâneo. Como resultado, foi desenvolvida uma *Application Programming Interface* (API) chamada AuthenticationApi, juntamente com três bibliotecas internas para simplificar a integração tanto para aplicações web como para serviços. Foi também criada uma aplicação web para gerir a whitelist de clientes, sejam estes aplicações web ou processos técnicos. A API foi rigorosamente testada, alcançando uma cobertura de código de 96,1% através de testes unitários e de integração, e foi implementada com sucesso em duas localizações geográficas, Nova Iorque e Paris. Foram implementados logs estruturados, oferecendo uma visão detalhada do desempenho e padrões de utilização da API. Atualmente, a API está a ser utilizada em produção e serve como uma peça de infraestrutura fundamental para a equipa.

Palavras-chaves: Autenticação, Web, Single Sign On, JSON Web Tokens, PKCE

Contents

List of Figures

List of Tables

Listings

List of Acronyms

API Application Programming Interface. iii, iv, 1–3, 25, 26, 32, 34–36, 38, 41, 44, 47–59

CKE Content Encryption Key. 13

ECP Enhanced Client or Proxy. 9, 10

GAAP Global Adaptative Access Platform. 25, 26, 28–30, 33, 34, 51, 52

HTTP Hypertext Transfer Protocol. 6–10, 14, 20, 30, 35, 36, 51

HTTPS Hyper Text Transfer Protocol Secure. 20, 21

IdP Identity Provider. 1, 2, 4–6, 9–11, 17, 21, 23, 26

IdPs Identity Providers. 4, 6, 11, 24

IoT Internet of Things. 23

JOSE JSON Object Signing and Encryption. 12

JSON JavaScript Object Notation. 11–14, 36, 39

JWE JSON Web Encryption. 11–14, 21

JWS JSON Web Signature. 11, 12, 14, 21

JWT JSON Web Token. 2, 11, 12, 14–17, 22, 24, 26, 28–30, 34–43, 49, 53, 56

JWTs JSON Web Tokens. iii, iv, 1, 2, 4, 11, 17, 20, 34, 36, 53

MACs Message Authentication Codes. 11, 12

MFA Multi-Factor Authentication. 4, 19, 21, 23, 24

NTLM Network Level Trust Manager. 25, 30, 34, 53

OTP One Time Password. 18, 25

PKCE Proof Key for Code Exchange. 2, 14, 17, 48, 49, 58

PPID Pairwise Pseudonymous Identifiers. 4, 23

RFC Request For Comments. 32, 49, 58

RP Relying Party. 4, 5, 21

RPs Relying Parties. 4, 5, 10, 17, 23

SAML SAML Assertion Markup Language. 1, 2, 5–11, 20, 23, 24, 28, 48, 51

SAML 2.0 SAML Assertion Markup Language 2.0. 1, 4, 5, 17, 19, 20, 24, 28

SGX Software Guard Extensions. 23, 24

SOAP Simple Object Access Protocol. 6, 7, 10

SP Service Provider. 5–7, 9, 10, 26

SPs Service Providers. 6, 10, 11

SSO Single Sign-On. iii, iv, 1, 2, 4–6, 8, 10, 17–26, 29–34, 36, 44, 46, 48, 50, 51, 56, 58

TLS Transport Layer Security. 14, 20

URL Uniform Resource Locator. 14

Chapter 1

Introduction

Web applications have been around for some years, and more are being created every day. As they have become more prevalent, many businesses have adopted them as their preferred type of application. However, not all industry sectors progress at the same pace, some advance more rapidly than others. For example, IT companies typically use the latest and best web tools available, while sectors, such as banking, normally lag behind. In the current web era, offering web applications is more convenient than requiring users to download software. Consequently, there is a growing demand for secure and reliable mechanisms to ensure user web authentication and authorisation. These mechanisms enable the handling of sensitive tasks and prevent unauthorised access. With a diverse user base, various tasks and roles emerge, leading to the implementation of role-based access control, where different users have access to different features within the same application.

Various approaches are utilised to provide authorisation and authentication. The most common one is the use of a username and password [40], which is a form of Single-Factor Authentication. Two factor and Multi factor Authentication require two or more forms of identification to grant user access [40]. SSO authentication allows users to log in once to access multiple applications [40] [46]. SSO is often used with SAML Assertion Markup Language 2.0 (SAML 2.0), also refereed as SAML Assertion Markup Language (SAML) in this document, to exchange identity information, such as user details and roles, across different systems. OAuth 2.0 [17] and OpenID Connect [7] are commonly used to provide SSO functionality across the web [47], [32]. Frequently, token-based authentication is employed, such as using JWTs [24] where a token is needed to access online resources. To retrieve this token, a login is required, and whoever possesses it is granted access [40].

1.1 Context and Motivation

The goal of SSO systems is to provide users with the possibility of logging in once and gain access to multiple applications, which is particularly useful within a corporate environment where users need to use multiple applications to perform their job.

This work proposes the development of a Web API for SSO authentication, leveraging an existing corporate Identity Provider (IdP) to meet the authentication needs of a specific team within the banking organisation. Although there are existing solutions [46], the

sensitive nature of the banking sector mandates the use of in-house applications, where full control of the code is essential, therefore, all code must remain private. SAML will be used to communicate with the IdP, enabling the API to identify users. The Proof Key for Code Exchange (PKCE) protocol will be implemented allowing communication between the client web app and the API. The API will be designed to run multiple instances concurrently behind a load balancer. A distributed cache system will ensure correct functioning of the API, allowing authentication to start on one instance and finish on another. Additionally, a structured logging strategy will be defined, paving the way for future dashboard visualisations and analytics. JWTs [24], secured with asymmetric encryption, will serve as the authentication mechanism for client web apps. This Web API will centralise authentication and authorisation for all web applications used by the team, offloading these responsibilities from individual applications. It is an important step in the web evolution of the team, allowing the creation of new web applications with more complex and sensitive user tasks.

Secure authentication and authorisation are crucial aspects of modern web applications. Incorrect design or implementation can lead to significant security issues, such as unauthorised users performing restricted actions or gaining access to sensitive resources. Ensuring that the right users have access to the right resources is crucial for the security and integrity of a system.

1.2 Objectives

The main objective of this project is to establish a secure and efficient authentication and authorisation environment for all web applications created by the corporation's development team. The aim is to develop an SSO Authentication system capable of efficiently authenticating both human users and technical accounts. The system must be able to perform under various locations and be configurable enough to only allow predefined applications to use it. To accomplish this, several objectives were defined:

- SSO capability: This feature will allow client web applications to provide SSO functionality to users. JWTs, which include user details and roles, will be issued. Additionally, refresh tokens will also be provided, enabling faster retrieval of a new JSON Web Token (JWT) after the previous one has expired.
- Windows login: The system will support Windows login for technical accounts that require authentication to consume services using the JWT provided by AuthenticationApi. This will be useful for processes that run on a schedule such as overnight scripts.
- Multiple locations: The API must be deployable across multiple locations with redundant services. Distributed caching and load balancers will be utilised to maintain consistency and availability across different geographical regions.
- Logging: A structured logging strategy will be implemented. This approach will enable the creation of dashboards, other data analysis activities.
- API access libraries: To simplify the integration process, libraries will be developed to be used by client web applications These libraries will abstract the process of requesting JWTs, requiring minimal configuration for web application developers.

- Management console: A web application will be, allowing administrators to perform CRUD (Create, Read, Update, Delete) operations on the API's whitelist, simplifying management tasks.

1.3 Results

The project resulted in the successful development and deployment of AuthenticationApi, with instances deployed across two regions: Europe and the United States. This multi-region deployment ensures low latency for users in both locations. Over time, the API has handled daily usage without any performance issues or user complaints. The most complex login process, the SSO Login Flow, averages 1.5 seconds, excluding the application load time, while refresh token login flow, which will be mostly used, takes about 0.5 seconds. The API is scalable, supports multiple regions and has been integrated into a production environment, where, every day, plays a crucial role by supporting web applications and other technical processes.

Testing was an essential aspect of the project's success, with a total of 378 tests created to ensure the correct functioning of the API. These tests provided 96.1% coverage of the API's core code, significantly reducing the risk of bugs during future updates or modifications. Additionally, various features were implemented, such as structured logging and refresh tokens, which made possible analysing the current API state and enhanced user experience by reducing login times.

Overall, the project successfully achieved its initial objectives, delivering a secure, scalable, and high-performance SSO solution that has been operating reliably since its deployment. Future enhancements, such as advanced monitoring, caching improvements, and more detailed user log analysis, remain possible, ensuring the system can evolve to meet future requirements.

1.4 Document Structure

This document is composed of eight chapters. The first chapter serves as an introduction to the work presented, where context, motivation and goals are described. The second chapter introduces the theory behind the concepts discussed throughout the document as well as a brief critical analysis. The third chapter focuses on the current state of web and SSO authentication, highlighting current solutions, standards, and the outlook for this field. The fourth chapter reviews similar projects related to SSO web authentication by other authors. The fifth chapter presents the solution design, explaining all the components related to the project, how all the authentication flows operate and how the distributed system works. The sixth chapter goes into the implementation details of the API, libraries, and management console, such as endpoints and the token generation process. The seventh chapter outlines the achievements of the project, the current API deployment and the creation of unit and integration tests. Finally, in the eighth chapter conclusions are taken from the developed project and possible future work is detailed.

Chapter 2

Theoretical Background

This chapter presents the theoretical foundations that underlie the design and implementation of modern authentication systems. It begins with an overview of authentication methods, protocols, and standards, which serve as the backbone to authentication system, such as SSO, Multi-Factor Authentication (MFA), SAML 2.0, JWTs and refresh tokens. Additionally, it discusses distributed systems and security considerations that ensure the resilience, scalability, and efficiency of web-based authentication solutions.

2.1 Single Sign-On Authentication

SSO services are used as an infrastructure for streamlined identity management and authentication. In an SSO system, the IdP is the entity that authenticates the user, it is responsible for containing user details and providing an authentication process that any Relying Party (RP) can use [2], [14]. The RP is an entity, such as a web application, which provides some service to a user. It interacts with the IdP, where a trust relationship exists, to authenticate the users and decide if they are or are not allowed to access a specific resource [46].

The primary advantage of SSO is user convenience, typically requiring a single authentication for accessing multiple Relying Parties (RPs) [46], [2], for example a username and password, but multi-factor authentication can be also used in SSO systems [28]. Additional advantages include the delegation of the authentication responsibility from the RPs to the Identity Providers (IdPs) [14], minimising the security risk concerning the user management [46] and the reduction of the risk of weaker passwords, this happens because users only need to remember one password, instead of multiple ones, increasing the chances of choosing a stronger password [46], [2].

However, SSO comes with its own set of challenges, particularly regarding user privacy. The US National Institute of Standards and Technology suggests the adoption of Pairwise Pseudonymous Identifiers (PPID) or privacy-enhancing cryptographic protocols to prevent tracking activities [13]. This measure is effective at preventing IdPs from tracking user activity, however, it does not prevent collusive RPs from doing so. To solve this problem, solutions were proposed [14], [50]. Some other challenges encountered by SSO are, the necessity of an IdP which, in case of malfunctioning, will impact several RPs, the danger of one attacker gaining access to the SSO account, giving access to all related RPs.

Lastly, the time, cost, and setup procedure that must be undertaken by companies that aim to use it, might be impractical, making it not easy to implement in smaller companies [46].

There are two high level use cases where SSO is used, web Single Sign-On and identity federation SSO [35]. In the Web SSO use case the user is authenticated in one IdP and can access resources in RPs applications. This is made possible as there is a business agreement between the IdP and the RPs, where they agree on a federated identity for the user. In the Identity Federation case the SSO infrastructure is deployed having in mind the user can authenticate across multiple domains with SSO. Most of the RPs contain a local identity for users, only known to them, and each identity is linked to the federated identity that is used to communicate across different entities.

2.2 SAML 2.0

The SAML 2.0 [35] is an XML-based framework designed for exchanging security information between online entities. It is commonly used to provide SSO across multiple services although it can also be used in other contexts [3].

The core of SAML 2.0 revolves around the exchange of SAML assertions, which are XML-based representations of user authentication information. The participants of the SAML protocol are known as SAML asserting party and SAML relying party. The SAML asserting party, also called SAML authority, is responsible for generating the SAML assertions, which are then used by the SAML relying parties. Both entities can make direct requests to other SAML entities, the requesting party is called the SAML requester, and the responding party is the SAML responder. The responder may or may not accept the SAML assertion, depending on whether it has or not a trust relationship with the asserting party, which was the original one generating the assertion.

In the SSO use case, there is also the definition of IdP and Service Provider (SP), above mentioned as RP. The multi-domain SSO is the most important use case where SAML is applied [35] , where a user can authenticate across multiple domains without having to log in separately for each one. For example, if a user has a session established with web site ‘a’ and at some point it is redirected to web site ‘b’, if there is a trust relationship between web sites ‘a’ and ‘b’, where ‘a’ acts as an IdP and ‘b’ acts as a service provider, then the user will not need to recreate a new session with web site ‘b’. Since there is already an established session with the IdP that is trusted by the web site ‘b’. This scenario is described as IdP-initiated web SSO where the session was first established with the IdP and only after the user was redirected to the SP. In contrast, in an SP-initiated web SSO scenario, the user initially attempts to access the SP. If no authentication is present, the SP redirects the user to the IdP for authentication. An assertion is generated by the IdP and validated by the SP, allowing the user to be authenticated and access the requested resources.

SAML assertions [4] contain statements about an entity which contain information such as email address, authorisations, name, groups that the user belongs to and so on. The assertions are usually created when requested to an asserting party but can also be delivered to a relying party in an unsolicited manner. There are three different statements that can be used in an assertion:

- Authentication statements: Generated by the entity that authenticates the user, these statements specify how the user was authenticated and the authentication time.
- Attribute statements: Contain attributes about the authenticated user such as the user's job title.
- Authorisation decision statements: Define what the subject is allowed to do, such as accessing a specific resource.

The SAML protocols [4] define how requests and responses should be structured to exchange assertions between parties. These protocols ensure that communication between IdPs and Service Providers (SPs) is standardised and secure. There are some request/response protocols:

- Assertion Query and Request Protocol: Defines the queries that can be used to retrieve SAML assertions previously emitted by the IdP. The request part of the protocol specifies that assertions can be asked to an asserting party using an assertion ID, given that the requester knows this ID. The query part of the protocol defines how to request new or existing assertions based on specific statement types.
- Authentication Request Protocol: Defines how an agent (for example a browser) or a subject can request assertions constituted by authentication statements that will allow an establishment of a security context with one or more replying parties. The response contains at least one authentication statement and may include additional attribute statements. It is used by the Web Browser SSO Profiles.
- Artifact Resolution Protocol: Provides a mechanism that allows the transport of SAML protocol messages in a SAML binding using an artifact (a small, fixed-length value) instead of a value. A sender will send the artifact instead of a binding message. The receiver in conjunction with another SAML binding will treat the artifact as the original protocol message. This is useful when certain bindings do not support the needed size for the request message, and that request is sent via another channel while the response is received from the same channel as the artifact was sent.
- Name Identifier Management Protocol: Provides a protocol for establishing, modifying, or ceasing the name used to identify a subject. If an identity provider wants to change the name of a certain subject it will send a ¡ManageNameIDRequest¿ message to the service providers.
- Single Logout Protocol: It allows to terminate active sessions associated to a subject across multiple service providers provided by a specific IdP.
- Name Identifier Mapping Protocol: This protocol allows an entity, for example a SP, to request, to an IdP, a name identifier for a subject in a specific format if it already shares an identifier with that same IdP. One common use case is when a service provider wants to communicate with another service provider that does not share an identifier for the subject. The service provider can use an identity provider as an intermediary to map its own identifier to a new identifier that is compatible with the second service provider.

SAML bindings [31] define the messaging protocols (Hypertext Transfer Protocol (HTTP), Simple Object Access Protocol (SOAP)) used to transmit the SAML messages between

the various intervenients. The following are the existing SAML bindings:

- SAML SOAP Binding: SOAP is an xml-based protocol for exchanging information. This binding is used when transmitting SAML messages via SOAP. The SAML requester may add arbitrary headers to SOAP message, but none is required. The SAML responder must process the SAML request without requiring any headers. The SAML protocol elements must be encapsulated within the SOAP body.
- Reverse SOAP (PAOS) Binding: This binding supports SAML authentication where the initiator is not the SP but rather another HTTP requester acting as a SOAP responder or intermediary. The flow begins with the HTTP requester making a call to the SP. The SP will reply with the SAML request. Right after, the HTTP requester will call the SP sending in the SOAP body the SAML response and the SP will reply specifying if the authentication was successful or not. This binding supports the Enhanced Client or Proxy profile, described below, where the HTTP requester acts as a middleman in the authentication process.
- HTTP Redirect Binding: This binding allows the exchange of SAML messages as URL parameters and their transmission through HTTP redirects. This binding can be used in conjunction with HTTP POST binding or HTTP Artifact binding, particularly in cases where there are URL length limitations. This binding allows for the usage of RelayState which is a small piece of data added by the SAML requester that must be returned by the SAML responder. The sequence flow of this binding is illustrated in Figure 2.1.

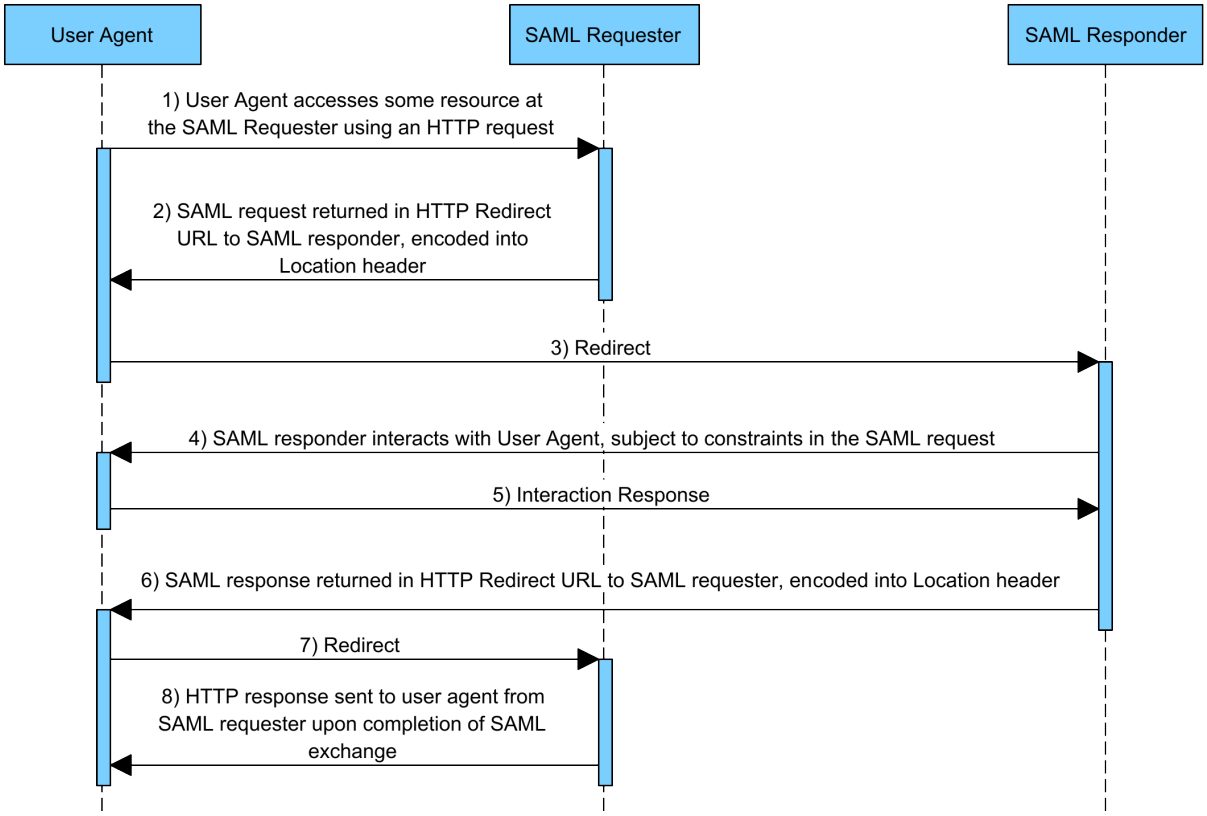


Figure 2.1: HTTP Redirect Binding flow, taken from [31]

- HTTP POST Binding: This binding allows for the exchange of SAML protocol

messages via an HTML form, encoded in base64, and transmitted using the HTTP POST method. Similarly to the HTTP Redirect Binding, it also supports the usage of RelayState and for the combination with HTTP Redirect Binding or HTTP Artifact Binding. The sequence flow is described in Figure 2.2.

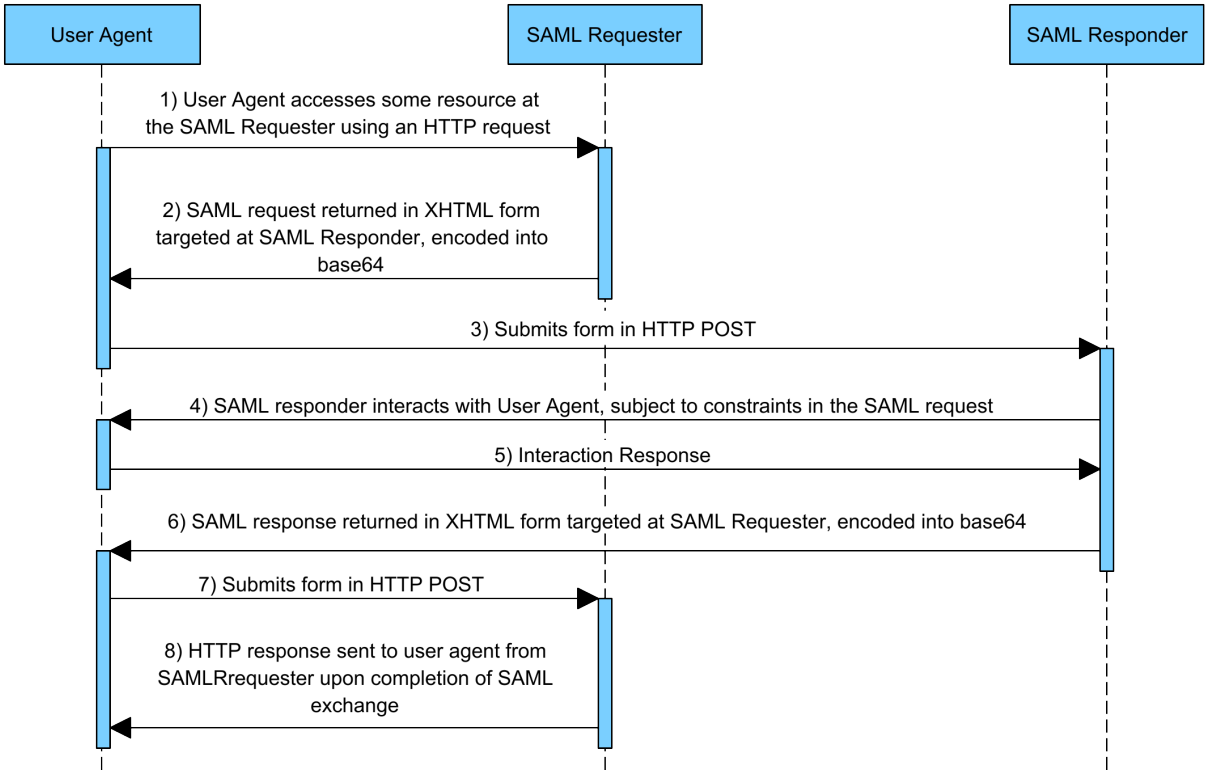


Figure 2.2: HTTP Post Binding flow, taken from [31]

- HTTP Artifact Binding: In this binding, both the SAML request and response can be transmitted using a reference called an artifact instead of the actual message. A separate binding is utilised to exchange the artifact for the actual message. This binding can be combined with HTTP Redirect Binding or HTTP Post Binding. This protocol is used when the SAML request and responder communicate using an HTTP agent, such as a browser, as an intermediary and the transmission of messages with that agent is limited. The flow is similar the two HTTP bindings specific above but there is an extra step centred in artifact resolution. The flow illustrated in Figure 2.3 represents a flow where artifact resolution is utilised in both the response and reply exchange.

- SAML URI Binding: With this binding it is possible to, instead of sending an assertion inside a SAML message, an URI is sent that will always be resolved into the same assertion or a transport-specific error. The URI will not resolve the complete SAML response, only one assertion.

The SAML profiles [18] define the usage of SAML to solve a specific use case. The profile can define what bindings, protocols and assertions are used and allow for interoperability in a specific environment. The existing SAML profile are the following:

- Web Browser SSO Profile: This profile defines how to implement Single Sign-On functionality within web browser-based applications. It enables users to authen-

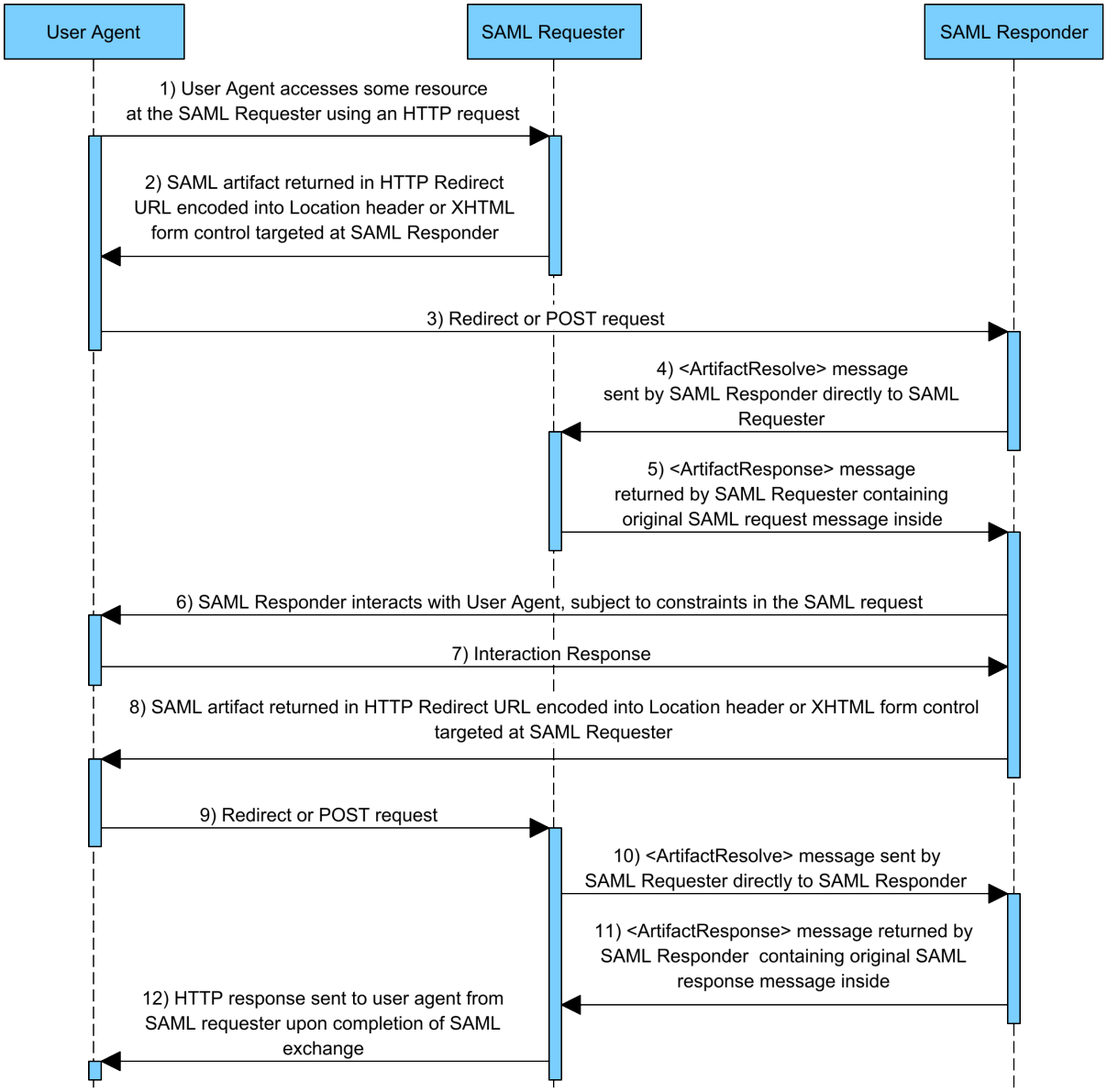


Figure 2.3: HTTP Artifact Binding flow, taken from [31]

ticate once with the IdP and have access to multiple SP without needing to re-authenticate. The flow starts with the user accessing a SP, then, if not authenticated, then the SP will determine which IdP to use, it will construct an AuthnRequest and redirect the user agent to the IdP. The IdP will identify the subject and then reply with a response message to the SP, after this, based on the response, the SP will or not allow the subject to access it's resources. This profile will use the SAML Authentication Request protocol and the bindings HTTP Redirect, HTTP Post and HTTP Artifact.

- Enhanced Client or Proxy (ECP) Profile: In this profile a client (such as a web browser) or a proxy are involved in the authentication process. It is used in situations where direct communication between an SP and an IdP is not possible or desired. The flow begins with the ECP requesting access to some resource at the SP, then, if it does not have a security context the SP will reply with a AuthnResquest message using the PAOS binding, the, ECP determines what IdP it will use and sends an

AuthnRequest to the IdP using SAML SOAP binding. The IdP identifies the subject and provides a response message to the ECP using the SAML SOAP binding. The ECP sends the Response to the SP using the PAOS binding and the SP will or will not allow the subject to access the initially requested resource.

- Identity Provider Discovery Profile: This profile allows for an SP to discover what IdP was used by the subject to perform authentication within the Web Broser SSO profile. This is made possible by a cookie that has the name "_saml_idp". The cookie value must be a set of one or more URI encoded using base64 and separated by a single space character. This cookie is set after an IdP authenticates a user. If the cookie already has the URI of the IdP, it must be removed and added again, this is important so that the most recent used IdP will be the last element of the set.

- Single Logout Profile: This profile allows for a subject or an IdP to terminate the subject's active sessions with all relying parties. When a session participant wants to terminate all existing sessions, it sends a LogoutRequest to the IdP. This step may not be part of the flow as the IdP can terminate sessions without receiving any LogoutRequest. Then, the IdP will send a LogoutRequest message to all session participants (relying parties). After the RPs receive the request, they will respond with a LogoutResponse message. The IdP will send a final LogoutResponse to the session participant. The session participant should use an asynchronous binding (HTTP Redirect, POST or Artifact). The IdP can use any binding to send the LogoutRequest to the relying parties.

- Name Identifier Management Profile: This profile is used when an IdP wants to change the identifier of a subject that was first shared with other SPs or when a SP wants to change its own alias of a subject and make sure that the IdP will recognize that alias or when the IdP or SP wants to inform that will stop issuing or recognizing a specific identifier for the subject. The flow consists in one of the providers sending a ManageNameIDRequest message to the other provider and the last one will reply with a ManageNameIDResponse message. Synchronous and Asynchronous bindings can be used.

- Artifact Resolution Profile: The profile allows for the resolving of an artifact (small piece of data) to a protocol message. This profile is leveraged by the HTTP Artifact binding in conjunction with another synchronous binding. The flow is started by a requesting entity that sends an ArtifactResolve message to the responding entity (containing the artifact), then, the last one will reply with an ArtifactResponse (containing the corresponding protocol message).

- Assertion Query/Request Profile: This Profile is utilised to retrieve pre-existing assertions regarding a subject, and this request can be made either by reference or through direct querying. The flow consists of a SAML requester contacting a SAML authority by sending a query or request message and the authority replying with a status code and it will include the matched assertion if the request is successful. This profile must use synchronous binding.

- Name Identifier Mapping Profile: This profile allows for a user to have different name identifiers. It describes how to map the subject's name into another name that also references the same subject. The flow consists in one system entity sending

a NameIDMappingRequest to an IdP and the IdP replying with a NameIDMappingResponse. This profile must be used with a synchronous binding.

The SAML metadata [5], contained in an XML file, defines the configuration of the SAML environment, such as supported bindings, roles, attributes, usage of encryption and so on. The metadata can be generated by IdPs, describing their capabilities, public keys, and endpoints, and SPs, detailing their endpoints, supported bindings, and other necessary configuration information. The file is consumed by both entities to understand how they can communicate between them.

The SAML authentication context [30] is an XML document that provides information regarding the method and strength of the user authentication that was used during the login process. This can include mechanisms for identification, for minimising compromise of credentials, for storing and protecting credentials and authentication methods. The service providers can, if wished, request to the IdP a different type of authentication.

Figure 2.4, demonstrates how all the above-mentioned concepts are interconnected.

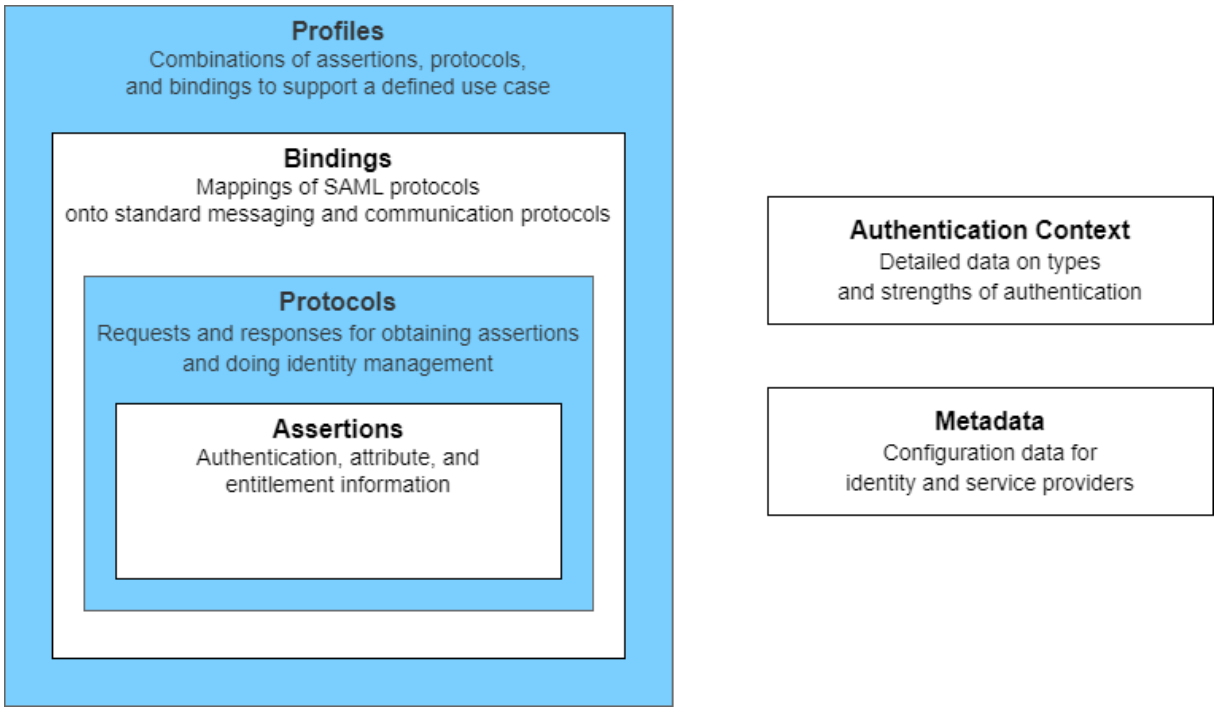


Figure 2.4: SAML Technical Overview adapted from [35]

2.3 JSON Web Tokens

JWT [24] is a compact mechanism for securely transmitting claims between two parties in the form of a JavaScript Object Notation (JSON) object. Claims are key/value pairs, where the key is a string, and the value is an arbitrary JSON value. JWTs have two different implementations, JSON Web Signature (JWS) [23] and JSON Web Encryption (JWE) [26]. JWS will protect the integrity of the token using digital signatures or Message Authentication Codes (MACs). JWE allows for confidentiality of the JWT payload by encrypting it.

The JWS is a representation of JSON data digitally signed or MACed. An example is illustrated in Figure 2.5. There are two serializations formats for JWS tokens, JWS Compact Serialization and JWS JSON Serialization. Both serializations use base64 [27] to encode the protected header, the payload, and the signature. JWS implementation [20] consists of three segments of text separated by periods, the sections are:

- JSON Object Signing and Encryption (JOSE) Header: The JOSE header contains the cryptographic methods and parameters utilised to generate the token. It is divided into JWS protected header and JWS unprotected header. The protected header information is part of the data considered when signing the token, whereas the unprotected header is not. The unprotected header can only be used with JWT JSON Serialization, while JWT Compact Serialization only allows the usage of a protected header. This section is represented in red in Figure 2.5
- JWS Payload: Contains an arbitrary value, usually a JSON object, represented in octets (array of 8-bit elements) which are later encoded to base64. This object usually contains claims relating to the user, such as roles and name. This section is represented in purple in Figure 2.5
- JWS Signature: A digital signature or MAC of the JWS protected header and payload. This section is represented in blue in Figure 2.5

In JWS implementation, there are two ways of verifying the integrity of the token. Digital signatures uniquely belong to a specific entity, allowing verification of the token's source. However, in MACs, the key is shared between multiple parties. With the MACs approach it is not possible to prove the token's origin, so it involves a decision considered when designing the system.

Figure 2.5: JWS example

The JWE [26] represents encrypted JSON content, providing integrity protection. An example is illustrated in Figure 2.6. Its implementation, using the JWE Compact Serialisation, has five sections, separated by periods, which are:

- JOSE Header: The JWE header consists of three potential parts:

  - JWE Protected header: Contains parameters that are integrity protected by the authenticated encryption operation.
  - JWE Shared Unprotected Header: Destined to multiple recipients and is not integrity protected. Can only be used with JWE JSON Serialisation.
  - JWE Per-Recipient Unprotected Header: Destined to a single recipient of the token, these parameters are also not integrity protected. Can only be used with JWE JSON Serialisation.

  This section is represented in red in Figure 2.6.

- JWE Encrypted Key: The result of encrypting the Content Encryption Key (CKE) with the recipient's public key. The CKE is the random key generated to encrypt the plaintext, which produces the JWE Ciphertext. This section is represented in green in Figure 2.6
- JWE Initialisation Vector: A random vector used in the encryption process to add an extra layer of security to the ciphertext. This section is represented in yellow in Figure 2.6.
- JWE Ciphertext: The result of encrypting the plaintext using the CEK as the encryption key, with the Initialisation Vector and the Additional Authenticated Data. Only those possessing the recipient's private key can decrypt this content. This section is represented in purple in Figure 2.6.
- JWE Authentication Tag: The 128-bit output of the encryption operation undertaken when generating the JWE Ciphertext. This section is represented in blue in Figure 2.6.

Figure 2.6: JWE example

The final structure of the token, using the JWE Compact Serialisation, is the composition of the following values separated by periods:

- BASE64URL(UTF8(JWE Protected Header))

- BASE64URL(JWE Encrypted Key)
- BASE64URL(JWE Initialisation Vector)
- BASE64URL(JWE Ciphertext)
- BASE64URL(JWE Authentication Tag)

JWE tokens also have an alternative serialisation method called JSON Serialisation, which is neither Uniform Resource Locator (URL) safe nor optimised for compactness. Instead, it is represented as a JSON object.

There are differences between JWS and JWE. JWS is composed of 3 segments separated by two periods and JWE is composed of 5 segments separated by four periods. JWS contains a payload member while JWE contains a ciphertext member. The algorithms used and present in the header for both implementations are different. Finally JWE contain an "enc" header while JWS do not.

JWS are useful when no sensitive information is included in the token's payload. However, if the payload contains sensitive data that requires protection, then JWE is more appropriate choice as it contains a ciphertext section (encrypted) instead of a payload one (not encrypted).

The JWT can be used with Bearer authorisation [25]. In this case, whoever possesses the token (a "bearer") has access to the resources that are associated with it, being able to perform actions as an authenticated user. Therefore, tokens must be protected from attackers during storage and transportation. The bearer tokens are commonly used with HTTP requests, being included in the authorisation header in the form "Bearer <token>" where "<token>" is the JWT itself.

2.4 Proof Key for Code Exchange

PKCE [1], is a security extension to the OAuth 2.0 authorisation framework. It was created to mitigate the authorisation code interception attack, a vulnerability that arises when OAuth 2.0 [15] clients are attacked with code interception. In this type of attack, the attacker intercepts the authorisation code provided by an authorisation endpoint where communication is not protected by Transport Layer Security (TLS). Once the attacker gains access to this code, they can retrieve the access token, thereby gaining unauthorised access to the user's account.

The normal flow, for this scenario, starts with the application requesting an OAuth 2.0 authorisation via the browser/operating system (step 1). The request is then forwarded to the authorisation server using TLS communication (step 2). The server returns the authorisation code to the browser (step 3) which redirects it to the original application (step 4.1). Following this the application uses this code to request an access token (similar to step 5), and it will be returned by the authorisation server (similar to step 6).

The attack, illustrated in Figure 2.7, happens on step 4.2, where a malicious agent manages to register an application on the client device using the same a custom URI scheme that is used by the original application (for this, the operating system must allow the registration of a custom URI scheme by more than one application). Other conditions for

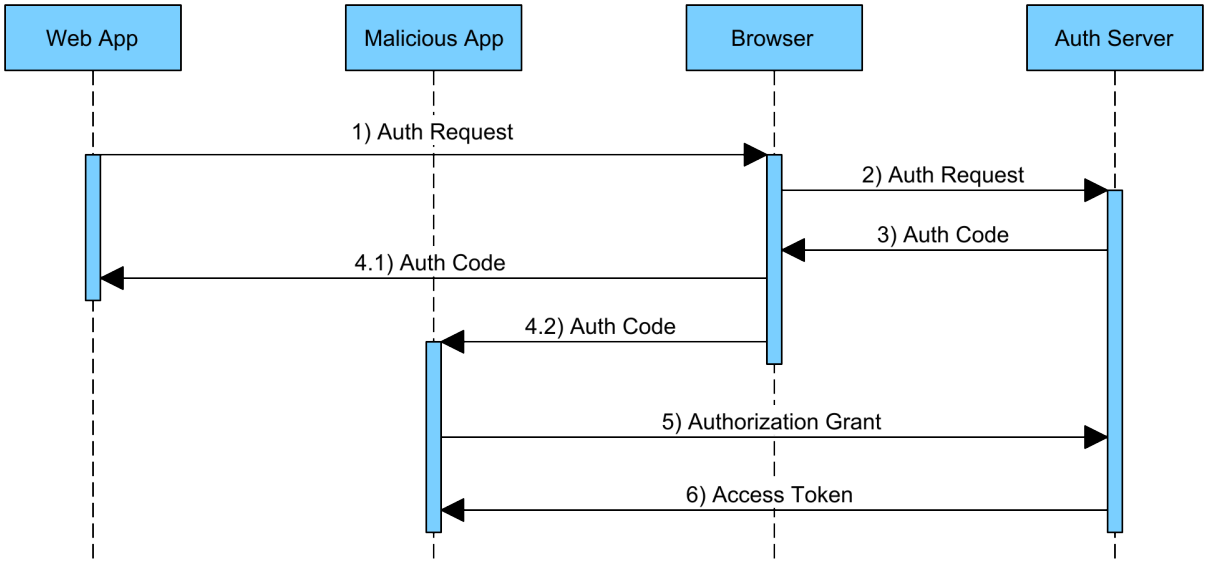


Figure 2.7: Authorisation Code Interception, adapted from [1]

the attack to happen are if OAuth 2.0 authorisation code grant is used and if the attacker has access to the client id and client secret.

To mitigate this attack, PKCE introduces a dynamically generated code_verifier, which is a random string known only to the original app. This is a value only possessed by the original app and it is transformed to a code_challenge via a cryptographic function (SHA256) and sent in the step 1. Later, the authorisation server will only issue an access token if the token request includes the original code_verifier. The server verifies it by applying the same hash function (SHA256) and checking whether the result matches the original code_challenge. The client also, must send a value for the code_challenge_method, in the initial request. This can either be S256, if SHA 256 was used to generate the code_challenge, or plain, if the code_challenge is the same as the code_verifier.

2.5 Refresh Tokens

Refresh tokens [15], enable the retrieval of an access token, such as JWT [24], without requiring the user to re-authenticate. Unlike an access token, a refresh token is a string that cannot be used to gain access to resources directly, they are only intended to request a new access token when the current one is expired or no longer valid. They are optionally issued at the same time as the access token and can have multiple usages, or a single time usage. When the token has a single time usage, after one usage it becomes invalid, and a new one will be generated to replace it.

The main advantage it provides is the possibility of requesting a new access token without the need for re-authentication, which enhances the user experience and saves time. Consequently, another advantage is that it enables authorisation servers to issue short lived access tokens as they are now easily retrieved without user intervention. Refresh tokens, are, in opposite, long lived as they do not provide access to any resource and serve only to get a new access token.

1. Initially, the client makes a request to the authorisation server.

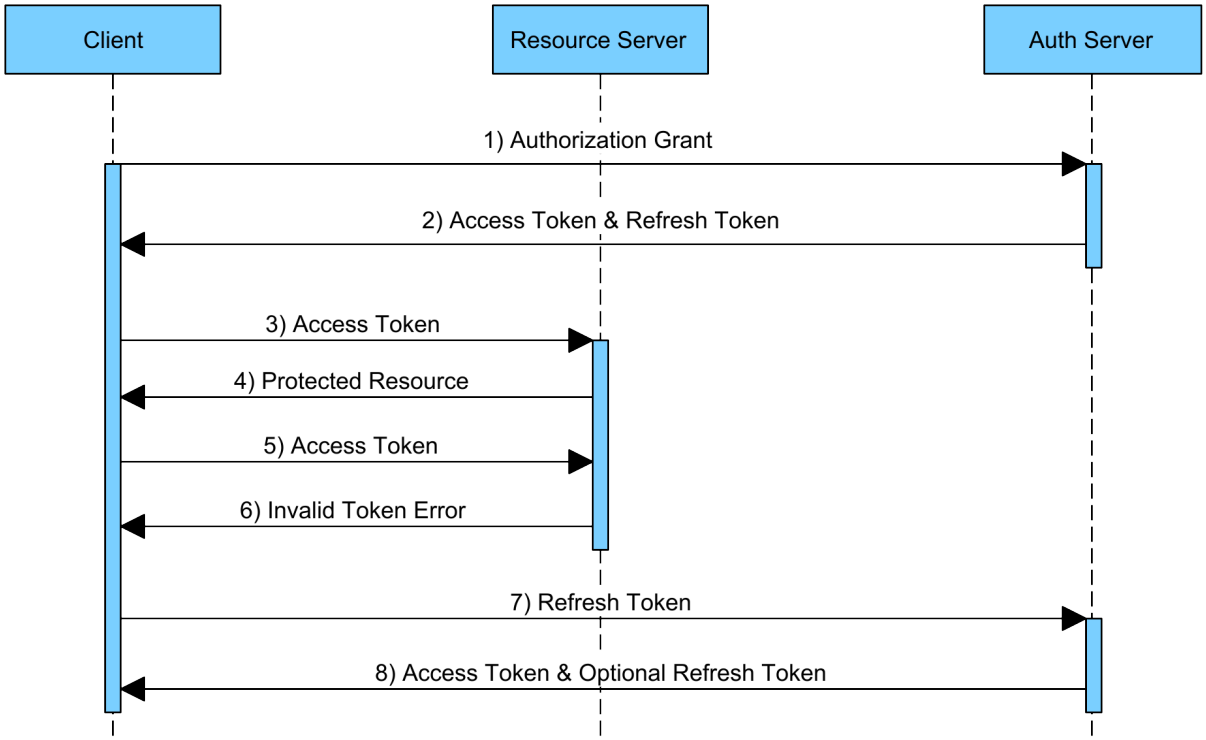


Figure 2.8: Refresh Token flow, adapted from [15]

2. The authorisation server will respond with an access token, for example a JWT, and a refresh token to be latter used.

3. The client uses the access token to get access to resources hosted in a server that trusts the authorisation server.

4. The resource server provides the resource if the access token is still valid.

5. When the token is no longer valid, if the client knows it, it can skip to step 7, otherwise, it will request a resource to the resource server.

6. The resource server will reply with an unauthorised message.

7. The client proceeds with sending the refresh token to the authorisation server, requesting a new access token.

8. The authorisation server will reply with a new access token and optionally a new refresh token.

The refresh token flow, shown in Figure 2.8, can be defined as follows:

In this flow, the steps 3 and 4 will repeat until the access token expires, after step 8, the client will resume at step 3. If the refresh token expires, then the client will start at step 1. In terms of storage, a refresh token must be kept confidential and only shared between the client and the authorisation server that issued it. The authorisation server must correlate a refresh token to the client to whom it was issued and not allow its usage from a different client.

2.6 Distributed Systems

A distributed system [48] is one that is composed by multiple single elements that work together to a common goal and are perceived as a single system. When designing a distributed system, it should be transparent to the fact that it is distributed, it should be scalable, open, and coherent. For transparency, the system should be agnostic to machine architecture, location, resource reallocation and replication. It should be easy for users and applications to access shared resources, and they should be kept consistent across the system.

Another important aspect is to keep the system scalable [48], in terms of size, meaning adding more users and resources without a big loss of performance, in terms of geographic location, meaning that, even if the users and resources are located far apart the delays are not significantly noticed and in terms of administration, meaning that it can be easily managed even if it encompasses many administrative organisations. Techniques such as distributed cache can help to have a scalable system.

Finally, a distributed system must have fault tolerance [48], meaning that if one part of it is not operational, the remaining should continue running correctly. Systems should be built in a way that they can automatically recover from these types of failures without much impact.

2.7 Critical Analysis

The concepts mentioned in this section are fundamental to the design and implementation of AuthenticationApi. Each one of them plays a crucial role in ensuring secure, efficient, and scalable authentication.

The use of SSO and SAML 2.0, allows for centralised identity management, which reduces the complexity of handling multiple authentication credentials across various applications. The JWTs provide a compact and secure way to transmit user claims, such as roles, between intervening parties, which is essential to grant users access to different services. PKCE is relevant to secure the authentication process against authorisation code interception attack. Distributed systems principles ensure that the authentication infrastructure remains resilient, scalable, and capable of serving users across different geographic locations without performance degradation.

However, these technologies have some challenges. The reliance on a central IdP in SSO systems introduces a single point of failure. If the IdP becomes compromised or unavailable, all RPs are affected. The usage of JWT requires careful attention when it comes to security configurations, such as ensuring that the JWT is properly signed by a trusted party. PKCE while effective against the authorisation code interception attack, it increases complexity in the login flow, thus making it slower. The usage of a distributed system also introduces challenges in its scalability and management.

In conclusion, the technologies discussed are essential to the development of a secure, scalable, and effective authentication system. While they offer significant benefits, careful consideration of their potential drawbacks is essential to mitigate risks.

Chapter 3

State of the Art

In this chapter, an overview of the current state of the art is given. It highlights authentication and authorisation standards, future trends, current SSO solutions and protocols. Additionally, it delves into some related work that have similarities to this project. The landscape of web authentication has evolved significantly in recent decades, driven by the need for secure, seamless, and user-friendly access to web resources. SSO solutions have emerged as modern identity and access management systems, enabling users to authenticate once and gain access to multiple related applications without repeating the login.

3.1 Authentication and Authorisation

The most common method of authentication is the combination of a username and password. Usually, the username can be a chosen one or something else such as an email address. The password consists of various characters defined by the user. The more complex the combination of characters is, the harder it is for an attacker to detect it [40]. This system has the advantage of being easy to use.

"The average person owns about twenty-five online accounts but only the half of the users have different passwords in each account" [40]. Having the same password across different accounts leads to a security risk. If an attacker discovers one of the passwords, then more accounts become compromised.

Another disadvantage is the complexity of the passwords utilised. "Extremely simple passwords, which may include the name of the user, date of birth etc. are vulnerable to phishing attacks." [40]. If attackers can in some way get the user's password, they can replay them. Unfortunately, most of the users fail to follow the recommended best practices of single factor authentication.

Two factor authentication [40] brings additional security to the authentication process by combining single factor authentication with another method of identification such as a One Time Password (OTP) or any biometric element. The second step required for authentication prevents the scenario where the password of a user is compromised, making it necessary for the attacker to have access to the second authentication mechanism. It is easy to use and implement as most people have access to a second device where they can perform this additional step. The main disadvantages are additional hardware

costs, a more complex authentication process and restricted access when one of the two authentication elements is unavailable.

MFA brings additional levels of security by integrating more than 2 authentication methods together [40]. MFA can be used in most sensitive areas, an example is accessing a vault which might require the user to present a physical card, biometrics, and a PIN. With the evolution of technology, it is becoming increasingly easier and less expensive to integrate this type of authentication, but there might still be associated costs that prevent mass adoption as well as the need for multiple manual interactions from the user to authenticate itself.

3.2 Overview of Current SSO Solutions

SSO solutions have become essential to enterprise security strategies. They provide a centralised authentication and authorisation mechanism that enhances user convenience while maintaining security.

There are various enterprise SSO solutions such as such as Okta [38], OneLogin [39], Microsoft Entra ID [6], Google Identity Platform [34] and Auth0 [19]. Additionally, there are open-source solutions like Keycloak [20] and WSO2 Identity Server [49].

Characteristics / Solutions	Passwordless Authentication	Third-Party Applications Integration	Open Source	Free to use
Okta	Yes	SAML, OIDC, SWA, SCIM	No	No
OneLogin	Yes	SAML, OIDC, WS-Federation	No	No
Microsoft Entra ID	Yes	SAML, OIDC, WS-Federation	No	No
Google Identity Platform	Yes	SAML, OIDC, social media, email, phone	No	Until 49,999 monthly users
Keycloak	Only with One-time Password	SAML, OIDC	Yes	Yes
Auth0	Yes	SAML, OIDC, WS-Federation	No	Until 7,500 users
WSO2 Identity Server	Yes	SAML, OIDC, WS-Federation	Yes	Yes

Table 3.1: SSO solutions characteristics

These platforms are adopted due to their comprehensive features, security, and integration capabilities with other applications. They support standard authentication protocols such as SAML 2.0, OpenID Connect, and OAuth 2.0, enabling secure and high-performance SSO functionality. Moreover, they offer integration with existing user directories like LDAP and Active Directory, enhancing compatibility with existing IT infrastructures. Other key features of these platforms include MFA for added security and password-less login options, which improve user convenience while maintaining high security standards.

Additionally, they provide a centralised management console for administrators. In Table 3.1 there is a comparison between some key characteristics of the above solutions.

3.3 Overview of Existing Protocols and Standards

SAML 2.0 [35], OpenID Connect [9] and OAuth 2.0 [15], are among the most widely used protocols/standards in SSO implementations. The main difference is that OAuth 2.0 is a framework that can be used to control authorisation to protected resources while SAML 2.0 and OpenID Connect are standard protocols used for federated authentication.

SAML 2.0 is traditionally favoured in enterprise environments for its robust security and extensive attribute support, making it ideal for applications requiring detailed user information [35]. On the other hand, OAuth 2.0 and OpenID Connect are preferred in modern web and mobile applications due to their simplicity and flexibility. OAuth 2.0, for example, allows third-party applications to access user resources without exposing credentials, while OpenID Connect is a layer on top of the OAuth 2.0 to provide authentication and identity management [44].

Several security considerations must be taken into account when using the above protocols, for example, SAML 2.0 can be exposed to:

- XML Signature Wrapping Attacks: SAML 2.0 uses XML for encoding messages, making it vulnerable to this attack. It can be minimized by using XML schema validation [21].
- Replay Attacks: The assertions can be replayed by a malicious entity to try to gain access. Mitigation strategies include using short-lived or one-time assertions and enforcing Hyper Text Transfer Protocol Secure (HTTPS) connections to prevent interception and message reuse [16].
- Man in the middle attacks: In this attack, a malicious entity is trying to intercept the communication between SAML parties, to prevent this, all communications should be encrypted using TLS and digital certificates should be validated [16].

The OAuth 2.0 framework [15] is one of the most used frameworks worldwide to provide authorisation [47]. It enables third-party applications to gain access to HTTP services by taking advantage of JWTs. These tokens have a restrained scope and help in creating an authorisation layer that remove from the resource owner the need to manage the authentication of the client. There are some precautions that must be considered when using it [36]:

- Token/code leakage: Access tokens can leak through logs, HTTP referrer headers or browser history when included in URL query parameters. To mitigate this risk, POST parameters should be used, logging configuration should be appropriate and token scope and duration can be reduced.
- Token guessing: Attackers may try to guess token values based on known patterns from previously issued tokens. To mitigate, tokens should have a reasonable level of entropy, should be protected by a digital signature and have short duration.
- Access Token Phishing: An attacker will pretend to be a trustable resource server but is acting as a malicious entity to collect access tokens from clients. To mitigate,

clients should not contact unknown resource servers, and they can associate the URL of the resource server with the access token.

OpenID Connect is built on top of the OAuth 2.0 framework but specifically designed to provide authentication by using cryptographically secured tokens [44]. The main addition done by OpenID Connect is adding ID tokens which, unlike access tokens, provides information about the authentication event itself, such as the user's email, name, and authentication time [44]. ID tokens can either be represented in the form of JWS or JWE. There are also security considerations in OIDC [7], such as:

- IdP Mix-up Attacks: Attackers will trick the RP into authenticating a user with an honest IdP, but thinking it is authenticating with a malicious IdP. Then, because of this confusion, the RP will use the authorisation code or access token received from the honest IdP at the malicious IdP, giving this malicious entity access to impersonate the user. A mitigation strategy is making the IdP include its identity in the authorisation response and make the RP validate it.
- Transport Layer Security: The security of OpenID Connect depends on the usage of HTTPS connections to prevent interception and tampering attacks, preventing the stealing of sensitive information such as tokens or user credentials.

3.4 Modern Authentication Trends

The landscape of web authentication is continuously evolving, with new trends and technologies emerging to address the limitations of traditional methods and improve them.

Password-less authentication is gaining traction as it enhances security and user convenience. Several ways exist today to provide this type of authentication [41], such as social login, magic links, one-time codes, biometrics, or passkeys. Social logins, enable users to login using an existing social media account. Magic links is a type of authentication were the user inputs their email address, a login link will be generated and sent to that email. One-time codes can also be sent by email or SMS to the user trying to login. Biometric authentication is based on biometric features of the user such as fingerprint. Passkeys [12] use public-key cryptography and provide a secure credential-based authentication linked to a specific device, normally the authentication is done using the device login method, such as biometrics.

MFA [40] is nowadays considered a critical component of any robust authentication system. By requiring multiple forms of verification, such as a password combined with a fingerprint scan or a one-time code, MFA significantly reduces the risk of unauthorised access.

3.5 Related Work

There are multiple works done around SSO authentication with different systems created, in this section, some of the most recent ones are described.

In [11], a cloud-based authentication system for fast and secure SSO authentication was implemented, called D-IAM. It leverages SSO, OAuth 2.0, and multi-thread programming

allowing for a high number of user access requests in a cloud platform environment. The implementation consists of three main modules:

- Identity and SSO Authentication: Verifies user identities based on anonymous credential checking and interacts with the OAuth system to request SSO tokens.
- OAuth system: Generates JWT in response to requests from the Identity and SSO Authentication Module.
- Authorisation system: Maintains access control policies and dynamically enforces user privileges during application access. Resource owners can define and modify policies existing in an access policy database, which is also accessed by the OAuth system.

A resource database exists to keep a list of resources that are bound to each user. Another database called personally identifiable identity database stores the hash of user credentials to permit user authentication. A log system keeps track of the activities happening in the system. A comparison of this system with Keycloak [48] was done and a significant performance improvement was noticed under high volumes of authentication requests.

In [10], a decentralised SSO and identity access management system, named D2-IAM, was designed to enhance security and efficiency in multi-application cloud environments. The system leverages blockchain technology to ensure secure and traceable access control. The system is divided in multiple components:

- Access Control Service Gateway: Core system service responsible for managing access requests. It liaises with smart contracts to conduct SSO authentication, token revocation, enforcing authorisation and invoking preventive access control.
- SSO Authentication smart contract: Responsible to authenticate users and generate a SSO token that enables access to various resources. It connects to the personally identifiable information and resources databases.
- Authorisation smart contract: Responsible for validating and enforcing user privileges.
- Token revocation smart contract: Responsible for revoke tokens when a new one is generated or when requested by a system administrator.
- Preventive-based access control smart contract: It allows for user session monitoring and managing, for example, it terminates a session of a user that is violating some defined rule.
- Resource Database: Contains the resources and the respective access policies.
- Personally Identifiable Information Database: Stores users hashed credentials.

The proposed solution is resistant to a central point of failure due to its decentralised nature. The system was compared with 3 others showing a higher throughput and less time to authenticate the users.

In [2] user behaviour history was integrated into the login process. A point-based system was implemented to determine whether the user is trustworthy, depending on that it will or not be able to instantly login. The proposed solution considers factors such as the number of password attempts, IP address consistency, and user agent type. If the user

logs in using different IP address or User-Agent, it will negatively impact the number of points, as will a high number of login retries. If the user does not have enough points a login challenge will be created based on 3 numbers and the correct number sent by email, the user then must select the correct number to login.

In [14], an enhancement of the user privacy while using SSO authentication is proposed by integrating PPID and Intel Software Guard Extensions (SGX) to prevent identity providers or collusive relying parties to track the user activities. The introduction of PPIDs prevented collusive RPs from linking user login activity, but it did not prevent the IdP from doing the same. To solve this, it was proposed that the PPIDs be generated by user-controlled SGX enclaves, which can prove its run-time integrity to remote entities, instead of the by the IdP. The PPIDs are generated in the user machine.

In [28], the integration of MFA with SSO and SAML is addressed. In this implementation, the IdP communicates with an authentication server that provides authentication techniques by interacting with MFA technologies. This integration aims to enhance security by requiring multiple forms of authentication, thus making it harder for attackers to gain unauthorised access.

3.6 Future of Authentication

As technology advances, the future of authentication is set for significant transformation driven by emerging trends and innovations. Many try to leverage biometric characteristics to perform authentication. In [8] an analysis was done of 108 Electroencephalography based user authentication experiments from 1998 to 2022. The goal of Electroencephalography based authentication is to recognise the user based on their brain activity. As currently, devices such as smartwatches and other Internet of Things (IoT) devices are widely used, in [33] a study was conducted on the challenges of biometric authentication in this IoT devices.

As currently, devices such as smartwatches and other IoT devices are widely used, in [33] a study was conducted on the challenges of biometric authentication in this IoT devices. It points out 7 authentication methods that are still under investigation: signature, keystroke dynamics [42], gait, activity-centric metrics [42], voice, breath, and heart biometrics.

- Signature biometrics: Captures dynamic features such as velocity, acceleration, and pressure based on user signature.
- Keystroke dynamics: Analyses the typing patterns of users, including the timing and pressure of keystrokes.
- Gait: Uses the walking patterns of individuals for authentication.
- Activity-Centric: Relies on specific day to day activities performed by users, such as running, sitting, standing, sleeping, and walking.
- Voice Recognition: Identifies users based on their unique vocal features.
- Breath Biometrics: Utilises the breathing patterns and rhythms of individuals.
- Heart Biometrics: Measures heart-related metrics such as ECG and PPG signals.

These techniques are still under active development, with ongoing research focusing on improving their accuracy, robustness, and usability in various environments. The continuous advancement in sensor technology, data processing capabilities, and machine learning contributes to the evolution of these techniques, making them increasingly viable for real-world applications.

3.7 Critical Analysis

The analysis of various authentication methods, including SSO, MFA, Single-Factor Authentication, and emerging trends like password-less authentication and biometric-based methods, is directly relevant to the project. SSO solutions, supported and based on protocols like SAML 2.0, OpenID Connect, and OAuth 2.0, are important for implementing a seamless and secure authentication system. The adoption of SSO provides several advantages, including enhanced security, user convenience, and centralised management of identities. The ability to use one login reduces the risk of weaker passwords protecting the user and easing accesses.

Understanding these protocols' strengths and weaknesses allows for the design of a robust system that meet the security demands of modern web applications. Furthermore, the consideration of modern trends such as password-less authentication and biometric verification offers insights into future-proofing the system, ensuring it remains relevant and adaptable as technology evolves.

Even though various SSO solutions currently exist, they are mostly not free and do not meet the specific corporate requirements of this project, which include private code ownership and integration with some legacy systems that are still in use as of today.

While the immediate focus of this project is on implementing a secure and efficient SSO solution, there is potential for future integration of advanced authentication methods such as biometrics, password-less authentication with magic links, or passkeys. These methods offer additional layers of security and user convenience, aligning with modern web trends.

When analysing other SSO projects, it is common to see the usage of SAML and JWT in the implementations as the base for authentication and authorisation. Some implementations leverage the distributed nature of blockchain technology, while some others use more traditional technologies. Efforts were also undertaken to increase the user privacy when using a SSO infrastructure [14], [50], even though, that, for example, in [14], a specific hardware with Intel SGX is needed to prevent IdPs from tracking the user history.

In conclusion, the above state of the art reflection provides a foundation that will be considered throughout the development of this project. While there is always room for improvement and future enhancements, this project aims to bring together the mentioned concepts to create a practical and effective solution to a real-world scenario that meets the needs of a specific corporate environment.

Chapter 4

Solution Design

This chapter presents an overview of the system architecture, all the login flows, how all the components relate to each other and the planned timeline. The design of the AuthenticationApi addresses the challenges of secure, scalable, and efficient web authentication across multiple applications and platforms within a corporate team. Each design decision explained in this chapter prioritises scalability, security, and user experience, ensuring that the system meets current needs while remaining flexible enough to accommodate future enhancements. An important focus was also given to simplifying the integration process, ensuring that web applications can easily interact with the API.

4.1 Conceptual Architecture

The AuthenticationApi is a single component that interacts with other surrounding components, illustrated in Figure 4.1. This interaction grants authentication and authorisation for every client web application and is also used to authenticate windows processes running with a technical account. The components present in this architecture are the Global Adaptative Access Platform (GAAP), AuthenticationApi, Client Web App (being any app), frontend and backend libraries for web authentication, Windows Technical Process and Active directory that will provide windows Network Level Trust Manager (NTLM) authentication for processes running on windows machine with a technical user.

During this project, AuthenticationApi and the libraries were developed. GAAP service already existed and was leveraged to provide SSO authentication capabilities. The client Web App can represent any web application that uses AuthenticationApi. The Active Directory is used to provide NTLM authentication to windows processes, which are running using a technical account and so, having no browser cannot utilise the same SSO login method as web apps.

GAAP was created for the company and is used to provide several authentication methods such as biometrics, SMS, email OTP or mobile notifications. With this service the company was able to adopt a “security by design” approach and standardize the accesses to applications and portals. This makes it that all applications created must adhere to the norm by utilising GAAP platform. It will be with GAAP that the user will be authenticated, thus, as a trust relationship exists between GAAP and the rest of the company services, users will be authenticated across other applications without additional effort.

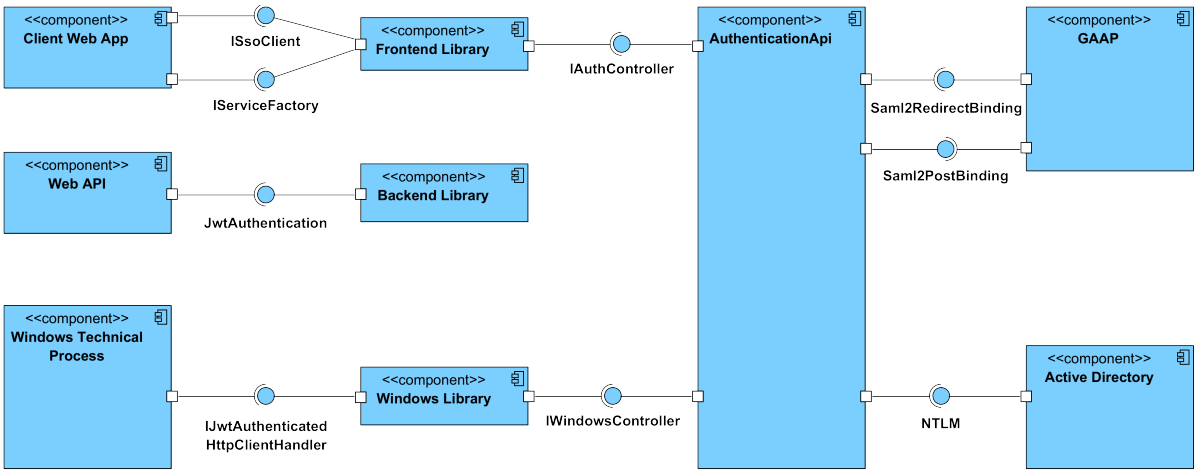


Figure 4.1: Component Diagram

The internal frontend and backend libraries provide authentication to any web app or web API that wishes to contact AuthenticationApi. They allow for an out of the box integration with AuthenticationApi by simply doing some configurations. The frontend library is intended to be used in frontend web applications. It will leverages the authentication and authorisation standards so that the usual attributes and features can be used during the application development, standardising permissions, and accesses. The backend library aims at providing authentication for web APIs that aim to be consumed by the frontend apps using the frontend library. These libraries abstract all the authentication and setup into a few methods and configurations, easily usable and documented.

The Client Web App, Web API and Windows Technical Process will be the consumers of the authentication provided by AuthenticationApi. The main difference between them lies on the libraries they use. Client Web App, as it is any web frontend application, uses the frontend library, the Web APIs will use the backend library and the windows technical process, as it is a process running with a technical user, thus not having a browser will use a separate library, the windows library. Client Web App and Windows Technical Process are the ones that will be making authenticated calls to the Web APIs. The Web APIs are as simple as service providers that require a JWT token emitted by AuthenticationApi to be accessed.

AuthenticationApi acts as a IdP for the team web applications, APIs and processes, but as a SP for GAAP. It is responsible for the login process, serving not only as an abstraction to connect to GAAP, but also it will produce an authentication response with all details that the web app or process needs.

4.2 SSO Login Flow

The AuthenticationApi enables users to login into multiple apps, through a SSO process as shown in Figure 4.2. This process involves three endpoints: login, callback, and token. The Login endpoint initiates the SSO login flow via a browser redirect in step 3. The callback endpoint receives POST requests from GAAP containing user details such as email address. The token endpoint is called in the final steps to retrieve the JWT, completing user authentication process. The steps needed to perform authentication are

outlined below and illustrated in Figure 4.2.

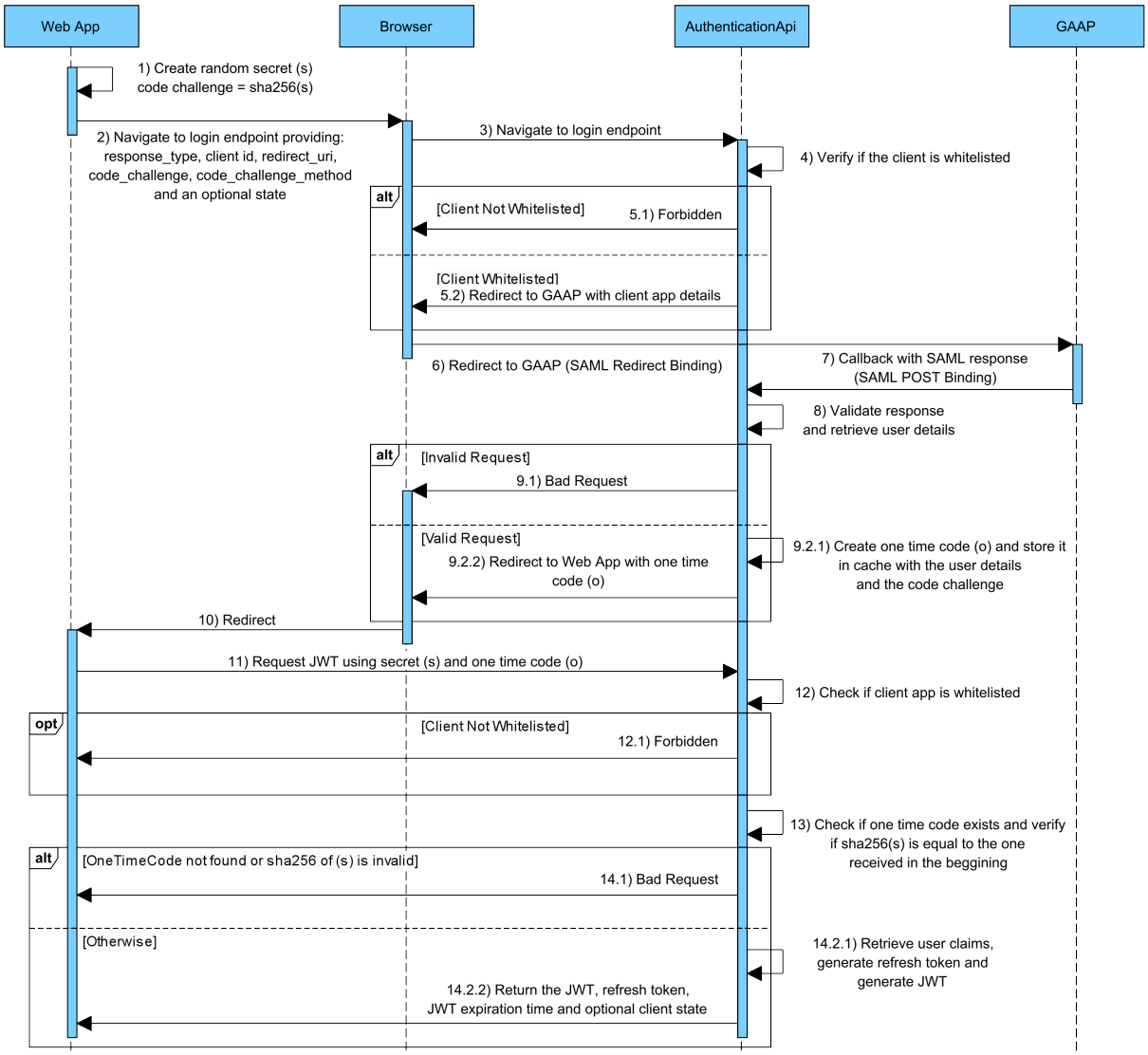


Figure 4.2: SSO Login flow

In step 1) the client web application, also referred as client, generates a random string of 64 characters[1], referred to as the secret (s), and the respective SHA-256 hash. This secret is stored locally on the client side.

In step 2) the web application redirects the user to AuthenticationApi login endpoint providing various parameters:

- Response type: It Indicates whether the login type uses a refresh token or a one-time code.
- Client ID: The unique identifier for the web application.
- Redirect URI: The address to which the one-time code will be sent via a redirect in step 9.2.2).

[1]Possible characters are alphanumeric, capital and lower case, dash (-), dot (.), underscore (_) and tilde (~)

- Code challenge: The SHA-256 hash of the randomly generated secret (s).
- Code challenge method: Represents how the code challenge was generated. In this implementation, only SHA-256 is supported, but according to [1] it is also possible to implement "plain", where the secret and code challenge are the same.
- Client state (optional): An arbitrary string that the web application wishes to receive back in step 10).

In step 3) the browser is simply redirected to AuthenticationApi according to step 2).

In step 4) AuthenticationApi verifies if requesting client web application is whitelisted, and if the provided redirect URI matches the whitelist entry. If the validation fails, a forbidden response is sent in step 5.1). If the validation is successful, the user is redirected to GAAP in step 5.2), using SAML 2.0 Redirect Binding, including the details provided in step 2) as a RelayState query [31].

In step 6) the browser is redirected to GAAP according to step 5). In step 7) GAAP sends a POST request to the callback endpoint of AuthenticationApi, using SAML 2.0 Post Binding. The body of the POST form request contains the RelayState query in clear text and the SAML Response, encoded in base64, which is an XML document containing the attributes of the user that is logging in, such as email and name, being the email the most important one, latter used to retrieve the user login which serves as a unique identifier for all the users across the team's applications.

In step 8) AuthenticationApi validates the response using the SAML metadata file, verifying response integrity, validating the signature, audience, and issuer. It extracts the SAML attributes and subject present in the assertion converting them into key value pair claims. The RelayState query is also retrieved, giving AuthenticationApi the context needed to continue with the login. If any validation fails, then, a bad request is sent in step 9.1), otherwise, the flow proceeds in step 9.2.1).

During step 9.2.1) a one-time code is generated, valid for 1 minute, this code will be used by the web app to finalise the login in step 11). The code formed by a random bytes sequence with 16 length transformed into a GUID, giving 32 alphanumerical characters. This code will be stored in local and distributed cache along with user claims, code challenge, code challenge method, client state and the timestamp of code generation. In step 9.2.2) and 10) the browser is redirected to the specified redirect URI from step 2) with the one-time code as a URI parameter.

In step 11) the web application uses the one-time code and initially generated secret (s) to request a JWT access token from the 'token' endpoint.

In step 12) AuthenticationApi checks if the requesting web application is whitelisted, if not, a forbidden response is sent in the optional step 12.1).

In step 13) the one-time code and remaining details stored in step 9.2.1) are retrieved from cache. The SHA-256 hash of the secret (s), provided in step 11), is compared with the cached code challenge. If they do not match, a bad request is issued in Step 14.1.

Upon successful validation, in step 14.2.1), a JWT is generated containing the user login, user roles and the user group. A refresh token is also created for quicker re-authentication in the future. The refresh token is stored in a distributed cache along with the client id and the user login retrieved from the email provided by GAAP in step 8).

To end the flow, in step 14.2.2) the JWT, the refresh token, the JWT expiration time and the optional client state are sent to the web application completing the SSO authentication. At this point, the web app can use the JWT to request protected resources from multiple services that trust AuthenticationApi access tokens.

4.3 Refresh Token Login Flow

Client apps that have previously logged in can use AuthenticationApi to quickly obtain a new access token (JWT) by leveraging refresh tokens. A refresh token login greatly enhances the speed of access token (JWT) retrieval as the flow is less complex and there is no need to redirect between the app, AuthenticationApi and GAAP. As illustrated in step 14.2.2) of Figure 4.2, a refresh token is issued alongside the JWT, valid for a configurable duration. With refresh tokens clients can directly request a new access token without re-authenticating through the complete SSO login flow. The refresh token login flow is represented in Figure 4.3.

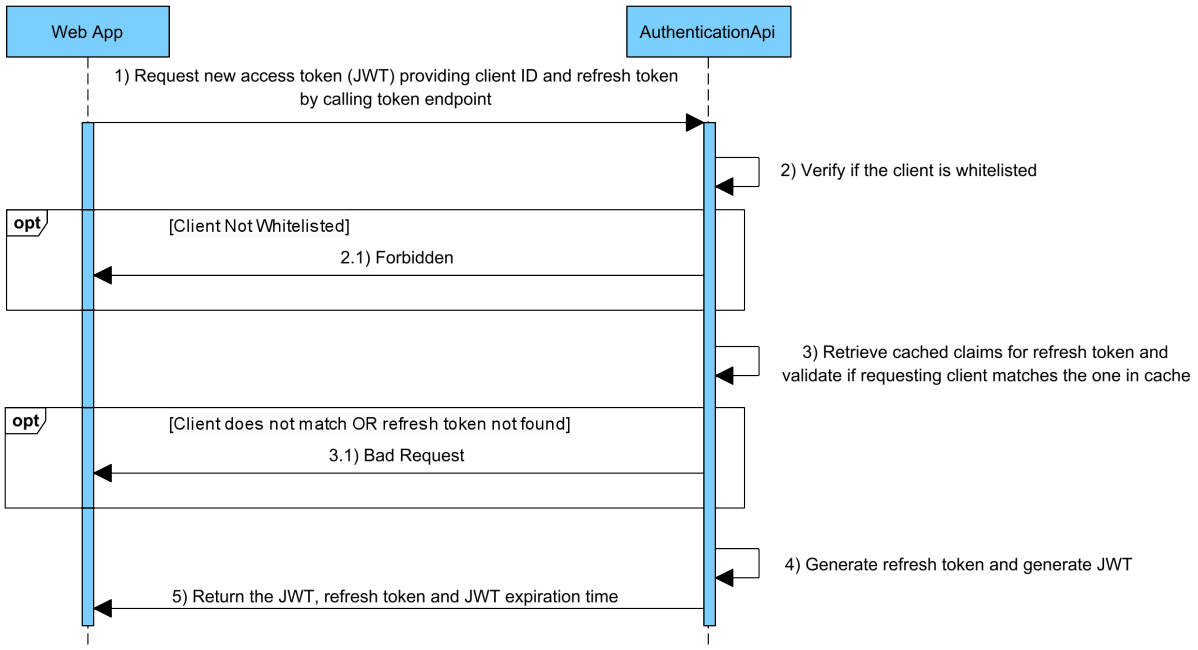


Figure 4.3: Refresh Token Login Flow

In step 1) the user requests an access token (JWT), providing the client ID and the refresh token by calling the token endpoint, which is the same as the one mentioned in the SSO login flow. The main difference is that this request contains the refresh token instead of the one-time code and a flag to signal this is a refresh token login.

In step 2) AuthenticationApi validates whether the client is or not whitelisted, returning a forbidden response in case it is not in step 2.1).

In step 3) if the provided client ID matches the original client ID, stored when the refresh token was created, AuthenticationApi proceeds to next step. Otherwise, it returns a Bad Request in step 3.1).

In step 4) AuthenticationApi retrieves cached claims, current roles, and group of the user to generate a new JWT. Additionally, a new refresh token will be generated.

Finally, in step 5) a response is sent to the client containing the JWT, the refresh token and the expiration time for the JWT.

4.4 Windows Login Flow

Web applications typically use either the SSO login flow or the refresh token flow to retrieve a JWT. However, for authenticating technical (non-human) users who run processes without needing a browser, windows NTLM login is leveraged to validate user credentials instead of GAAP. These processes typically include overnight scripts or background tasks that process data at specific time intervals and run on a server without the need for human user intervention.

This flow is illustrated in Figure 4.4. In step 1), the process will request a JWT and provide the client ID by calling the windows token endpoint. In steps 2.1) and 2.2), NTLM authentication is performed to ensure that the requesting user exists and to validate its credentials. If NTLM authentication succeeds, then in step 3) and 4) validations are performed to ensure the user has permission to use this flow and if the client is whitelisted. In case one of the validations fail, a forbidden response is sent in step 3.1) and 4.1). If all validations are successfully, then, in step 5) the token is generated and along with its expiration time is sent to the process. Once the flow is completed, the process can make authenticated HTTP requests to the desired services using the obtained JWT.

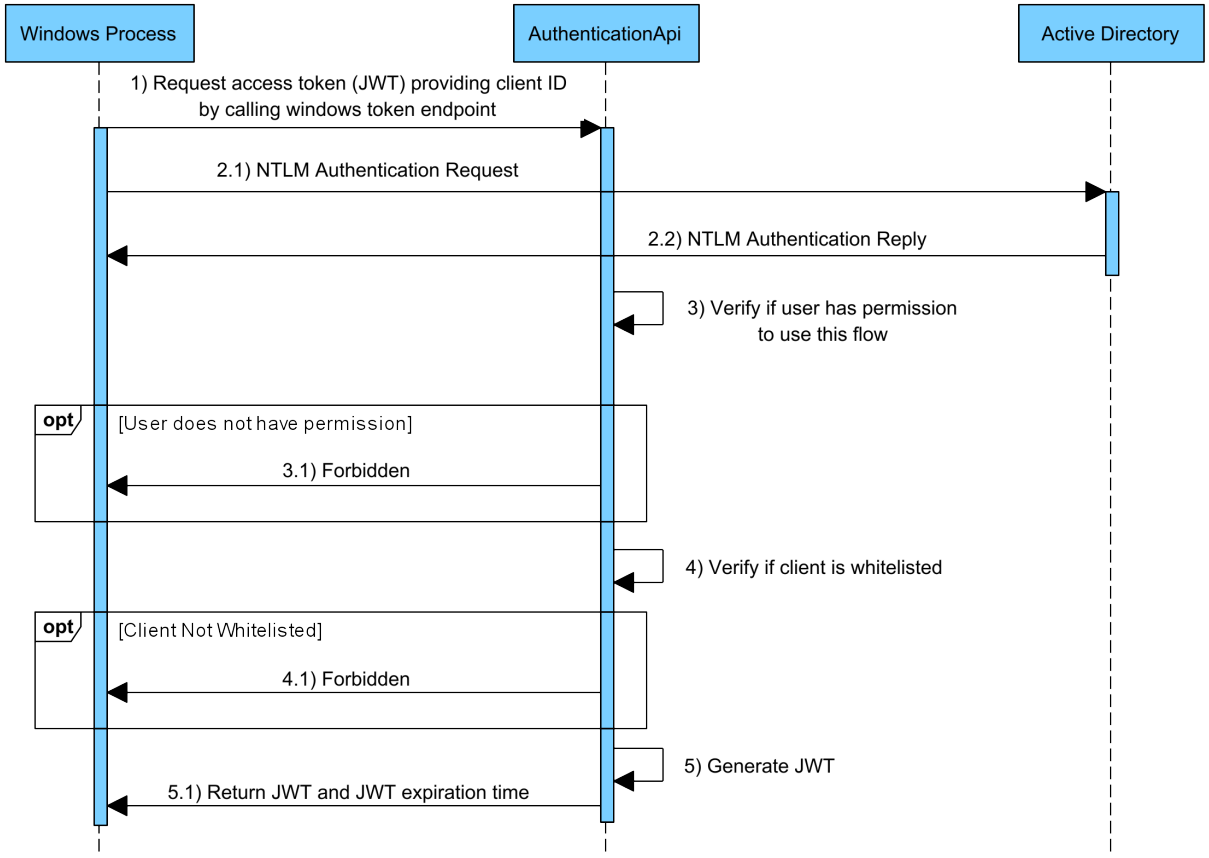


Figure 4.4: Windows Login Flow

4.5 Distributed Instances

AuthenticationApi is designed to serve multiple web applications across the globe. To achieve this, its design must encompass scalability at its core. To this end, SQL distributed cache and MyCloud load balancers were utilised.

The plan is to have AuthenticationApi running in Paris and New York as illustrated in Figure 4.5. In both locations, two machines in separate data centres will be used to run this application. The load balancers, created using the internal MyCloud instance, allow for adding as many machines as needed behind it. If the authentication call volume increases to the point that two machines are too slow to handle all the traffic, a third machine can be easily added to the load balancer of the impacted location. This way the system is easily scalable in case of need and resilient in case one of the data centres goes offline.

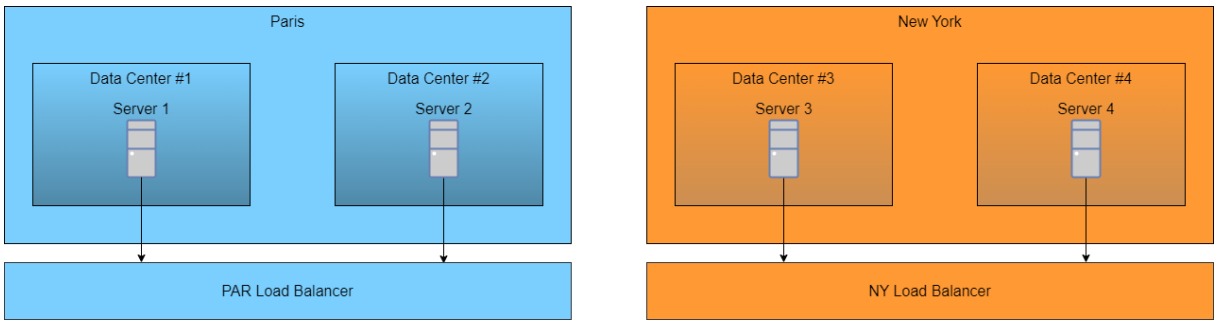


Figure 4.5: Load Balancer Setup

Using a distributed cache is not only beneficial, but mandatory due to the nature of the SSO Login flow explained in section 4.2 This is necessary because the authentication process can start on Server 1 but be finalised on Server 2. For example, in step 3, the server handling the request might be machine 1, but in steps 7 and 11, it could be machine 2 as the load balancer does not guarantee the same machine will be used by the same calling user. With a distributed cache in place, as shown in Figure 4.6, we can guarantee that even though the flow might pass thru multiple servers, all of them are able to maintain the same authentication context throughout the entire login.

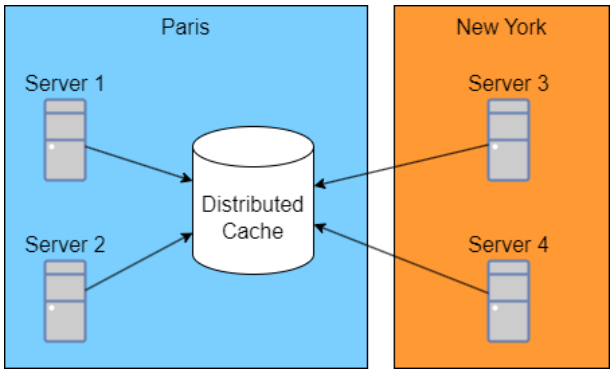


Figure 4.6: Distributed Cache Setup

As seen in Figure 4.6, there is only one distributed cache serving both locations, which will inevitably slow down the login for New York users. This limitation comes from the

current team infrastructure, which only supports SQL Server as the distributed cache. Furthermore, the SQL Server setup includes a single write database located in Paris, with read-only instances elsewhere, making it impossible to create a dedicated cache for New York users at present.

Ideally, a distributed cache should be created for each location where Authentication-Api operates. The system is designed with this future enhancement in mind. By using a distributed cache abstraction[2] it makes it easier to switch to alternative distributed cache technologies, such as Redis [43], [37], with minimal changes. Only the startup configuration and certain configuration files would need updating if new cache instances or technologies were introduced.

By utilising load balancers and distributed caches, it is possible to scale up the system as needed. Multiple machines can be running the API, and more distributed caches can be created in other locations to enhance resilience and performance.

4.6 Planned Timeline

The development of AuthenticationApi is structured around key milestones and deliverables, each targeting a specific feature or improvement. This roadmap spans from mid-2023 through the third quarter of 2024, outlining goals and their respective time frames.

By the end of 2023, the primary goal is to deliver the first working version of the API, along with the associated connection libraries. This version will only support SSO login flow, laying the groundwork for the API's authentication functionality. Upon ending this milestone web applications will be able to use the API to authenticate their users.

In the first quarter of 2024, the focus will shift toward expanding the API's robustness and scalability. Integration testing and unit tests will be added to cover the core functionalities of the API, ensuring a higher level of reliability, and minimising the risk of future issues. Additionally, the API will be made scalable by introducing distributed cache, allowing multiple instances to run concurrently. To set up for future enhancements, the whitelisted clients will be moved from program settings to a dedicated database table, which will eventually be managed through a web application.

The second quarter of 2024 will focus on releasing the second version of the API and access libraries, ensuring that it conforms with industry standards, such as Request For Comments (RFC) 7636 [1]. This version will introduce key functionalities like the refresh token login flow and Windows login, expanding the API's capabilities to support both human and technical users. In parallel, a web application will be developed to allow for the management of the client whitelist, enabling the team to perform CRUD operations through an intuitive interface rather than manual database modifications.

During the third quarter of 2024, further enhancements will be introduced. Structured logging will be implemented to enable better monitoring and analysis of the API's performance and user activity. In addition, the API will be deployed in a second location outside of Europe, New York, ensuring improved performance for users in other regions by reducing latency and enhancing global accessibility.

[2]https://learn.microsoft.com/aspnet/core/performance/caching/distributed

4.7 Critical Analysis

The design of AuthenticationApi presents an approach to solving the challenges of secure, distributed, and scalable authentication within an always evolving corporate environment where the system was urgently needed within a specific team as there was nothing in place to authenticate web users, so web apps could not be developed.

The focus of the design was seamless user experience which was possible by leveraging GAAP as an identity provider. Since users are already authenticated in GAAP, there is no need to input usernames or passwords during the SSO login flow, which eliminates friction in the login process and allows users to access services quickly and efficiently. Another critical design feature is the development of access libraries for both frontend and backend web applications. These libraries greatly simplify the integration with AuthenticationApi, reducing the risk of integration errors, accelerating the development process and standardising authentication across all web apps. The refresh token functionality further enhances the user experience by automatically refreshing tokens in the background, a process that is completely transparent to the user and ensures uninterrupted access to web applications. In addition to human user authentication, the Windows Login flow, enables technical accounts to authenticate themselves securely. This is particularly useful for overnight scripts that run on an automatic schedule.

However, the design also faces certain limitations, particularly regarding the distributed cache implementation. The project is constrained by the internal infrastructure, which currently only provides a single SQL Server based cache located in Europe. While this solution works well within the available resources, it introduces limitations when scaling across regions. For instance, there is no available SQL Server in New York containing a write data base, which may introduce potential latency issues for applications or users located in that region.

Overall, this design focuses on securely, offering the best possible experience to both users, who will use the web applications, and developers which will create them. It also faces challenges around the current infrastructure that will need to be addressed to ensure optimal performance in a distributed, multi-region environment.

Chapter 5

Implementation

This chapter describes the implementation of AuthenticationApi, using C# and .NET Core, detailing the components, API endpoints, and supporting features that were developed to meet the project's authentication and security requirements. It covers the endpoints supporting all the mentioned login flows, the creation of access libraries for easy integration, the creation of the JWTs and the development of a management console to ease client whitelist management. Additionally, this chapter delves into the logging, DevOps setup, and the release process, offering a comprehensive view of how the system is monitored, built, and deployed.

5.1 API Endpoints

To support the three login flows mentioned in the previous chapter, four API endpoints were created. For the SSO Login Flow, there are three endpoints, login, callback, and token. The Refresh Token Login Flow shares the token endpoint with the SSO Login Flow. Additionally, for the Windows Login Flow there is a separate called windows token endpoint. The specific actions performed by these endpoints are detailed further in the Solution Design chapter.

- Login: This endpoint serves as the starting point of the SSO Login Flow. It redirects the browser to GAAP for user authentication and performs initial client whitelist validation.
- Callback: This endpoint handles POST requests from GAAP, storing the user details and login information in the distributed cache. It then generates a one-time code that can be used to retrieve the cached login information via the token endpoint.
- Token: This endpoint generates JWTs, either by validating a one-time code and a secret or by using a refresh token to issue a new access token.
- Windows Token: This endpoint allows technical (non-human) users to request a JWT by leveraging NTLM windows authentication.

The API swagger documentation is shown in Figure 5.1, where all the four endpoints are represented twice, as the API supports multiple production versions. The endpoint without "trafic/v2" prefix use a predefined version set in the API configuration, which in this

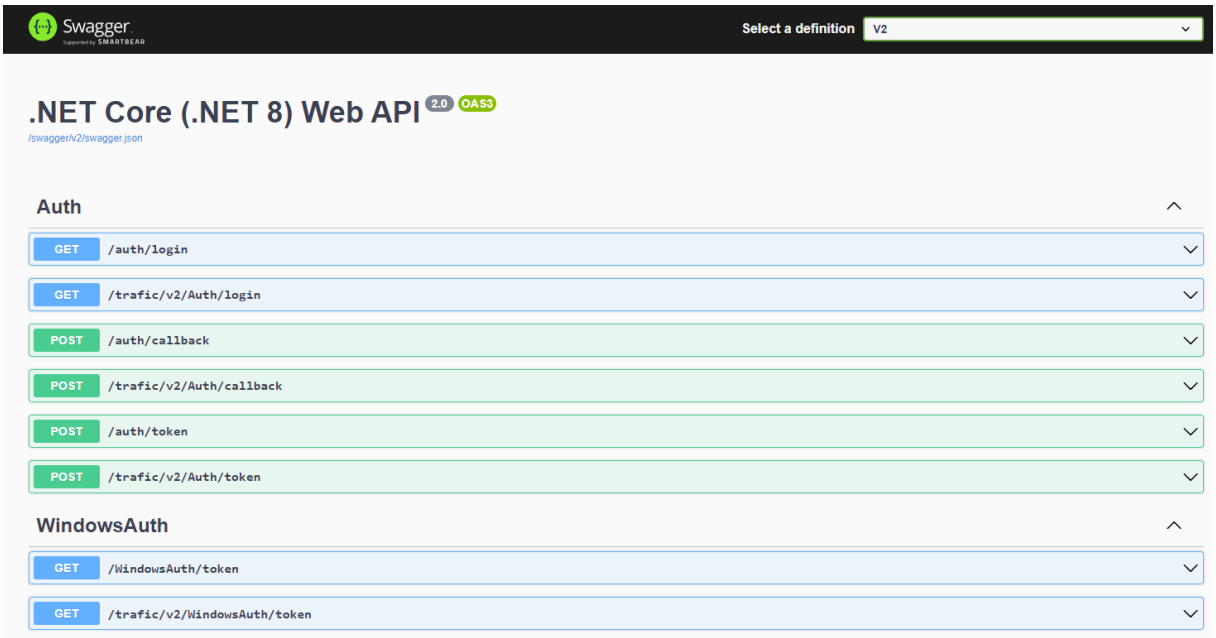


Figure 5.1: AuthenticationApi Endpoints

case is version 2. This versioning feature was especially important during the migration from version 1 to version 2, allowing both versions to coexist. It allowed new applications to adopt the updated version while non-migrated applications continued operating with the older version, ensuring a smooth transition without service disruption.

5.2 Connection Libraries

To abstract the connection to AuthenticationApi, three internal NuGet libraries were created, as illustrated in Figure 4.1. These libraries, developed in C#, are used in different contexts, and provided different functionalities to achieve the same goal, authenticate users.

The frontend library was created to be used by Blazor[1] web applications. This library provides several features:

- Automatic login: The NotAuthorizedComponent, which can be placed in the root of the web application, automatically initiates the login process if the user is not authenticated or displays a customizable view if they are authenticated but unauthorised. During the authentication process, an optional display component can be defined.
- Automatic login refresh: By utilising Delegating Handlers, this feature ensures that HTTP requests include a valid JWT. If the token is set to expire in under 30 seconds, the handler automatically requests a new JWT. Two handlers are provided: one for general HTTP clients and another for Refit[2] , simplifying API calls. The handlers append the JWT to the HTTP request's authentication header. Additionally, a factory class was created, providing a REST service based on any Refit interface,

[1]Blazor web site – https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor
[2]Refit source code – https://github.com/reactiveui/refit

using the Refit handler to ensure all HTTP calls are authenticated.

- Standard authentication state provider: The library includes a custom authentication state provider, allowing client web applications to utilise existing Blazor authentication tags and components[3]. These tags and components can be easily integrated into any page or component to manage user access based on roles and policies.

- Out of the box components: The library offers components to facilitate authentication configuration. One of them is the reload rights button, an abstract Razor class that contains the logic to request a new JWT using the authentication client mentioned bellow. Web applications can implement the visualisation for this button and specify actions to execute post-authentication, such as displaying a notification. The library also provides a page, OneTimeCodePage, and a component, OneTimeCodeComponent, to handle and request JWTs using a one-time code.

- Authentication client: This client class simplifies the handling of all login aspects in a frontend web application. It enables web apps to request a new JWT either forcefully or when it is about to expire. It also allows obtaining a JWT from a one-time code, useful for custom implementations of the one-time code receive page. Additionally, it includes a method to redirect the browser to the URL the user was on before the SSO authentication began. These methods are available for custom implementations, though the library already provides all necessary pages and components, making the use of this client not necessary.

- Local and session storage usage: Client applications can store JWTs and refresh tokens in either local storage or session storage. Using session storage requires more frequent logins, as tokens are cleared when the browser is closed, necessitating a new SSO login each time the browser is reopened. Local storage, on the other hand, retains the refresh token, which is valid for up to seven days, allowing faster authentication through the refresh token login process, which is quicker than a full SSO login.

The backend library is integrated into web APIs that provide secured services to client web applications. It is designed to simplify the setup of AuthenticationApi within .NET Core web APIs. By simply adding a single line of code in the startup file and the respective configuration in the JSON settings file, developers can enable Bearer authentication seamlessly. The library configures all necessary authentication parameters, such as the token issuer (AuthenticationApi) and the public key used for validating JWT signatures.

The Windows library is designed to enable technical processes to authenticate themselves, as opposed to web app users. This library is intended for authenticating technical accounts used to run processes such as overnight scripts or background tasks. Similarly to the frontend library, it provides an HTTP client handler that can be used in any HTTP client to ensure that HTTP REST requests are always authenticated. If the token is expired, the handler automatically requests a new JWT and attach it to the authorisation header as a bearer token.

[3]Blazor authentication and authorisation – https://learn.microsoft.com/aspnet/core/blazor/security

5.3 Clients Whitelist

The client whitelist allows for controlling which apps can use AuthenticationApi login. To be part of the whitelist, a setup must be done for each app. In this setup, the properties shown in Figure 5.2 must be defined. These properties are:

- Client ID: A unique identifier for each application. Even if the app is deployed on multiple machines, it must use the same Client ID across all instances.
- Token Expiration Minutes: This parameter defines how long, in minutes, the JWT will remain valid from the moment of its creation. Once this time elapses, the token expires, and the client will need to request a new one.
- Client URIs: A list of valid URIs associated with the application. During the login process, only these specified URIs are allowed, preventing unauthorised redirection or usage from unauthorised applications.
- Client Rights: A list of rights the client application requires to manage user access to various features. The rights scheme was already implemented and used by other applications before this project, to conform with it, the fields Application and RightType are needed. The Application entity simply holds an application name. The RightType entity is associated with 1 application and specifies a name for a right, if it is active and a description. The UserRights entity contains a list of all current users' rights.

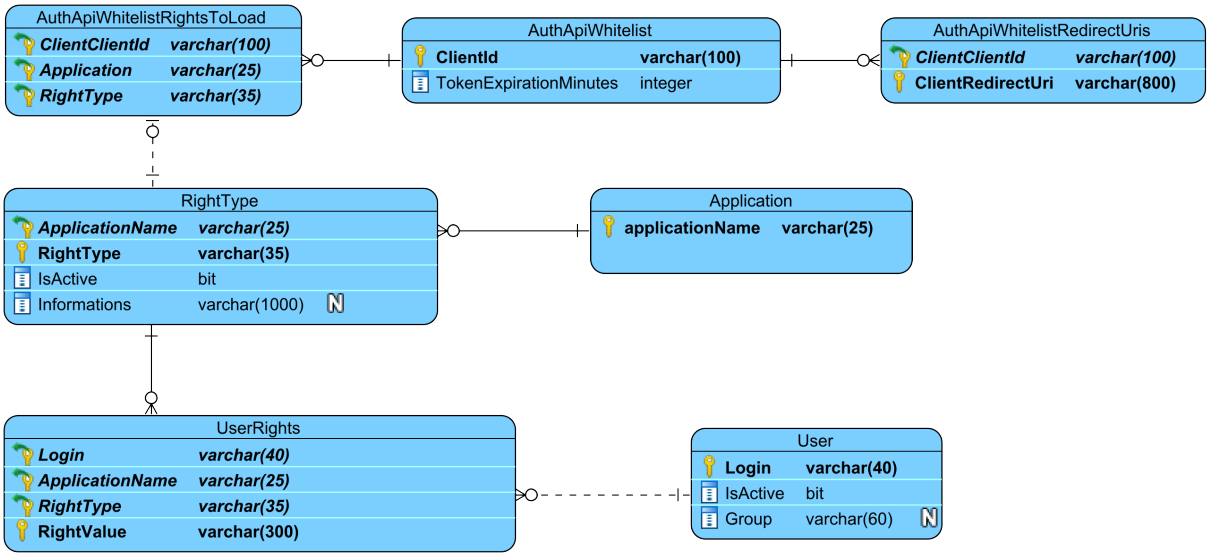


Figure 5.2: Client Whitelist ER diagram

5.4 Management Console

A Blazor web application was developed to manage the client whitelist, enabling CRUD operations (Create, Read, Update, Delete) for client configurations. The tool manages three tables, illustrated in Figure 5.2: AuthApiWhitelist, AuthApiWhitelistRightsToLoad and AuthApiWhitelistRedirectUris. A screenshot of the tool is shown in Figure 5.3. This interface allows users to view all client details, create a new client by clicking the "Add

Client" button, delete a client by clicking the red icon, and update client information by adding or removing rights, modifying URIs, or changing the expiration time of the JWT token in minutes.

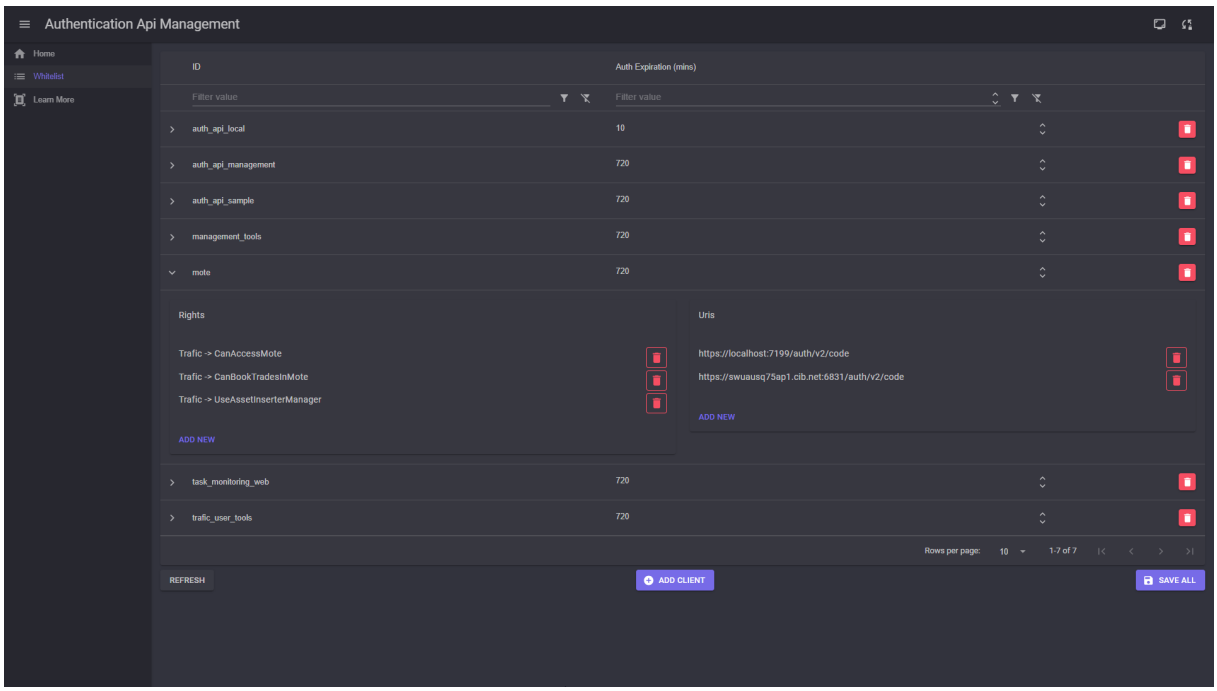


Figure 5.3: Management Console

5.4.1 Endpoints

To support these features, an API was developed for consumption by the Blazor app. This API comprises two controllers: ApplicationController and WhitelistedClientController, with their respective endpoints illustrated in Figure 5.4.

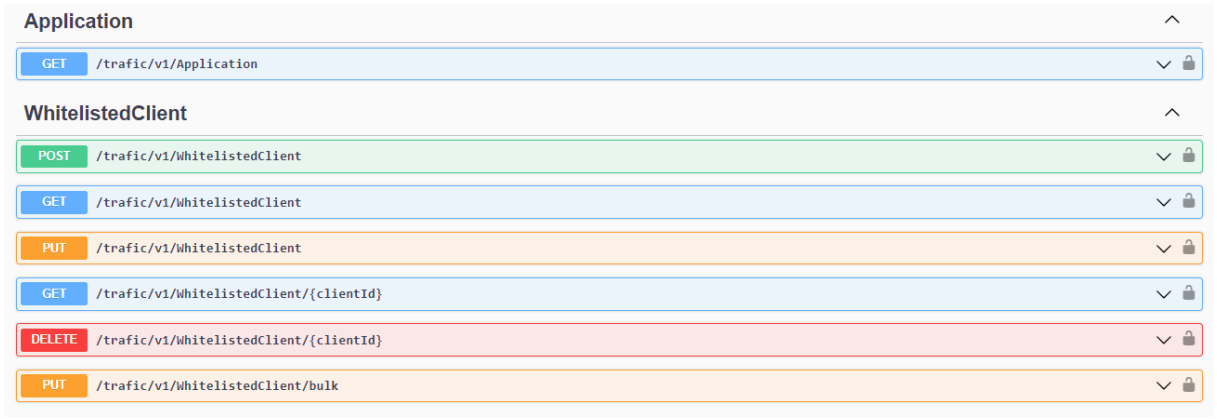


Figure 5.4: Management Console Endpoints

The ApplicationController has a single endpoint, "application", which retrieves all applications and their associated rights from the database. This data is then stored in cache by the Blazor app for fast access.

The WhitelistedClientContoller includes six endpoints:

- The POST: Allows the creation of a new client by submitting client details in the request body, including the client ID, token expiration time (in minutes), rights, and URIs.
- The first GET: Retrieves a list of all whitelisted clients.
- The first PUT: Similar to the first POST method, but it updates an existing client instead of creating a new one.
- The second GET: Retrieves the details of a specific client based on the provided ID as a URL parameter.
- The DELETE: Removes a client from the whitelist based on the provided ID as a URL parameter.
- The last PUT: Allows the update of multiple clients in a single call by passing a list of JSON objects containing the details for each client in the request body.

These endpoints provide the necessary functionality for managing the whitelist and ensuring that clients configurations can be easily created, updated, and maintained. The update and delete endpoints are secured by the rights CanEditAuthApiManagement and CanDeleteAuthApiClients respectively. By leveraging these endpoints, users of the Blazor app can seamlessly interact with the whitelist, supporting the dynamic management of clients.

5.4.2 Features

Adding a new client to the whitelist is a straightforward process. By clicking the "Add Client" button, shown in Figure 5.3, a dialog window appears, as shown in Figure 5.5. The dialog prompts users to input the client ID, JWT expiration time (in minutes), the rights that will be loaded into the JWT, and URIs where the one-time code will be received.

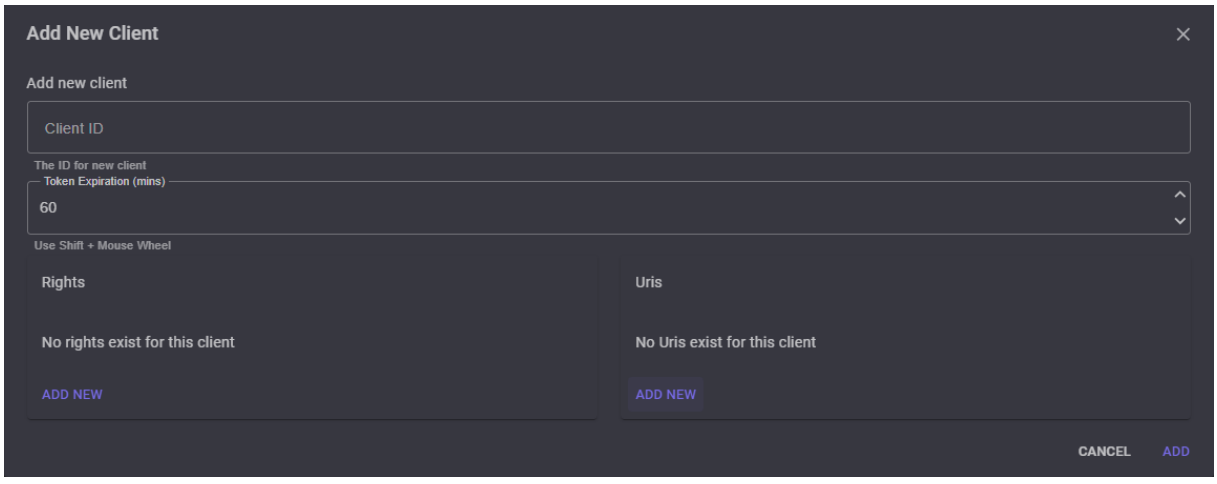


Figure 5.5: Management Console Add New Client

When clicking the "Add New" button in the rights section, the dialog shown in Figure 5.6 appears. This dialog contains two drop-down lists: the first displays the available Applications (also known internally as ATSA Modules), and once an application is selected, the second list shows the associated rights. Clicking "Refresh" will update the cache and

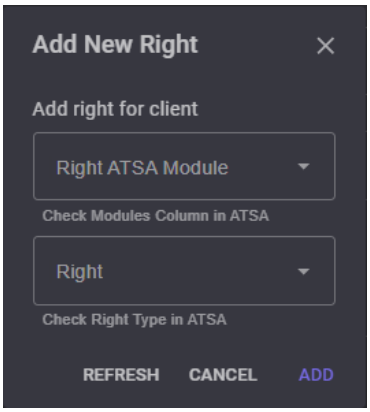


Figure 5.6: Management Console Add New Right

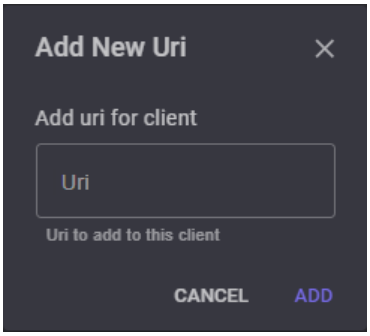


Figure 5.7: Management Console Add URI

retrieve the rights from database. Clicking the button "Add" will add the selected right to the client. This dialog is used for both creating new clients and updating existing ones.

For adding URIs to a client, a similar dialog is shown, as illustrated in Figure 5.7. Users must input the URI in a text field. If the URI matches the predefined regex, it is added to the client configuration when "Add" is clicked. This dialog is used for both creating and updating clients.

The application supports both dark and light themes. The theme of the app is set by default to match the system theme, but it can be changed by clicking the first button on the top right corner of the navigation bar. An example of the dark theme is shown in Figure 5.3, and a light theme example in Figure 5.8. The navigation bar also includes button to refresh the authentication. This feature is helpful when a new right has been just granted, and the user does not want to wait for the JWT to refresh automatically.

5.4.3 Frameworks used

To create this web application, the main technologies and frameworks used were Blazor[4], MudBlazor[5] and .NET Core[6].

Blazor is a web framework that allows developers to build web applications using C# and .NET instead of JavaScript. It leverages Web Assembly, enabling C# client-side

[4]Blazor web site – https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor
[5]MudBlazor web site – https://mudblazor.com/
[6].NET web site – https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet

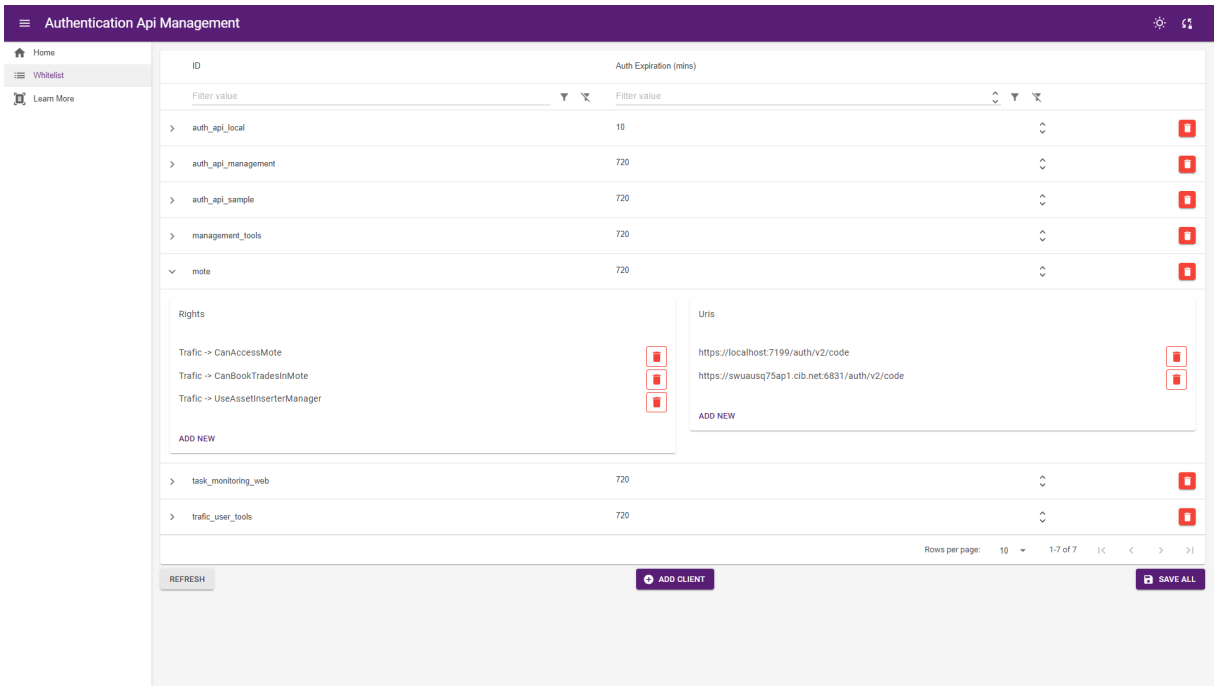


Figure 5.8: Management Console Light Theme

web applications to run directly in the browser. Blazor also supports server-side hosting, providing developers with the flexibility to choose the hosting model that best suits their needs. Like other modern frameworks, Blazor allows the creation of reusable components, allowing their usage in multiple pages or other applications.

MudBlazor is an open-source component library that provides pre-built UI components such as tables, buttons, text inputs, icons, navigation bars, and more. It accelerates web development by offering a wide range of standard components that are already built and highly customizable, allowing developers to quickly create user interfaces.

.NET Core is a cross platform, high-performance framework that helps building modern web applications and their respective REST APIs It uses C# as the programming language, which enables sharing of Data Transfer Objects between Blazor and the API project. This integration simplifies the development process and ensures consistency between the frontend and backend. The use of .NET Core allows for creation of efficient, maintainable, and high-performance web applications.

5.5 JWT Creation

The final authentication step of AuthenticationApi is the generation of a JWT. This token enables users to make authenticated requests to multiple services that have a trust relationship with AuthenticationApi.

The JWT is generated using an asymmetric signature algorithm called RSA/SHA-256 [22]. This cryptographic algorithm leverages a private key to sign the token and a corresponding public key to validate it. AuthenticationApi holds the private key, which it uses to sign the JWT. The public key is distributed to all services consuming the token, enabling them to validate its authenticity. The private key is stored in an encrypted XL Deploy

```
{

"http://schemas.microsoft.com/ws/2008/06/identity/claim
s/windowsaccountname": "userlogin",

"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/
givenname": "John",

"http://schemas.xmlsoap.org/ws/2005/05/identity/claims/
surname": "Doe",

"http://schemas.microsoft.com/ws/2008/06/identity/claim
s/role": [
    "CanUseFeatureA",
    "CanUseFeatureB"
  ],

"http://schemas.microsoft.com/ws/2008/06/identity/claim
s/groupsid": "GroupA",
  "exp": 1722735874,
  "iss": "AuthenticationApi"
}
```


Figure 5.9: JWT Payload example

dictionary.

During the token generation process, claims, which are key-pair values, will be loaded into the payload of the JWT. The token includes the following claims: first name, surname, user login, email address, user group, user roles, issuer, and expiration time. The retrieval of user roles is dependent on the configuration of the whitelisted client application. A role will only be included if it is specified in the AuthApiWhitelistRightsToLoad table, shown in Figure 5.2, and if the user has that role. The user group is always included in token payload. Similarly to the roles, the expiration time of the token is set in the client application configuration, which can range from 1 to 60 minutes since token generation in production environment and up until 720 in test environment. An example of the JWT payload is illustrated in Figure 23.

5.6 Logging

To ensure efficient logging in AuthenticationApi, Elasticsearch and Kibana were utilised for log storage analysis and visualisation. Elasticsearch [29] is a distributed search and analytics engine designed for handling large volumes of data making it ideal for structured log storage. Kibana [45] is an open-source data visualisation and exploration tool that works in conjunction with Elasticsearch. It provides a user-friendly interface to visualise the data stored in Elasticsearch, allowing users to create interactive dashboards where it is possible to monitor key metrics, identify trends, and gain insights through custom visualisations. Additionally, users can search and filter application logs. These technologies were chosen for their scalability, speed, and ability to handle large volumes of log data in near real time.

With these tools in mind, it is important to log events, in a structured form, which helps understanding the behaviour of the system and analyse it. The structured logs were added in the main calls done to AuthenticationApi, specifically on the login, callback, token, and windows token endpoints, covering all login flows.

The events captured by these logs are when a non-whitelisted client request is detected or when a request is successfully completed. Only one of these two logs will be logged depending on the client whitelist validation. The fields used for the structured logs, shown in Figure 5.10, excluding the message and timestamp fields, are the following:

- Endpoint: Indicates which endpoint was called, the possible values are "[Login]", "[Callback]", "[Token]" and "[WindowsToken]".
- Client ID: Represents the unique ID of the client that made the request, identifying the specific app.
- Whitelisted: A Boolean value that denotes whether the provided client ID and URI are part of the whitelist. URI verification only occurs in the login endpoint.
- Elapsed milliseconds: The time, in milliseconds, taken since receiving the REST request to sending the response (applicable to all endpoints).
- Response Type: Defines the method used to request a JWT with the possible values being authorisation_code, refresh_token, windows_token.
- Redirect URI: Represents the URI provided in the login and callback endpoint requests.
- User: Represents the authenticated user's login; this field is not available in the login endpoint.
- User group: Represents the group the user belongs to and is only available in token and windows token endpoint.
- Expiration in Minutes: Represents the expiration time of the issued JWT in minutes. It is only available in token and windows token endpoints.

Time	endpoint	client_id	whitelisted	elapsedms	response_type	redirect_uri	user	user_group	expiration_mins
Aug 12, 2024 @ 19:20:01.217	[WindowsToken]	mote	true	375.345	windows_token	-	fernandesjo	MOE Trafic	720
Aug 12, 2024 @ 19:19:49.594	[Token]	auth_api_local	true	1,198.754	refresh_token	-	fernandesjo	MOE Trafic	10
Aug 12, 2024 @ 19:19:41.676	[Token]	auth_api_local	true	1,852.261	authorization_code	-	fernandesjo	MOE Trafic	10
Aug 12, 2024 @ 19:19:33.416	[Callback]	auth_api_local	true	1,869.269	authorization_code	https://localhost:44390	fernandesjo	-	-
Aug 12, 2024 @ 19:19:26.455	[Login]	auth_api_local	true	2,919.117	authorization_code	https://localhost:44390	-	-	-

Figure 5.10: Example Logs

To leverage the structured logs and have some metrics related to the authentication a dashboard was created, as seen in Figure 5.11. The dashboard contains the following elements:

- Human Logins by App: A pie chart of all web applications client ids that use AuthenticationApi to authenticate human users.
- Human Logins by User: Represents the number of generated tokens for each web application user.

- Human Logins: A list correlating users to the number of tokens generated on each client web application.
- Windows Login by App: A pie chart representing the client IDs of processes requesting tokens via the Windows Login Flow.
- Windows Login by User: Represents the number of generated tokens for each user that requested it via Windows Login Flow, specifically for non-human users.
- Windows Logins: A list correlating users to the number of tokens generated using Windows Login Flow.
- Average of Elapsed Milliseconds: Five cards representing the amount of time in milliseconds that the 4 endpoints take to complete. The token endpoint contains two cards, one for a token generated via the SSO Login Flow and another for the tokens generated via the Refresh Token Login Flow.
- Non-Whitelisted Requests: Represents the number of requests done to AuthenticationApi providing a client ID that is not part of the whitelist.
- Tokens Generated by User Group: Represents the percentage of tokens generated grouped by user group, including windows and human logins.
- Non-Whitelisted Requests list: Represents all the non-whitelisted client ids that made a request to AuthenticationApi.

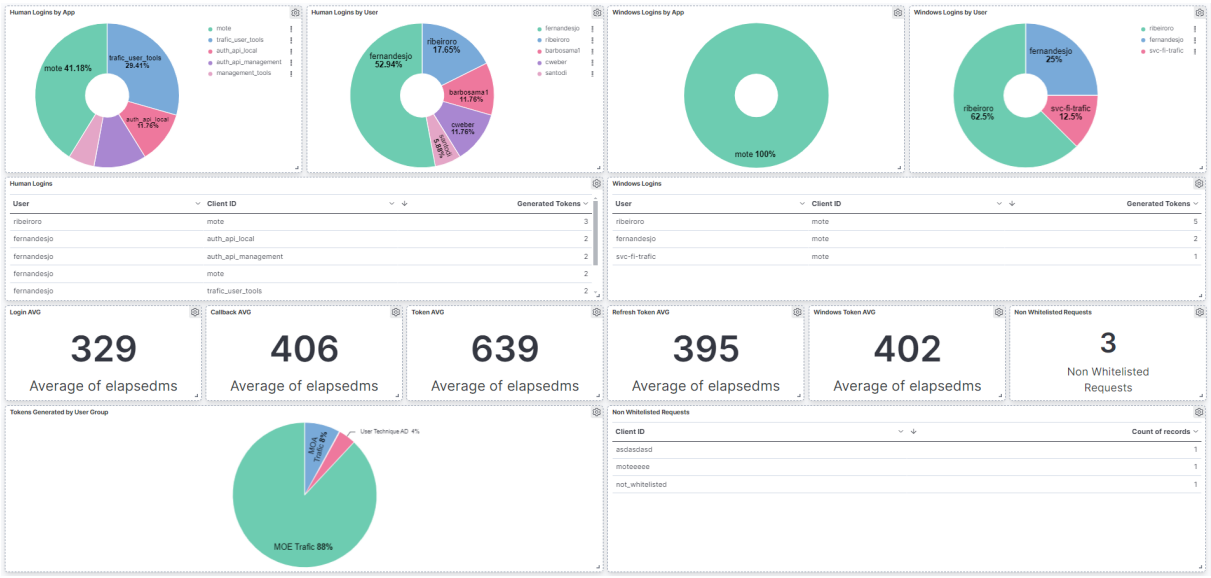


Figure 5.11: Kibana Dashboard

5.7 DevOps Environment

To enable the development of this project, several tools were used to facilitate the process which are represented in Figure 5.12. Bitbucket was used to store the produced code, leveraging Git as a version control system allowing for efficient version tracking. JIRA was utilised to define the requirements and functionalities of the API. Jenkins enabled continuous integration and continuous delivery by automating the processes of building,

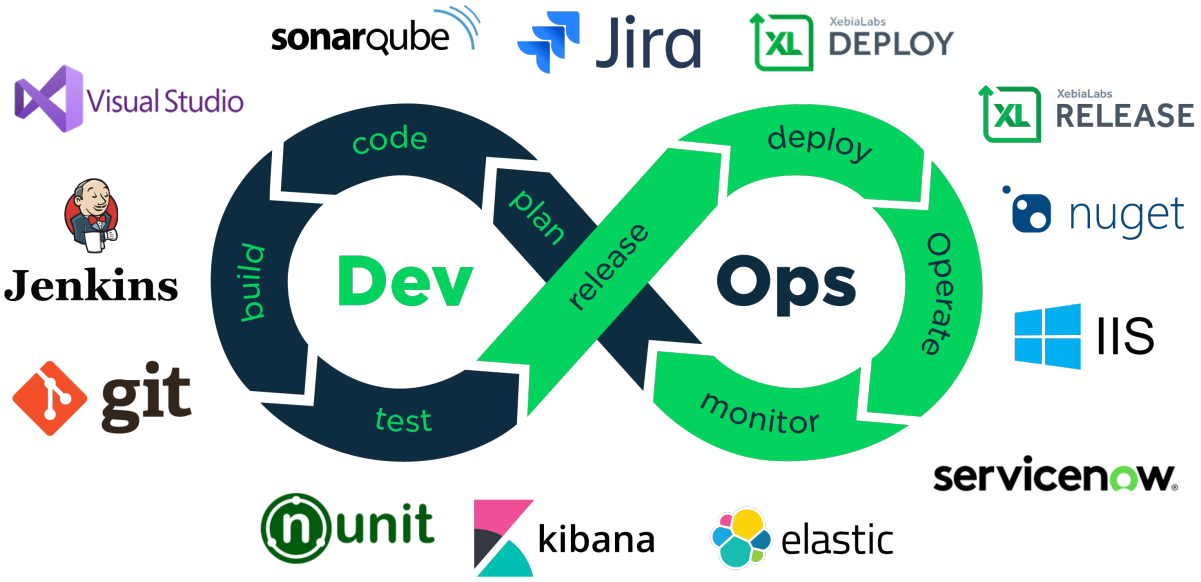


Figure 5.12: DevOps Environment

testing, code analysis with SonarQube, creating NuGet packages for the created libraries, creating XL Deploy packages, and deploying them automatically to test environments. XL Deploy served to easily retain application versions and deploying them into test and production environments. The releases of new versions were managed using XL Release, which allowed not only for the deployment but also for other tasks as creating change requests in ServiceNow and sending automatic email communications about the new release. The applications, after released, run in Internet Information Services which can host web applications and services such as AuthenticationApi. Kibana is used to monitor the applications by searching for logs and creating dashboards with metrics.

5.8 Release process

The release process for AuthenticationApi is managed through an automated XL Release pipeline, as shown in Figure 5.13. This process integrates several tools, including XL Deploy, Jenkins, and a private ServiceNow instance (referred to internally as SUN). The release flow is split into four stages: deploying in the BENCH environment, deploying in Production, conducting post-release checks and communication.

Some steps in the process were intentionally excluded from Figure 26, for example in the communications stage, as they are management-driven and do not directly impact the technical aspects of the release and are not relevant to this project. The purple-highlighted steps in the image are related to change request tickets created automatically in ServiceNow. These tickets require approval from management before the release can proceed. Grey steps in the flow represent script tasks, which include retrieving the current time and expected release time, information necessary for creating the ServiceNow change request. Red steps denote manual actions. Yellow steps represent automated email notifications sent to the team, which are integrated with Microsoft Teams to alert everyone about the release status.

The first stage in the release process involves deploying the new version in the BENCH

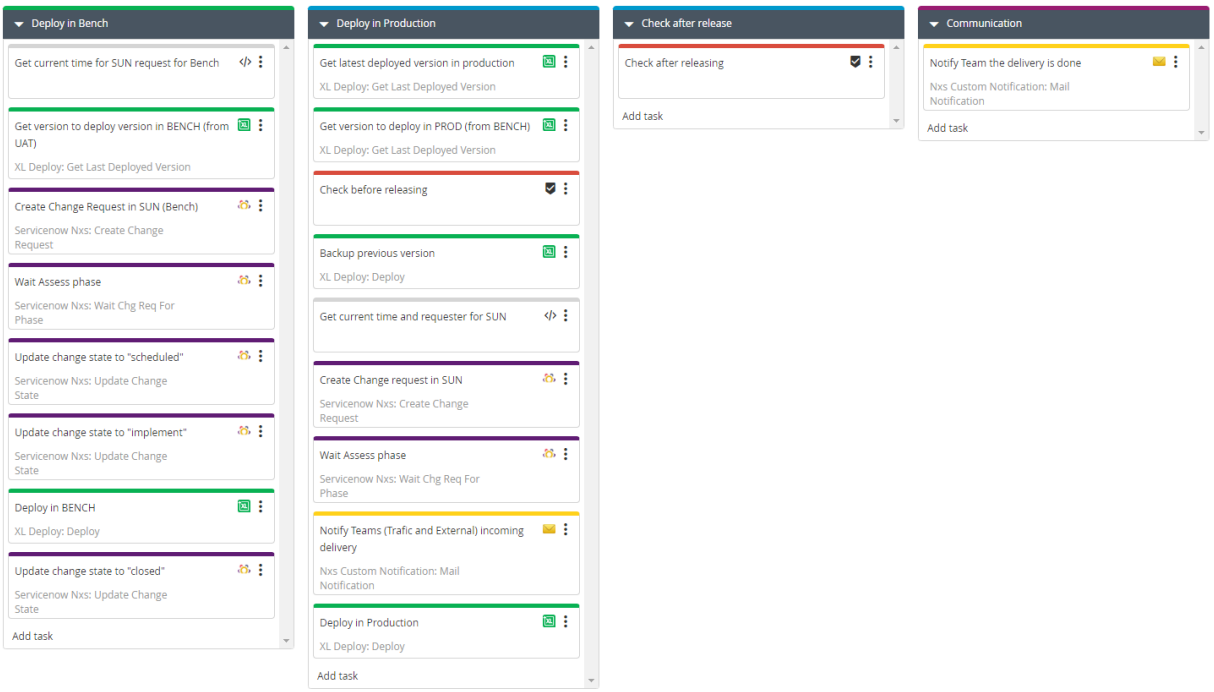


Figure 5.13: XL Release template

environment, which stands for Benchmark. During this stage, the version deployed in the UAT environment will be promoted to BENCH. This version is automatically deployed in UAT, by a Jenkins agent, whenever a build is created from the repository master branch. Change request tickets are automatically generated and processed in ServiceNow during this step. Although there is no formal step to validate if everything is working correctly in BENCH environment, as everything was already tested before in UAT environment, it is normally done.

In the Production deployment stage, the version to be deployed is retrieved from the BENCH environment. A manual check is conducted to verify that this is the intended version to release, ensuring that no unintended changes occurred in the BENCH environment in the meantime. Consequently, the current production version is backed up in an empty XL Deploy environment, and the release proceeds. A notification is sent to the team via Microsoft Teams to indicate that the release has begun. After the deployment is over, a manual check, shown in Figure 5.14, is conducted to verify that authentication in web applications is functioning as expected with the new AuthenticationApi version.

Finally, once the release is complete, a final notification is sent to Microsoft Teams, informing the team that the deployment has successfully finished.

5.9 Critical Analysis

The implementation of AuthenticationApi presents several strengths, but it also brings challenges that require consideration. One advantage is the flexibility introduced by multiple login flows, the SSO, Refresh Token, and Windows Login. While these multiple flows offer authentication options suitable for different use cases, they also add a layer of complexity. Each flow needs careful testing to ensure that no security vulnerabilities arise and that all flows ultimately produce similar tokens.

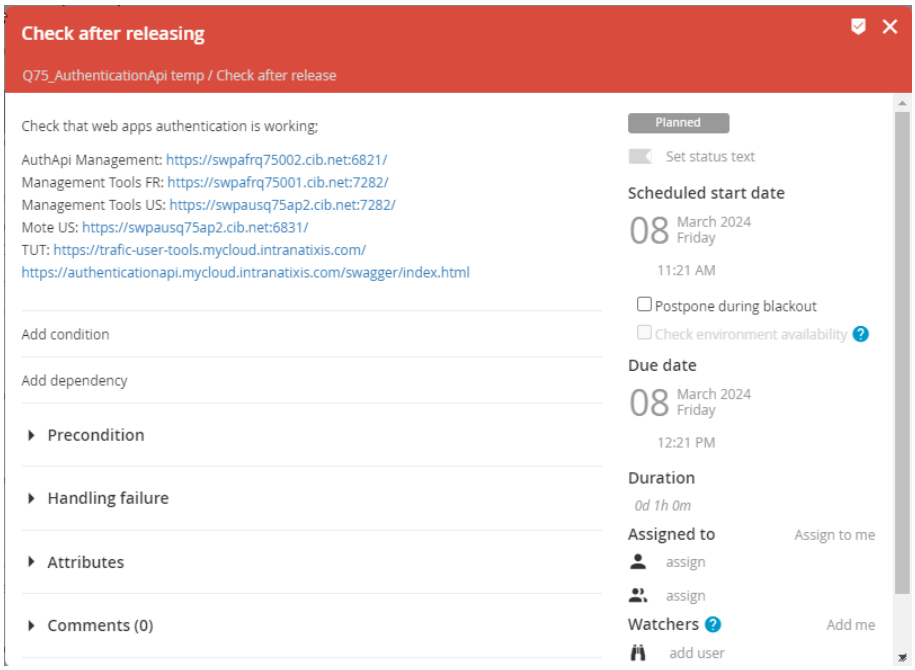


Figure 5.14: Check after release step

The management of cryptographic keys used for signing tokens also requires careful handling. Ensuring the proper storage and security of these keys is vital to preventing vulnerabilities such as unauthorised token signing or token forgery. A breach in key management could undermine the security of the entire authentication system.

The connection libraries developed for the API significantly ease the integration process for developers working with Razor Web Applications on the frontend and .NET Core web APIs on the backend. However, these libraries are currently limited to these specific frameworks. If future use cases demand support for other frontend or backend technologies, additional libraries would need to be developed, potentially increasing the system's complexity and maintenance burden.

The release process designed for the API is generally streamlined but still involves some manual intervention. These manual steps are crucial in ensuring that everything is functioning correctly after releasing a new version as unexpected problems can arise even after all the tests priorly done. The manual steps are simple to do, but these checks could be automated in the future.

Lastly, the management console provides an effective way to modify the whitelist of clients authorized to use the API, eliminating the need for manual database intervention or hard-coded configurations. This feature significantly improves the system's usability and operational efficiency, allowing administrators to quickly and easily manage client web applications access.

In summary, while the design and implementation of AuthenticationApi offer flexibility, ease of integration, and enhanced security, it also introduces complexities and dependencies that need to be managed carefully to ensure the system's long-term reliability, maintainability, and scalability.

Chapter 6

Results

This chapter presents the outcomes of AuthenticationApi project, focusing on the development timeline, testing and deployment across multiple regions. Each sub section provides a detailed look at the progress made since the initial implementation until its final deployment. The project's timeline captures the major milestones, while the testing and deployment demonstrate the API's reliability. These results highlight the project's successful delivery in a real-world production scenario.

6.1 Project Timeline and Deliverables

A Gantt chart describing the project timeline is illustrated in Figure 6.1. In this chart it is possible to see all the stages since the beginning of the project until today. Each stage has a colour, green means that during that stage, the first version of the API was being developed, the light blue relates to the second version of the API and the dark blue relates to the management console web app.

Task Name	Start	End	Duration
Initial SSO Capability	2023-05-19	2023-11-29	195 days
Access Libraries V1	2023-11-29	2024-01-16	49 days
Scalability Upgrade	2024-01-16	2024-03-04	49 days
Overall Improvements	2024-01-16	2024-04-26	102 days
SSO Capability V2	2024-03-03	2024-03-14	12 days
Access Libraries V2	2024-03-14	2024-03-24	11 days
Management Console	2024-03-18	2024-05-03	47 days
Windows Login	2024-05-02	2024-05-29	28 days
Refresh Tokens	2024-05-12	2024-05-31	20 days
Structured Logging	2024-07-29	2024-08-22	25 days
Deploy in New York	2024-08-25	2024-09-29	36 days

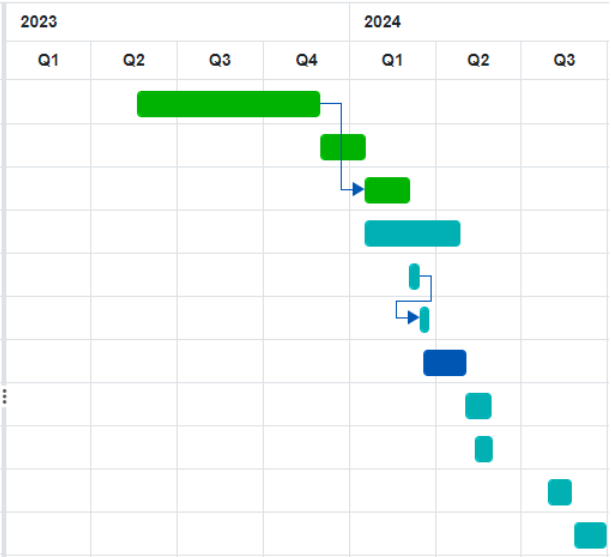


Figure 6.1: Project Timeline

The first stage, initial SSO capability, consisted of researching the concepts around SSO, SAML and authentication. Additionally simple SSO capability following the PKCE flow

was created. Although it did not comply with most of RFC 7636 [1], it served as a first version that would later be enhanced into the final product. This phase began in May 2023 and was concluded in November 2023. During this time, most of the research was conducted, concepts were learned, the target implementation was defined, and a simple working version was produced.

After having a stable first version, two internal NuGet libraries were developed to ease web applications access to AuthenticationApi, one for the frontend and another for the backend, this step took place between November 2023 until January 2024.

The next step involved enabling multiple instances of the API to run concurrently instead of just one. This work began in January 2024 and was completed in March 2024. This change introduced the use of a distributed SQL Server cache, allowing the authentication flow to start in one instance and finish in another.

Between January 2024 and April 2024, numerous enhancements were made, such as creating unit tests that cover more than 80% of the API code and integration tests to validate the login flow from start to finish. Additionally, hard-coded code was removed, including moving the client whitelist to data base tables instead of the program settings. The code-base was cleaned up, a sample client application was created, and the DevOps environment setup was finalised.

During March 2024, the API was enhanced to conform to RFC 7636, excluding section 4.4.1, related to specific error messages and section 5 related to compatibility with clients that do not use the PKCE extension. This required changes to endpoint parameters, secret generation, one-time code generation, the addition of an optional state parameter, and more. As a result, the NuGet libraries initially created also had to be updated to accommodate these changes.

During the months of March, April and May 2024, a web application was developed to manage the client whitelist, mentioned in the previous chapter. This application leverages AuthenticationApi for user authentication and enables the team to perform CRUD operations on the whitelist without having to manually do it in data base.

In May 2024, windows authentication was added to AuthenticationApi, enabling technical accounts to login. This feature allowed background processes and scripts to obtain a JWT and access services that require this token.

Also in May 2024, refresh tokens were introduced, significantly reducing the time required for users to request a new JWT once the previous one has expired, enabling the reduction of the JWT life from 60 minutes to 5 minutes. This improvement was made possible because the refresh token login is automatically handled by the access libraries and is seamless to the user, taking on average less than 1 second.

From July to August 2024, structured logging was introduced, allowing the visualisation of authentication metrics, and providing insights into user activity patterns. This enhancement enables future work, such as analysing data to detect suspicious login times or identifying potential authentication issues.

The final step was deploying the API in a second location outside of Europe, specifically in New York. By August 2025, the web application environment within the team had expanded, with more applications being developed. Some of these applications are based

in New York and require their own instances of AuthenticationApi to avoid high latency and long round trips to Europe.

The entire development process took 1 year and 4 months to complete. In the end, all initial objectives were met, delivered without any issues and the planned milestones were delivered on time. However, there is always room for improvement, and new features can be introduced any time. For example, logging more detailed user information, such as IP addresses, could enable more complex log analysis. Additionally, performance could be enhanced by caching the whitelist and refreshing it automatically by a request from the management console whenever changes are made. Finally, adding monitoring to the API would allow real-time tracking of the status of each instance.

6.2 Code Testing

The testing strategy for AuthenticationApi was designed to be comprehensive, given that the API is a critical piece of infrastructure. The goal is to achieve 90% code coverage for the API itself and 80% across all related projects combined. This extensive coverage ensures that the core functionalities are thoroughly tested, reducing the risk of undetected bugs in the system.

Integration tests play a crucial role in this strategy. They test the "happy path" of the authentication process by flowing through all SSO and refresh login endpoints. Additionally, non-successful cases were tested, such as when a provided client ID is not whitelisted or when required parameters are missing. These tests ensure that both valid and invalid scenarios are handled correctly by the API.

For unit testing, the goal is to cover as many branches as possible, asserting not only that the successful paths return the expected results, but also that the unsuccessful paths raise meaningful exceptions with the correct messages. This helps with troubleshooting and improves the overall maintainability of the code-base. Aiming for 100% test coverage was not necessary, as there are configuration and helper classes that do not add significant value when tested. The focus remained on functionalities where testing delivers the most value.

A total of 378 unit and integration tests were created, testing the API and surrounding elements, as shown in Figure 6.2. The frameworks and libraries used for testing included NUnit[1] as the test framework, NSubstitute[2] for mocking interfaces, Fluent Assertions[3] to ease test writing, bUnit[4] for Blazor UI tests, and Report Generator[5] to generate coverage reports, such as the one in Figure 6.3. For the main project, the API itself, 168 tests were created, 15 of which are integration tests, and the remaining are unit tests. For the NuGet libraries, 94 tests were created: 83 are for the frontend library, 4 for the backend library and 7 for the windows library. The management console backend had a total of 52 tests, while the frontend had 64.

With these 378 tests, an overall line coverage of 87.4% was achieved across all the .NET

[1]NUnit web site – https://nunit.org/
[2]NSubstitute web site – https://nsubstitute.github.io/
[3]Fluent Assertions web site – https://fluentassertions.com/
[4]BUnit web site – https://bunit.dev/
[5]Report Generator web site – https://reportgenerator.io/

Test	Duration
AuthenticationApi.Auth.Shared.Test (5)	19 ms
AuthenticationApi.DataAccess.Test (10)	203 ms
AuthenticationApi.Management.Client.Tests (64)	898 ms
AuthenticationApi.Management.Test (52)	97 ms
AuthenticationApi.Sso.Backend.Tests (4)	238 ms
AuthenticationApi.Sso.Frontend.Tests (83)	485 ms
AuthenticationApi.Test (153)	612 ms
AuthenticationApi.Windows.Backend.Tests (7)	137 ms

Figure 6.2: Tests run results

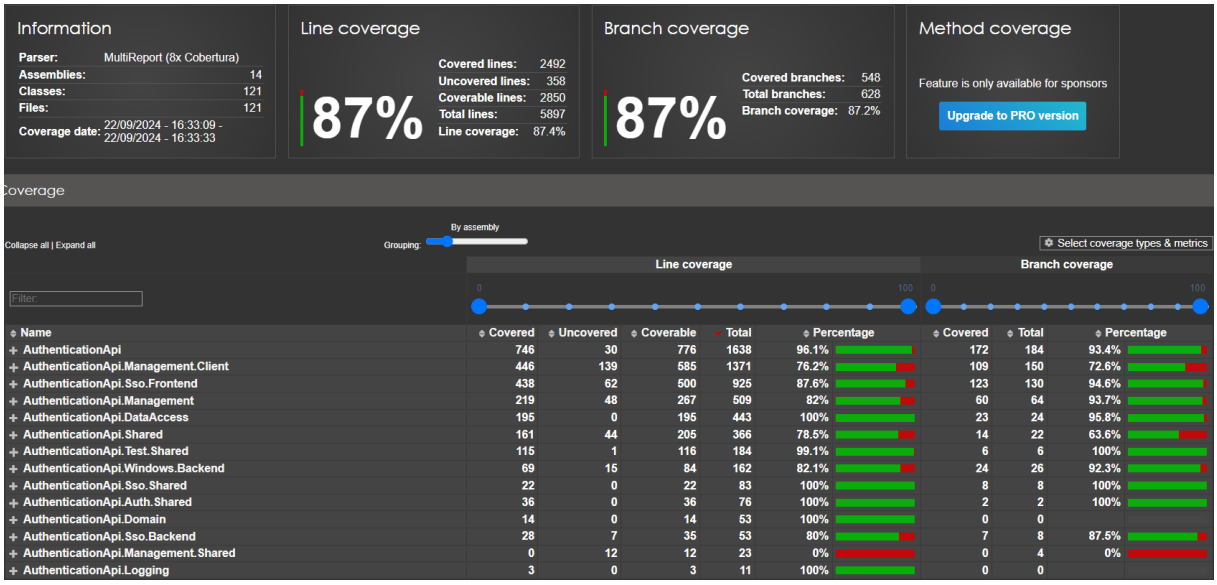


Figure 6.3: Test Coverage Report

Core projects. The API, being a critical piece of infrastructure, has a coverage of 96.1% as nearly all the methods and classes needed to be tested to ensure proper functionality. Some lines were intentionally excluded from testing, primarily those related to application startup, such as dependency injection setups, or those deemed irrelevant for testing, such as classes containing only configuration properties.

The integration tests cover the three main SSO login endpoints in the API: login, callback, and token. These tests were possible by mocking all external dependencies of the API. A custom web application factory was developed, utilising an in-memory database, a simulated SAML connection to GAAP, an in-memory distributed cache, and disabling all the logging. A custom HTTP client was created to interact with the API during integration tests. The endpoints were tested individually to ensure that they return the appropriate HTTP responses based on the requests made. Additionally, a test was created that covers the entire authentication flow from the login endpoint until the token request, both using the one-time code and refresh token.

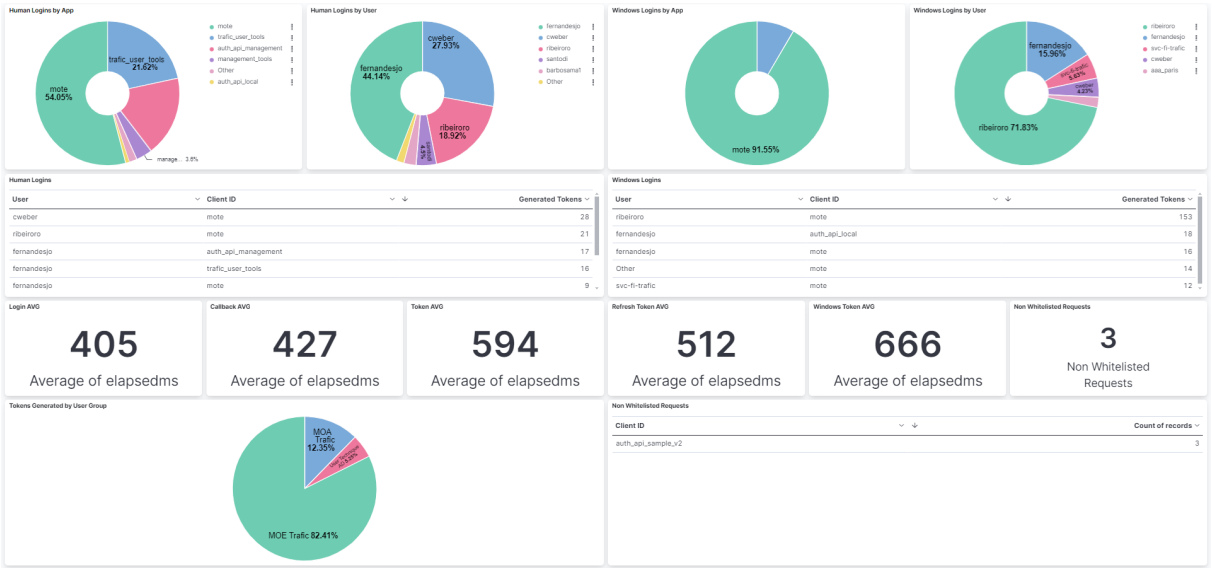


Figure 6.4: 21 days test environment metrics

6.3 API Deployment

As of today, the API has demonstrated reliable performance across multiple regions, with two instances running in Europe and two in the United States, as shown in Figure 4.5. The current code-base allows for easy expansion of the API to additional regions with minimal effort, only needing to configure the environment, GAAP connection, SSL certificate and load balancer.

In terms of performance, the average time taken to execute each endpoint of the API ranges between 400 and 600 milliseconds, as shown in Figure 6.4. These metrics were obtained from a test environment used daily by the team's developers to build their web applications. This environment consists of a single instance running on a Europe machine with 16 GB of RAM and an Intel Xeon Gold 5120 CPU. The total time consumed by AuthenticationApi for a complete login process is approximately 1.5 seconds. Additionally, there is some additional time taken by GAAP and the web application itself during loading, which was not measured.

Scaling the application up or down is straightforward. More instances of the API can be brought online by simply adding a new machine in the XL Deploy environment and registering its IP addresses with the load balancer. Although this process is manual, thus prevents automatic scaling, it is not a priority for the project, as the API load remains relatively consistent, given the known and limited user base. However, the API was designed with future enhancements in mind, since it is built on .NET Core 8, it can be containerised and deployed using modern technologies like Kubernetes, enabling more advanced scaling features. Additionally, it is easy to replace the current SQL distributed cache with another technology by modifying the startup file, as all classes use the IDistributedCache[6] abstraction.

Since the initial release on November 2023, even with minimal features, the API has operated without any user complaints, maintaining stability through all subsequent updates and improvements. Over time, the number of web applications within the team has

[6]Distributed cache – https://learn.microsoft.com/aspnet/core/performance/caching/distributed

grown from one to five, with three in Europe and two in the United States. Additionally, two micro-services have been developed that accept AuthenticationApi JWTs as an authentication method.

6.4 Metrics & Statistics

The metrics presented in this section are derived from the structured logs captured from the production environment between August 23 and September 25 of 2024. These metrics offer insight into the API's performance, login patterns, and usage. Comparisons are also made with the UAT test environment to assess differences in performance.

Before analysing the data, several pre-processing steps were taken to clean and anonymise the raw CSV data extracted from Kibana using Python, Jupyter notebooks, and the pandas, matplotlib, and seaborn libraries. Initially, unnecessary columns, such as "_id", "_index", "_score", and "_type", which did not contain relevant information, were removed. Other redundant columns related to process ID, thread ID, and environment name were also deleted, as well as duplicate columns that appeared with a ".keyword" suffix (for example client_id and client_id.keyword). After this, all columns were renamed for clarity, for example, "@timestamp" was renamed to "Timestamp." Missing values, represented as the character '-', were cleaned up by replacing them with an empty value. Data types were explicitly defined, converting the timestamp column to a date format and numeric fields such as "Elapsed Milliseconds" and "Expiration Minutes" to integers. The logs were also ordered chronologically by timestamp. Finally, for data privacy, user-specific and URI-related columns were anonymised; user logins, Client IDs, Redirect URIs, User Groups, and Host fields were replaced with labels like "user 1," "client 1," and so on. Additionally, missing User Group data was filled based on the corresponding user information. All Python scripts used for this process are available in Appendix A.

Across the four API endpoints, the average response times (in milliseconds) indicate that the API is performing well, even under regular usage in production. Three outliers were removed from the data due to an unusual elapsed time of over 4 seconds. From the metrics displayed in Figure 6.5 and Table 6.1 we can observe the following key points:

- The Login endpoint has an average response time of approximately 1.08 seconds, with most of the calls ranging between 91 and 1536 milliseconds.
- The Callback endpoint has an average time of 590 milliseconds, with most calls ranging from 341 to 840 milliseconds.
- The Token endpoint, responsible for issuing JWT tokens, averages 1.06 seconds, with most calls ranging from 326 to 1658 milliseconds.
- The Windows Token endpoint, used for NTLM login by technical processes, averages a higher time of 1.9 seconds, possibly due to the time take by the NTLM authentication process. Most of the calls range between 1.8 and 2.2 seconds.

The box plot in Figure 6.5 and Table 6.1 help visualising these metrics, where we can see the variance, distribution, averages, quantiles, minimal and maximum values for each endpoint's response time.

Users from different groups tend to log in at various times throughout the day, as shown in Figure 6.6. These login hours range from 9 AM to 10 PM (Portugal Time), which

Endpoint	Count	mean	std	min	25%	50%	75%	max
Login	157	1 080	755	4	91	1 289	1 536	3 206
Callback	77	590	335	84	341	544	840	1 288
Token	172	1 059	978	200	326	654	1 658	3 534
WindowsToken	5	1 901	820	641	1 805	1 945	2 216	2 897

Table 6.1: Elapsed Milliseconds metrics by Endpoint

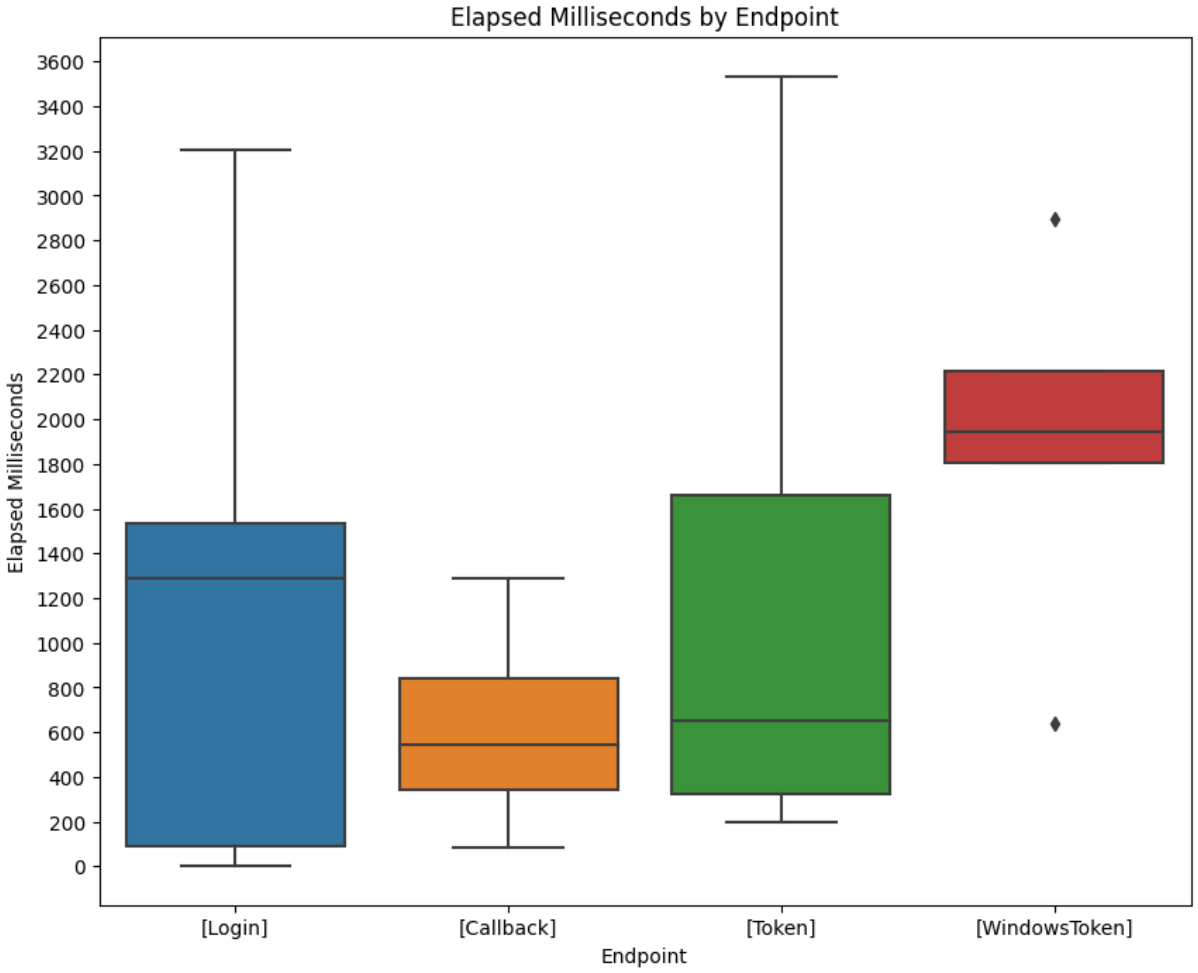


Figure 6.5: Elapsed Milliseconds by Endpoint

aligns with the expected usage patterns for European and New York-based users. Given that, at the time logs were taken, the API was yet to be deployed in New York, this distribution reflects activity from Europe and New York, during regular business hours, both using the European instance of the API.

In the UAT test environment, where fewer services are running concurrently and there is less overall demand through the day, the average response times are better as shown in Table 6.2.

- Login: 326 milliseconds
- Callback: 391 milliseconds
- Token: 606 milliseconds

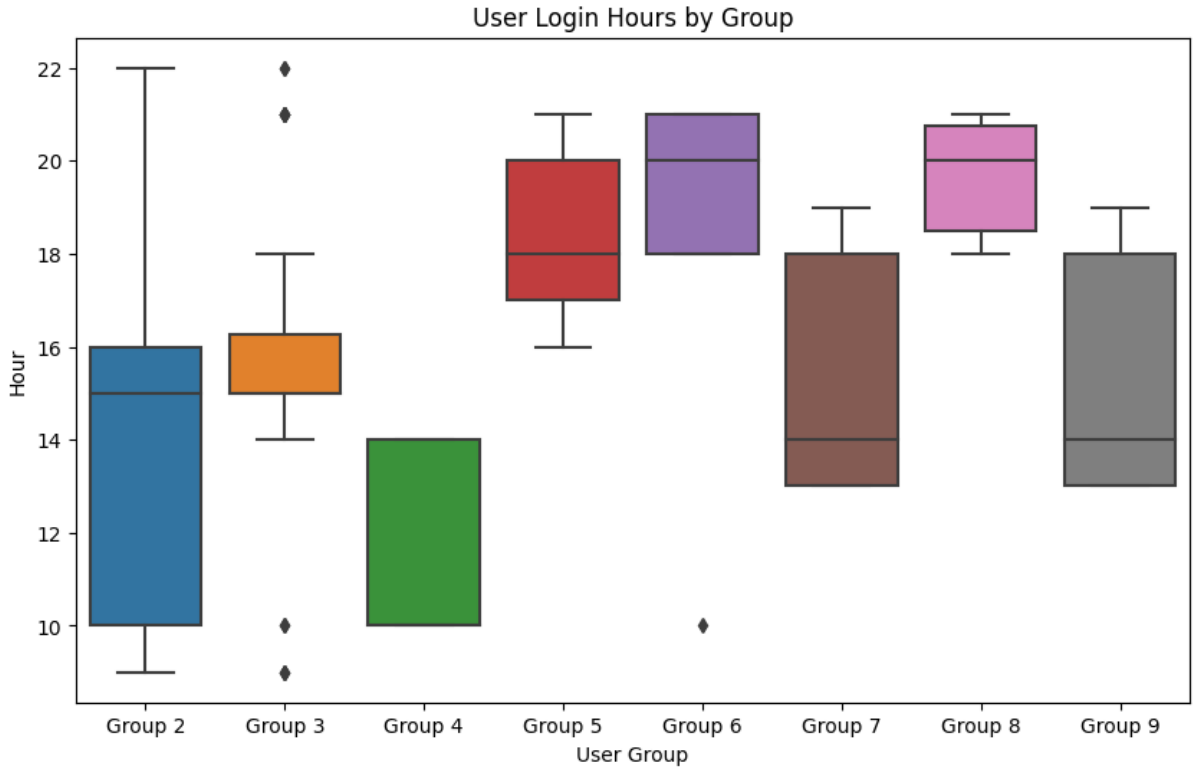


Figure 6.6: User Login Hours by Group

Endpoint	count	mean	std	min	25%	50%	75%	max
Login	110	326	755	4	8	61	106	3 402
Callback	102	391	366	80	142	345	450	2 527
Token	143	606	426	175	320	517	736	2 773
WindowsToken	130	401	515	66	128	186	405	3 563

Table 6.2: Elapsed Milliseconds metrics by Endpoint UAT environment

- Windows Token: 401 milliseconds

Although, there are some cases of long-lasting calls as some debugging could be taking place when the logs were generated, for this reason, logs with an elapsed time over 4 seconds were removed from the data before generating Table 6.2.

These differences likely originate from the UAT machine being dedicated solely to the API, resulting in fewer concurrent tasks. The production environment, however, hosts multiple services, which leads to slightly slower response times. Nevertheless, the API performance remains within acceptable ranges, especially when refresh token logins are used.

Figure 6.7 presents the breakdown of response types used by each client.

- Client 1 is not utilising refresh tokens, indicating that the app has yet to adopt the latest version of the Authentication API libraries.
- Client 2 and Client 3 have lower refresh token usage compared to authorisation tokens. This suggests that these applications may not be heavily used, with users returning more than 7 days after their last session, causing their refresh tokens to

expire.

- Client 5, the most active application, makes extensive use of refresh tokens, improving the overall user experience by minimising the need for SSO login flow, which is the longest one.
- Client 4 also exhibits a higher usage of refresh tokens than the full SSO login flow, despite lower frequency logins. This implies that a very small group of users is most likely using the app multiple times within a 7-day period.
- Client 3, utilising all three response types, which means that both a web application and a related technical process are using the same Client ID and configuration as they need the same rights loaded into the JWT.

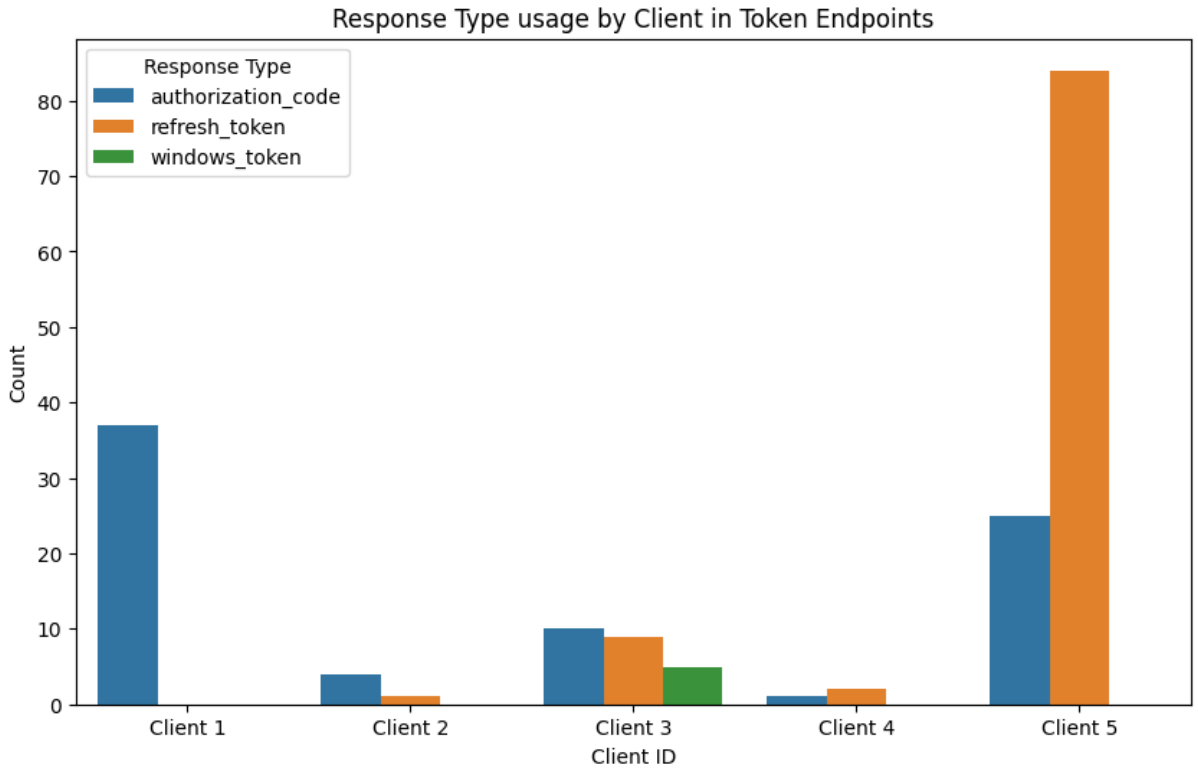


Figure 6.7: Response Type usage by Client in Token Endpoints

6.5 Critical Analysis

The project was developed and delivered within the expected time frame, as outlined in Section 6.1, with all deliverables being achieved on time. The performance metrics detailed in section 6.4 further validate the API's robustness, showing that the response times for the endpoints fall within acceptable ranges, even under the production load resulting from multiple services running concurrently on the same infrastructure. The usage analysis reveals areas for improvement, such as ensuring that clients adopt refresh tokens more consistently to optimise user experience. As seen, Client 5 reliance on refresh tokens results in a smoother user experience by reducing the need for full SSO logins. Encouraging other clients, such as Client 1, to update to the latest Authentication API libraries would further enhance overall performance and user satisfaction.

The collected data could be leveraged as the first step towards developing a real-time anomaly detection system, which would allow for proactive identification of unusual behaviours and potential security threats. The system could flag anomalies, such as unexpected spikes in response times or unusual login attempts, facilitating faster detection and resolution of potential issues.

Overall, a stable and well-functioning version of the API was created and delivered across 2 regions of the globe and has been working without problems since it's deployment. The API's performance, averaging 2.5 seconds per login process, has remained consistent and reliable, handling daily usage without user complaints. The extensive tests created, resulting in 96.1% coverage of the API core code, can prevent potential future bugs when changes will inevitably need to be made.

Chapter 7

Conclusion and Future Work

The primary objective of this project was to develop a secure, reliable, and scalable authentication system using SSO with support for both human and technical users. All the proposed objectives were successfully accomplished within the planned timeline. The AuthenticationApi now provides seamless SSO login support, complemented by refresh tokens for faster re-authentication, and Windows login for non-human users, such as automated processes. These key features solve the corporation's problems by ensuring that all types of users, whether human or not, can securely authenticate and interact with the system. The implementation of RFC 7636, known as PKCE, has standardised the SSO authentication process, adding an additional layer of security to protect against authorisation code interception attacks.

Furthermore, the management console web application has played a crucial role in simplifying the client management process by enabling the team to efficiently add new clients and update existing ones. Currently, due to the libraries created, five web applications are leveraging AuthenticationApi to authenticate users, and two of the team's thirty .NET Core microservices already support Bearer Authentication which is provided by AuthenticationApi.

Integration and unit tests were developed, covering 96.1% of the API's code-base. These tests ensure that the API maintains its reliability, reducing the risk of bugs or failures as updates are made. This high level of coverage assures that all core functionalities are thoroughly tested, contributing to the stability of the system.

The API has been successfully deployed across two locations, New York and Paris, serving users from both regions with minimal possible latency. The distributed nature of the deployment allows for scalability, high availability, and improved performance across geographical locations, providing users with an efficient and secure authentication experience.

Looking forward, there are several areas for future enhancement. One is to log additional properties, such as the user IP address, to detect potential anomalies, risks and enhance security monitoring. Stress testing the API will also provide valuable insights into how it handles large volumes of requests and ensure that the system behaves effectively under heavy load. Furthermore, integrating artificial intelligence into log analysis could enable smart cybersecurity alerts by detecting strange or abnormal login behaviours in real time. Additionally, containerising the API and deploying it using container orches-

tration technologies such as Kubernetes would provide significant benefits. Kubernetes would enable automatic scaling, ensuring that the system can seamlessly handle variable loads by adding or removing instances of the API based on demand. Finally, completely automating the release process, including checks to ensure users are authenticated correctly when opening web applications, will further streamline the system's deployment and maintenance processes.

Chapter 8

Bibliography

[1] J. Bradley and N. Agarwal. Proof key for code exchange by oauth public clients. RFC7636 `https://datatracker.ietf.org/doc/html/rfc7636`, 9 2015. Accessed: 2024-10-14.

[2] Ahmet Bucko, Kamer Vishi, Bujar Krasniqi, and Blerim Rexha. Enhancing jwt authentication and authorization in web applications based on user behavior history. *Computers*, 12:78, 4 2023.

[3] B. Campbell, C. Mortimore, and M. Jones. Security assertion markup language (saml) 2.0 profile for oauth 2.0 client authentication and authorization grants, 5 2015.

[4] Scott Cantor, John Kemp, Rob Philpott, and Eve Maler. Assertions and protocols for the oasis security assertion markup language (saml) v2.0, 2005.

[5] Scott Cantor, Jahan Moreh, Rob Philpott, and Eve Maler. Metadata for the oasis security assertion markup language (saml) v2.0, 2005.

[6] Microsoft Corporation. Microsoft entra id. `https://www.microsoft.com/en-us/security/business/identity-access/microsoft-entra-id`. Accessed: 2024-10-07.

[7] Daniel Fett, Ralf Kusters, and Guido Schmitz. The web sso standard openid connect: In-depth formal security analysis and security guidelines. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 189–202. IEEE, 8 2017.

[8] Christos A. Fidas and Dimitrios Lyras. A review of eeg-based user authentication: Trends and future research directions. *IEEE Access*, 11:22917–22934, 2023.

[9] OpenID Foundation. Openid. `https://openid.net/`. Accessed: 2024-10-07.

[10] Somchart Fugkeaw. Achieving decentralized and dynamic sso-identity access management system for multi-application outsourced in cloud. *IEEE Access*, 11:25480–25491, 2023.

[11] Somchart Fugkeaw, Intanont Langsanam, and Hasatorn Saviphan. Design and implementation of fast and secure sso authentication for multi-application services deployed in cloud. In *15th International Conference on Knowledge and Smart Technology, KST 2023*. Institute of Electrical and Electronics Engineers Inc., 2023.

[12] A Shaji George. The dawn of passkeys: Evaluating a passwordless future. *Partners Universal Innovative Research Publication*, 2(1):202–220, 2024.

[13] Paul A Grassi, Justin P Richer, Sarah K Squire, James L Fenton, Ellen M Nadeau, Naomi B Lefkovitz, Jamie M Danker, Yee-Yin Choong, Kristen K Greene, and Mary F Theofanos. Digital identity guidelines: federation and assertions, 6 2017.

[14] Chengqian Guo, Fan Lang, Qiongxiao Wang, and Jingqiang Lin. Up-sso: Enhancing the user privacy of sso by integrating ppid and sgx. In *2021 International Conference on Advanced Computing and Endogenous Security*, pages 01–05. IEEE, 4 2022.

[15] Dick Hardt. The oauth 2.0 authorization framework. RFC6749 `https://www.rfc-editor.org/info/rfc6749`, 10 2012. Accessed: 2024-10-14.

[16] Frederick Hirsch, Rob Philpott, and Eve Maler. Security and privacy considerations for the oasis security assertion markup language (saml) v2.0, 2005.

[17] Nazmul Hossain, Md. Alam Hossain, Md. Zobayer Hossain, Md. Hasan Imam Sohag, and Shawon Rahman. Oauth-sso: A framework to secure the oauth-based sso service for packaged web applications. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 1575–1578. IEEE, 8 2018.

[18] John Hughes, Scott Cantor, Jeff Hodges, Frederick Hirsch, Prateek Mishra, Rob Philpott, and Eve Maler. Profiles for the oasis security assertion markup language (saml) v2.0, 2005.

[19] Auth0 Inc. Auth0. `https://auth0.com/`. Accessed: 2024-10-07.

[20] JBoss Inc. Keycloak. `https://www.keycloak.org/`. Accessed: 2024-10-07.

[21] Meiko Jensen, Christopher Meyer, Juraj Somorovsky, and Jorg Schwenk. On the effectiveness of xml schema validation for countering xml signature wrapping attacks. In *2011 1st International Workshop on Securing Services on the Cloud (IWSSC)*, pages 7–13. IEEE, 9 2011.

[22] M. Jones. Json web algorithms (jwa). RFC7518 `https://datatracker.ietf.org/doc/html/rfc7518`, 5 2015. Accessed: 2024-10-14.

[23] M. Jones, J. Bradley, and N. Sakimura. Json web signature (jws). RFC7515 `https://datatracker.ietf.org/doc/html/rfc7515`, 5 2015. Accessed: 2024-10-14.

[24] M. Jones, J. Bradley, and N. Sakimura. Json web token (jwt). RFC7519 `https://datatracker.ietf.org/doc/html/rfc7519`, 5 2015. Accessed: 2024-10-14.

[25] M. Jones and D. Hardt. The oauth 2.0 authorization framework: Bearer token usage. RFC6750 `https://datatracker.ietf.org/doc/html/rfc6750`, 10 2012. Accessed: 2024-10-14.

[26] M. Jones and J. Hildebrand. Json web encryption (jwe). RFC7516 `https://datatracker.ietf.org/doc/html/rfc7516`, 5 2015. Accessed: 2024-10-14.

[27] S. Josefsson. The base16, base32, and base64 data encodings. RFC4648 `https://datatracker.ietf.org/doc/html/rfc4648`, 10 2006. Accessed: 2024-10-14.

[28] Nickson M. Karie, Victor R. Kebande, Richard A. Ikuesan, Mehdi Sookhak, and H. S. Venter. Hardening saml by integrating sso and multi-factor authentication (mfa) in the cloud. In *Proceedings of the 3rd International Conference on Networking, Information Systems Security*, pages 1–6. ACM, 3 2020.

[29] Nikita Kathare, O Vinati Reddy, and Vishalakshi Prabhu. A comprehensive study of elastic search. *Journal of Research in Science and Engineering*, 4, 11 2022.

[30] Jonh Kemp, Scott Cantor, Prateek Mishra, Rob Philpott, and Eve Maler. Authentication context for the oasis security assertion markup language (saml) v2.0, 2005.

[31] Jonh Kemp, Scott Cantor, Prateek Mishra, Rob Philpott, and Eve Maler. Bindings for the oasis security assertion markup language (saml) v2, 2005.

[32] Wanpeng Li, Chris J. Mitchell, and Thomas Chen. Oauthguard: Protecting user security and privacy with oauth 2.0 and openid connect. In *Proceedings of the 5th ACM Workshop on Security Standardisation Research Workshop*, pages 35–44. ACM, 11 2019.

[33] Chi Wei Lien and Sudip Vhaduri. Challenges and opportunities of biometric user authentication in the age of iot: A survey. *ACM Computing Surveys*, 56, 8 2023.

[34] Google LLC. Identity platform. `https://cloud.google.com/security/products/identity-platform`. Accessed: 2024-10-07.

[35] Hal Lockhart, Brian Campbell, Nick Ragouzis, John Hughes, Rob Philpott, Eve Maler, Paul Madsen, and Tom Scavo. Security assertion markup language (saml) v2.0 technical overview, 2008.

[36] M. McGloin and P. Hunt. Oauth 2.0 threat model and security considerations. RFC6819 `https://datatracker.ietf.org/doc/html/rfc6819`, 1 2013. Accessed: 2024-10-14.

[37] Jeremy Nelson. *Mastering Redis*. Packt Publisher, 5 2016.

[38] Inc. Okta. Okta. `https://www.okta.com/`. Accessed: 2024-10-07.

[39] Inc. OneLogin. Onelogin. `https://www.onelogin.com/`. Accessed: 2024-10-07.

[40] Maria Papathanasaki, Leandros Maglaras, and Nick Ayres. Modern authentication methods: A comprehensive survey. *AI, Computer Science and Robotics Technology*, 2022:1–24, 6 2022.

[41] Viral Parmar, Harshal A. Sanghvi, Riki H Patel, and Abhijit S. Pandya. A comprehensive study on passwordless authentication. In *2022 International Conference on Sustainable Computing and Data Communication Systems (ICSCDS)*, pages 1266–1275. IEEE, 4 2022.

[42] Praveen Kumar Rayani and Suvamoy Changder. Continuous user authentication on smartphone via behavioral biometrics: a survey. *Multimedia Tools and Applications*, 82:1633–1667, 1 2023.

[43] Redis and Salvatore Sanfilippo. Redis - the real-time data platform, 2024.

[44] Nat Sakimura, John Bradley, Michael B. Jones, Breno de Medeiros, and Chuck Mortimore. Openid connect core 1.0, 2023.

[45] Neel Shah, Darryl Willick, and Vijay Mago. A framework for social media data analytics using elasticsearch and kibana. *Wireless Networks*, 28:1179–1187, 4 2022.

[46] Naim Shaikh, Kishori Kasat, and Smita Jadhav. Secured authentication by single sign on (sso): A big picture. In *2022 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, pages 951–955. IEEE, 11 2022.

[47] Jaimandeep Singh and Naveen Kumar Chaudhary. Oauth 2.0 : Architectural design augmentation for mitigation of common security vulnerabilities. *Journal of Information Security and Applications*, 65:103091, 3 2022.

[48] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems*. CreateSpace Independent Publishing Platform, 2017.

[49] WSO2. Wso2 identity server. `https://wso2.com/identity-server/`. Accessed: 2024-10-07.

[50] Zhiyi Zhang, Michał Król, Alberto Sonnino, Lixia Zhang, and Etienne Rivière. El passo: Efficient and lightweight privacy-preserving single sign on. *Proceedings on Privacy Enhancing Technologies*, 2021:70–87, 4 2021.

Appendix A

Data Cleaning & Analysis

A.1 Data Cleaning

```
# Data Import
import pandas as pd
from IPython.display import display

logs_path = "AuthApi-All-Struct-Logs.csv"
final_file_name = "Cleaned-Logs.csv"

print("Reading logs from file:", logs_path)
logs = pd.read_csv(logs_path)

print("Logs read successfully")

# Remove not used columns
logs = logs.drop(columns=['_id', '_index', '_score', '_type', 'env', 'level', 'pid', 'process', 'threadid',
    'version', 'message'])
# Remove duplicate columns
logs = logs.drop(columns=[col for col in logs.columns if col.endswith('keyword')])

print("Unnecessary columns removed")

logs = logs.rename(columns={'@timestamp': 'Timestamp',
                            'client_id': 'Client-ID',
                            'elapsedms': 'Elapsed-Milliseconds',
                            'endpoint': 'Endpoint',
                            'expiration_mins': 'Expiration-Minutes',
                            'redirect_uri': 'Redirect-URI',
                            'response_type': 'Response-Type',
                            'user': 'User',
                            'host': 'Host',
                            'user_group': 'User-Group',
                            'whitelisted': 'Whitelisted'})

print("Columns renamed")

# Replace invalid values
logs['Expiration-Minutes'] = logs['Expiration-Minutes'].replace('-', 0)
logs.replace('-', '', inplace=True)

```

```
38 # Convert columns to respective types
39 logs['Timestamp'] = pd.to_datetime(logs['Timestamp'], format='%b-%d,-%Y-@-%H:%M:%S.%f')
40 print("Timestamps-converted-to-datetime")
41
42 logs['Elapsed-Milliseconds'] = logs['Elapsed-Milliseconds'].str.replace(',', '').astype(float).round().astype(int)
43 print("Elapsed-Milliseconds-converted-and-rounded-to-integer")
44
45 logs['Expiration-Minutes'] = logs['Expiration-Minutes'].astype(int)
46 logs['Expiration-Minutes'] = logs['Expiration-Minutes'].replace(0, '')
47 print("Expiration-Minutes-converted-to-int")
48
49 logs = logs.sort_values(by='Timestamp')
50 print("Logs-ordered-by-Timestamp")
51
52 # Hide real Values
53 logs['Client-ID'] = logs['Client-ID'].apply(lambda x: f"Client-{logs['Client-ID'].unique().tolist().index(x)+1}" if x else '')
54 logs['Redirect-URI'] = logs.apply(lambda x: f"URI-{logs['Redirect-URI'].unique().tolist().index(x['Redirect-URI'])+1}" if x['Redirect-URI'] else '', axis=1)
55 logs['User'] = logs['User'].apply(lambda x: x if x == '' else f"User-{logs['User'].unique().tolist()[1::].index(x)+1}")
56 logs['User-Group'] = logs['User-Group'].apply(lambda x: f"Group-{logs['User-Group'].unique().tolist().index(x)+1}" if x else '')
57 logs['Host'] = logs['Host'].apply(lambda x: f"Machine-{logs['Host'].unique().tolist().index(x)+1}" if x else '')
58 # Add User group where it is missing
59 logs_copy = logs.copy()
60 logs_copy = logs_copy[(logs_copy['User'] != '') & (logs_copy['User-Group'] != '')]
61
62 user_group_dict = logs_copy.groupby('User')['User-Group'].first().to_dict()
63 user_group_dict = {k: v for k, v in user_group_dict.items() if v != ''}
64
65 logs['User-Group'] = logs['User'].map(user_group_dict).fillna(logs['User-Group'])
66
67 # Save results to file
68 logs.to_csv(final_file_name, index=False, decimal=',')
69 print(f"Data-saved-to-\"{final_file_name}\"")
```

Listing A.1: Data Cleaning

A.2 Data Analysis

```
1  # Data Import
2  import pandas as pd
3  import matplotlib.pyplot as plt
4  import seaborn as sns
5
6  logs_path = "Cleaned-Logs.csv"
7  print("Reading-logs-from-file:", logs_path)
8  logs = pd.read_csv(logs_path)
9  print("Logs-read-successfully")
10
11 # Percentiles by endpoint removing outliers
12 elapsedMsByEndpoint = logs[logs['Elapsed-Milliseconds'] < 4000]
13 plt.figure(figsize=(10, 8)) # Increase the figure size
```

```
14 plt.yticks(range(0, 4001, 200))
15 plt.title('Elapsed-Milliseconds-by-Endpoint')
16 sns.boxplot(x='Endpoint', y='Elapsed-Milliseconds', data=elapsedMsByEndpoint)
17
18 # Table: Elapsed Milliseconds metrics by Endpoint
19 order = ['[Login]', '[Callback]', '[Token]', '[WindowsToken]']
20 percentiles = elapsedMsByEndpoint.groupby('Endpoint')['Elapsed-Milliseconds'].describe(
       percentiles=[0.25, 0.50, 0.75])
21 percentiles = percentiles.reindex(order)
22 percentiles
23
24 # Login Hours By User Group
25 ## Extract hour from Timestamp column
26 loginHoursPerUseGroup = logs.copy()
27 loginHoursPerUseGroup['Hour'] = pd.to_datetime(loginHoursPerUseGroup['Timestamp']).dt.
       hour
28
29 ## Create the boxplot
30 plt.figure(figsize=(10, 6))
31 sns.boxplot(x='User-Group', y='Hour', data=loginHoursPerUseGroup)
32 plt.xlabel('User-Group')
33 plt.ylabel('Hour')
34 plt.title('User-Login-Hours-by-Group')
35 plt.show()
36
37 # Response Type usage by client
38 ## Filter logs
39 filtered_logs = logs[(logs['Endpoint'] == '[Token]') | (logs['Endpoint'] == '[WindowsToken]')]
40
41 ## Plot the count of logs with endpoint "Token" by response type and client id
42 plt.figure(figsize=(10, 6))
43 sns.countplot(x='Client-ID', hue='Response-Type', data=filtered_logs)
44 plt.xlabel('Client-ID')
45 plt.ylabel('Count')
46 plt.title('Response-Type-usage-by-Client-in-Token-Endpoints')
47 plt.legend(title='Response-Type')
48 plt.show()
```

Listing A.2: Data Analysis

Appendix B

Continuous Development / Continuous Integration

B.1 Jenkins File

```
1
2  def isToTriggerNugetCreation() {
3    return (env.TRIGGER_NUGET_CREATION != null && env.
         TRIGGER_NUGET_CREATION.toBoolean())
4  }
5
6  def isToGenerateAllXldPackages() {
7    return isToTriggerNugetCreation()|| (env.GENERATE_ALL_XLD_PACKAGES != null &&
         env.GENERATE_ALL_XLD_PACKAGES.toBoolean())
8  }
9
10 pipeline {
11     agent {label '*'}
12     options {
13         timeout(time: 1, unit: 'HOURS')
14         disableConcurrentBuilds()
15         parallelsAlwaysFailFast()
16     }
17     environment {
18         MSBUILD = "\"${tool 'MsBuild2022'}\""
19         SOLUTION_NAME = "AuthenticationApi"
20         ROOT_PROJECT = "AuthenticationApi.sln"
21         MAJOR_VERSION = "6"
22         BASE_VERSION = "$MAJOR_VERSION." + "${env.BRANCH_NAME != "master" ?
               "1" : "0"}" + ".$BUILD_NUMBER"
23         VERSION = "${BASE_VERSION}" + "${env.BRANCH_NAME != "master" ? "-${
               env.BRANCH_NAME}" : ""}"
24         SHORT_BRANCH_LENGTH = Math.min("${env.BRANCH_NAME}".length(), 32)
25         XLD_VERSION = "${BASE_VERSION}" + "${env.BRANCH_NAME != "master" ?
               "-${env.BRANCH_NAME}".replace("_","").substring(0,"${env.
               SHORT_BRANCH_LENGTH}".toInteger()) : ""}"
26         XLD_ENV = "AuthenticationApi_UAT"
27         XLD_ENV_2 = "AuthenticationApi.Management_UAT"
28         YEAR = VersionNumber(versionNumberString: '${BUILD_DATE_FORMATTED, "
               YYYY"}');
```

```
29          MSBUILD_ASSEMBLY_ARG =”-p:PackageVersion=\”${VERSION}\” -p:Authors
                =\”*\” -p:Company=\”*\” /p:RepositoryType=\”git\” /p:RepositoryUrl=\”${
                GIT_URL}\” /p:RepositoryBranch=\”${GIT_BRANCH}\” /p:RepositoryCommit
                =\”${GIT_COMMIT}\” /p:GenericPackageDescription=\”AuthenticationApi\” /p:
                GenericPackageTags=\”*\” /p:Copyright=\”Copyright * ${YEAR}\””
30          TEST_RESULT = ”NunitResult.xml”
31          TEST_RESULT_DIRECTORY = ’test_results’
32          COVERAGE_RESULT=”${TEST_RESULT_DIRECTORY}/**/coverage.opencover.xml
                ”
33          IS_PRE_RELEASE = ”${env.BRANCH_NAME != ”master” ? ”true” : ”false”}”
34          ARTIFACTORY_CREDENTIALS = credentials(’*’)
35          SQ_MSBUILD = ”\”${tool ’sonar-scanner’}\\SonarScanner.MSBuild.exe\””
36          SONAR_PROJECT_KEY = ”$AuthenticationAPI”
37          NUGET_OUTPUT_DIR = ”NuPkgs”
38          NUGET_OUTPUT_PATH = ”$WORKSPACE\\DeployPackage”
39          ARTIFACTORY_SUBFOLDER = ”${ARTIFACTORY_NUGET_SERVER_URL}” + ”/
                AuthenticationApi”
40      }
41
42      parameters {
43          booleanParam(defaultValue: false, description: ’Create all XLD Packages?’, name: ’
                GENERATE_ALL_XLD_PACKAGES’)
44          booleanParam(defaultValue: false, description: ’Generate all NuGets? Also Generates all
                XLD packages’, name: ’TRIGGER_NUGET_CREATION’)
45      }
46
47      stages {
48          stage(’Prepare’) {
49              steps {
50                  script {
51                      currentBuild.displayName = env.VERSION
52                  }
53                  bat ”xcopy /Y ..\\..\\..\\CommonBuildBatches CommonBuildBatches /I /s”
54              }
55          }
56
57          stage(’Setup Goup’) {
58              parallel {
59                  stage(’Begin SonarQube analysis’) {
60                      steps {
61                          withSonarQubeEnv(’Sonar-Sqill’) {
62                              bat ”${SQ_MSBUILD} begin /k:${SONAR_PROJECT_KEY} /n:${
                                    SOLUTION_NAME} /v:${MAJOR_VERSION} /d:sonar.cs.
                                    nunit.reportsPaths=${TEST_RESULT_DIRECTORY}\\*.
                                    nunit.results.xml /d:sonar.cs.opencover.reportsPaths=${
                                    COVERAGE_RESULT} /d:sonar.links.ci=${JENKINS_URL}
                                    job/${JOB_NAME} /d:sonar.links.scm=${GIT_URL} /d:sonar.
                                    visualstudio.solution=${SOLUTION_NAME}.sln /d:sonar.
                                    exclusions=**/*Test/**”
63                          }
64                      }
65                  }
66                  stage(’Simian’) {
67                      steps {
68                          bat ’cmd /c call .\\CommonBuildBatches\\Simian.bat’
69                      }
70                  }
71              }
```

```
72          }
73
74          stage('Restore') {
75              steps {
76                  bat "dotnet restore ${SOLUTION_NAME}.sln -s ${
                        ARTIFACTORY_NUGET_SERVER_URL} -s ${
                        ARTIFACTORY_NUGET_SERVER_URL}"
77              }
78          }
79
80          stage('Build') {
81              steps {
82                  bat "dotnet build .\\${ROOT_PROJECT} --configuration Release -t:Rebuild
                         --verbosity:m --nologo ${MSBUILD_ASSEMBLY_ARG}"
83              }
84          }
85          stage('Unit Tests') {
86              steps {
87                  script {
88                      bat "dotnet test ${SOLUTION_NAME}.sln --settings .\\nunit.runsettings
                             --no-restore --collect:\"XPlat Code Coverage\" --logger \"nunit;
                            LogFileName={assembly}.nunit.results.xml\" --results-directory ${
                            TEST_RESULT_DIRECTORY} --test-adapter-path:. /p:
                            CollectCoverage=true /p:CoverletOutputFormat=opencover"
89                  }
90              }
91              post {
92                  always {
93                      nunit(testResultsPattern: "${TEST_RESULT_DIRECTORY}/*.nunit.
                            results.xml", failedTestsFailBuild: true, failIfNoResults: true)
94                  }
95              }
96          }
97          stage('Scan Group') {
98              parallel {
99                  stage('Scan For Issues') {
100                     steps {
101                         script {
102                             def msBuild = scanForIssues tool: msBuild()
103                             def simian = scanForIssues tool: simian(pattern: 'simianOutput.xml
                                    ')
104                             publishIssues issues: [msBuild, simian], referenceJobName: '
                                    AuthenticationApi/master',
105                             qualityGates: [[threshold: 6, type: 'DELTA', unstable: false], [
                                    threshold: 1, type: 'DELTA_HIGH', unstable: false], [threshold:
                                    6, type: 'DELTA_NORMAL', unstable: false]]
106                         }
107                     }
108                     post {
109                         unstable {
110                             error 'Some quality gates have been missed'
111                         }
112                     }
113                 }
114                 stage("End SonarQube analysis") {
115                     steps{
116                         withSonarQubeEnv('Sonar-Sqill') {
117                             bat "${SQ_MSBUILD} end"
```

```
118                         }
119                     }
120
121                 }
122             }
123         }
124
125         stage("Sonar Quality Gate"){
126             steps{
127                 script {
128                     def qg = waitForQualityGate()
129                     if (qg.status != 'SUCCESS' && qg.status != 'OK') {
130                         catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE') {
131                             error "Sonar quality gate failed"
132                         }
133                     }
134                 }
135             }
136         }
137
138         stage ('Nuget: Publish') {
139             when {
140                 anyOf {
141                     changeset 'src/AuthenticationApi.Sso.Frontend/**'
142                     changeset 'src/AuthenticationApi.Sso.Backend/**'
143                     changeset 'src/AuthenticationApi.Sso.Shared/**'
144                     changeset 'src/AuthenticationApi.Auth.Shared/**'
145                     changeset 'src/AuthenticationApi.Windows.Backend/**'
146                     expression {
147                         return isToTriggerNugetCreation() || env.BUILD_NUMBER == "1"
148                     }
149                 }
150             }
151             steps {
152                 bat "dotnet build .\\src\\AuthenticationApi.Sso.Frontend\\AuthenticationApi.
                        Sso.Frontend.csproj --nologo -c Release --no-incremental ${
                        MSBUILD_ASSEMBLY_ARG}"
153                 bat "dotnet build .\\src\\AuthenticationApi.Sso.Backend\\AuthenticationApi.
                        Sso.Backend.csproj --nologo -c Release --no-incremental ${
                        MSBUILD_ASSEMBLY_ARG}"
154                 bat "dotnet build .\\src\\AuthenticationApi.Sso.Shared\\AuthenticationApi.
                        Sso.Shared.csproj --nologo -c Release --no-incremental ${
                        MSBUILD_ASSEMBLY_ARG}"
155                 bat "dotnet build .\\src\\AuthenticationApi.Auth.Shared\\AuthenticationApi.
                        Auth.Shared.csproj --nologo -c Release --no-incremental ${
                        MSBUILD_ASSEMBLY_ARG}"
156                 bat "dotnet build .\\src\\AuthenticationApi.Windows.Backend\\
                        AuthenticationApi.Windows.Backend.csproj --nologo -c Release --no-
                        incremental ${MSBUILD_ASSEMBLY_ARG}"
157
158                 bat "dotnet pack .\\src\\AuthenticationApi.Sso.Frontend\\AuthenticationApi.
                        Sso.Frontend.csproj --no-build -c Release -o ${NUGET_OUTPUT_DIR
                        } ${MSBUILD_ASSEMBLY_ARG}"
159                 bat "dotnet pack .\\src\\AuthenticationApi.Sso.Backend\\AuthenticationApi.
                        Sso.Backend.csproj --no-build -c Release -o ${NUGET_OUTPUT_DIR
                        } ${MSBUILD_ASSEMBLY_ARG}"
160                 bat "dotnet pack .\\src\\AuthenticationApi.Sso.Shared\\AuthenticationApi.Sso
                        .Shared.csproj --no-build -c Release -o ${NUGET_OUTPUT_DIR} ${
```

```
                    MSBUILD_ASSEMBLY_ARG}"
161             bat "dotnet pack .\\src\\AuthenticationApi.Auth.Shared\\AuthenticationApi.
                    Auth.Shared.csproj --no-build -c Release -o ${NUGET_OUTPUT_DIR
                    } ${MSBUILD_ASSEMBLY_ARG}"
162             bat "dotnet pack .\\src\\AuthenticationApi.Windows.Backend\\
                    AuthenticationApi.Windows.Backend.csproj --no-build -c Release -o ${
                    NUGET_OUTPUT_DIR} ${MSBUILD_ASSEMBLY_ARG}"
163             powershell ".\\CommonBuildBatches\\Nuget\\Publish-NugetPackages-
                    Artifactory.ps1 -ArtifactoryUrl $ARTIFACTORY_SUBFOLDER -
                    Prerelease $IS_PRE_RELEASE -PublishDir $NUGET_OUTPUT_DIR -
                    ArtifactoryUser ${ARTIFACTORY_CREDENTIALS_USR} -
                    ArtifactoryPwd ${ARTIFACTORY_CREDENTIALS_PSW}"
164         }
165     }
166
167     stage('Create and Push Package Auth Api') {
168         when {
169             anyOf {
170                 changeset 'src/AuthenticationApi/**'
171                 changeset 'src/AuthenticationApi.DataAccess/**'
172                 changeset 'src/AuthenticationApi.Logging/**'
173                 changeset 'src/uthenticationApi.Shared/**'
174                 changeset 'src/uthenticationApi.Domain/**'
175                 expression {
176                     return isToGenerateAllXldPackages() || env.BUILD_NUMBER == "1"
177                 }
178             }
179         }
180         steps {
181             bat "dotnet build --no-incremental src\\AuthenticationApi\\
                    AuthenticationApi.csproj -c Release /p:Platform=\"x64\";SignAssembly=
                    true"
182             bat "xcopy \"src\\AuthenticationApi\\bin\\x64\\release\\net8.0\" \"src\\
                    AuthenticationApi\\artifacts\" /c /e /i /Y"
183             xldCreatePackage artifactsPath:'src\\AuthenticationApi', manifestPath:'src\\
                    AuthenticationApi\\deployit-manifest.xml', darPath:'AuthenticationApi-
                    $XLD_VERSION.dar'
184             xldPublishPackage serverCredentials: 'xldeploy-akb:443', darPath: '
                    AuthenticationApi-$XLD_VERSION.dar'
185         }
186     }
187
188     stage('Create & Push Package Auth Api Management') {
189         when {
190             anyOf {
191                 changeset 'management/AuthenticationApi.Management/**'
192                 changeset 'management/AuthenticationApi.Management.Client/**'
193                 changeset 'management/AuthenticationApi.Management.Shared/**'
194                 changeset 'src/AuthenticationApi.DataAccess/**'
195                 changeset 'src/AuthenticationApi.Logging/**'
196                 changeset 'src/AuthenticationApi.Shared/**'
197                 changeset 'src/AuthenticationApi.Sso.Frontend/**'
198                 changeset 'src/AuthenticationApi.Sso.Backend/**'
199                 changeset 'src/AuthenticationApi.Sso.Shared/**'
200                 changeset 'src/AuthenticationApi.Auth.Shared/**'
201                 changeset 'src/AuthenticationApi.Windows.Backend/**'
202                 expression {
203                     return isToGenerateAllXldPackages() || env.BUILD_NUMBER == "1"
```

```
204                     }
205                 }
206             }
207             steps {
208                 bat "dotnet publish -c Release management\\AuthenticationApi.Management
                        \\AuthenticationApi.Management.csproj -v m --output management\\
                        AuthenticationApi.Management\\artifacts --nologo ${
                        MSBUILD_ASSEMBLY_ARG}"
209                 xldCreatePackage artifactsPath:'management\\AuthenticationApi.Management',
                         manifestPath:'management\\AuthenticationApi.Management\\deployit-
                        manifest.xml', darPath:'AuthenticationApi.Management-$XLD_VERSION.
                        dar'
210                 xldPublishPackage serverCredentials: 'xldeploy-akb:443', darPath: '
                        AuthenticationApi.Management-$XLD_VERSION.dar'
211             }
212         }
213
214         stage('Deploy Group') {
215             when { branch 'master' }
216             parallel {
217                 stage('Deploy with XLD Auth Api') {
218                     when {
219                         anyOf {
220                             changeset 'src/AuthenticationApi/**'
221                             changeset 'src/AuthenticationApi.DataAccess/**'
222                             changeset 'src/AuthenticationApi.Logging/**'
223                             changeset 'src/AuthenticationApi.Shared/**'
224                             expression {
225                                 return isToGenerateAllXldPackages()
226                             }
227                         }
228                     }
229                     steps {
230                         catchError {
231                             xldDeploy serverCredentials: 'xldeploy-akb:443', environmentId: '
                                    Environments/AuthenticationApi/$XLD_ENV', packageId:'
                                    Applications/AuthenticationApi/AuthenticationApi/
                                    $XLD_VERSION'
232                         }
233                     }
234                 }
235                 stage('Deploy with XLD Auth Api Management') {
236                     when {
237                         anyOf {
238                             changeset 'management/AuthenticationApi.Management/**'
239                             changeset 'management/AuthenticationApi.Management.Client/**'
240                             changeset 'management/AuthenticationApi.Management.Shared/**'
241                             changeset 'src/AuthenticationApi.DataAccess/**'
242                             changeset 'src/AuthenticationApi.Logging/**'
243                             changeset 'src/AuthenticationApi.Shared/**'
244                             changeset 'src/AuthenticationApi.Sso.Frontend/**'
245                             changeset 'src/AuthenticationApi.Sso.Backend/**'
246                             changeset 'src/AuthenticationApi.Sso.Shared/**'
247                             changeset 'src/AuthenticationApi.Auth.Shared/**'
248                             changeset 'src/AuthenticationApi.Windows.Backend/**'
249                             expression {
250                                 return isToGenerateAllXldPackages()
251                             }
```

```
252                         }
253                     }
254                     steps {
255                         catchError {
256                             xldDeploy serverCredentials: 'xldeploy-akb:443', environmentId: '
                                    Environments/AuthenticationApi.Management/$XLD_ENV_2',
                                    packageId:'Applications/AuthenticationApi/AuthenticationApi.
                                    Management/$XLD_VERSION'
257                         }
258                     }
259                 }
260             }
261         }
262
263         stage('Tag Git repo') {
264             when { branch 'master' }
265             steps {
266                 bat "git tag -a $VERSION -m \"$SOLUTION_NAME-$MAJOR_VERSION
                        \""
267                 bat "git push origin $VERSION"
268             }
269         }
270     }
271     post {
272         always {
273             script {
274                 if (env.BRANCH_NAME != "master" ) {
275                     cleanWs()
276                 }
277             }
278         }
279     }
280 }
```

Listing B.1: Jenkins File

B.2 XL Deploy Environments

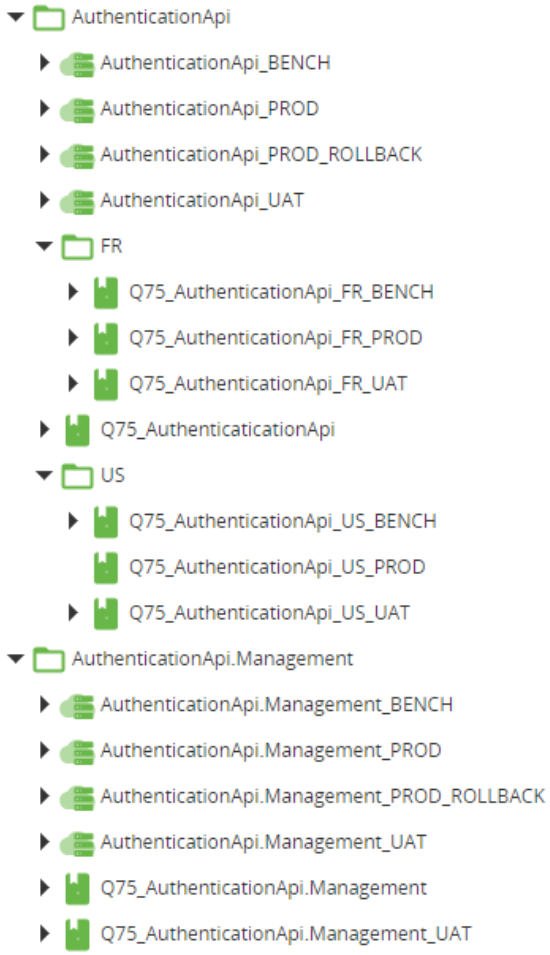


Figure B.1: XL Deploy Environments

Appendix C

Integrating Libraries

C.1 Backend Library

1. Add last version of package "AuthenticationApi.Sso.Backend" to your backend app

2. App settings file must be updated to include the settings bellow. This values are already setup in the XLD dictionaries for all environments

```
"Jwt": {
  "JwtPublicKey": "{{AuthApiJwtPublicKeyV2}}",
  "ValidIssuer": "{{AuthApiJwtIssuerV2}}"
}
```

Listing C.1: Backend Library app settings

3. Program.cs file must be edited to include the following

```
using AuthenticationApi.Sso.Backend;

...

builder.AddAuthApiJwtAuthenticationV2();
builder.Services.AddAuthorization();

// Before builder.Build();

...

// After app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

// Before app.Run();
```

Listing C.2: Backend Library program.cs

C.2 Frontend Library

C.2.1 Mandatory Steps

1. A frontend web app will need to import the latest version of package AuthenticationApi.Sso.Frontend

2. App settings file must be updated to include the following settings

```
"SsoApi": {
  "ClientId": "{{AuthApiClientIdV2}}",
  "ApiUrl": "{{AuthApiUrl}}",
  "ReceiveCodePath": "/auth/v2/code",
  "BeforeLoginBlackList": "", // if you want some page to not be used after
      authentication redirect. Add relative routes separated by semi column. You do not
      need this setting if it is an empty string.
  "AfterLoginRedirectUrl": "" // if you want the user to be redirected to a specific page
      after login. You do not need this setting if it is an empty string.
}
```

Listing C.3: Frontend Library app settings

3. In Program.cs add the bellow.

```
builder.AddAuthApiV2(useRefit: true); // If you do not use refit do not include the
    parameter
builder.Services.AddCascadingAuthenticationState();
```

Listing C.4: Frontend Library program.cs

There are optional parameters:

- useRefit: If you use refit to communicate with backend (default false)
- useLocalStorage: should be true, but if you want your user to use session storage for authentication set it as false (default true)

4. In _Imports.razor, if not present, add

```
@using Microsoft.AspNetCore.Components.Authorization
```

Listing C.5: Frontend Library _Imports.cs

5. If you declare HTTP clients, you have to use AddHttpClientAuthenticated method or add manually the AuthenticationHeaderHandler (If you need a new override of AddHttpClientAuthenticated add it in Auth Api package). If you use refit then, use the ServiceFactory instead to register http clients (step 7)

```
builder.Services.AddHttpClientAuthenticated<IBackEndClient, BackEndClient>(client
    => client.BaseAddress = new Uri(builder.HostEnvironment.BaseAddress));
```

Listing C.6: Frontend Library add authenticated http client

```
builder.Services.AddHttpClient<IBackEndClient, BackEndClient>(client => client.
    BaseAddress = baseUri)
    .AddHttpMessageHandler<AuthenticationHeaderHandlerRefit>(); //If you use refit

// OR
```

```
5 builder.Services.AddHttpClient<IBackEndClient, BackEndClient>(client => client.
      BaseAddress = baseUri)
6     .AddHttpMessageHandler<AuthenticationHeaderHandler>(); //If you do not use refit
```

Listing C.7: Frontend Library manually add authenticated http client

6. If you use refit then add the following to your Program.cs

```
1     builder.Services.AddScoped<IServiceFactory, ServiceFactory>(); // Always need to
          add
2
3     // IApplicationService must receive a refit interface in the constructor and create a
          service with that interface using IServiceFactory (example bellow)
4     builder.Services.AddScoped<IDummyService, DummyService>();
```

Listing C.8: Frontend Library Add Refit Service

```
1  public class DummyService : IDummyService
2  {
3      // This interface uses Refit HTTP annoations
4      private readonly IDummyApi _dummyApi;
5
6      public DummyService(IServiceFactory serviceFactory)
7      {
8          // You can optionally pass a URI to the Create method
9          _dummyApi = serviceFactory.Create<IDummyApi>();
10     }
11
12     public async Task<IEnumerable<DummyDomain>> GetAllDummies()
13     {
14         return await _dummyApi.GetAllDummies();
15     }
16 }
```

Listing C.9: Frontend Library Example Refit Service

7. In App.razor or Routes.razor add additional assemblies on Router tag

```
1     <Router AppAssembly="..."
2         AdditionalAssemblies="new[] { typeof(AuthenticationApi.Sso.Frontend.
              Components.OneTimeCodeV2.OneTimeCodePageV2).Assembly }">
```

Listing C.10: Frontend Library OneTimeCodeV2 Page Integration

8. If you are using a Blazor app with API and Frontend, then, in the API Program.cs add the last line in the code bellow, to include additional assemblies

```
1 app.MapRazorComponents<App>()
2 ...
3 .AddAdditionalAssemblies(typeof(AuthenticationApi.Sso.Frontend.Components.
      OneTimeCodeV2.OneTimeCodePageV2).Assembly);
```

Listing C.11: Frontend Library Add Initial Assemblies

Additionally, in App.razor file the render mode for Routes and HeadOutlet

```
1 <HeadOutlet @rendermode="new InteractiveWebAssemblyRenderMode(prerender: false)
      " />
2 <Routes @rendermode="new InteractiveWebAssemblyRenderMode(prerender: false)" />
```

Listing C.12: Frontend Library Update Render Mode

9. In all layouts add the AuthorizeView tag with the NotAuthorizedComponent inside: The NotAuthorizedComponent will automatically authenticate the user if it is not authenticated and you can customise what the user sees while the authentication is on going and when it is not authenticated.

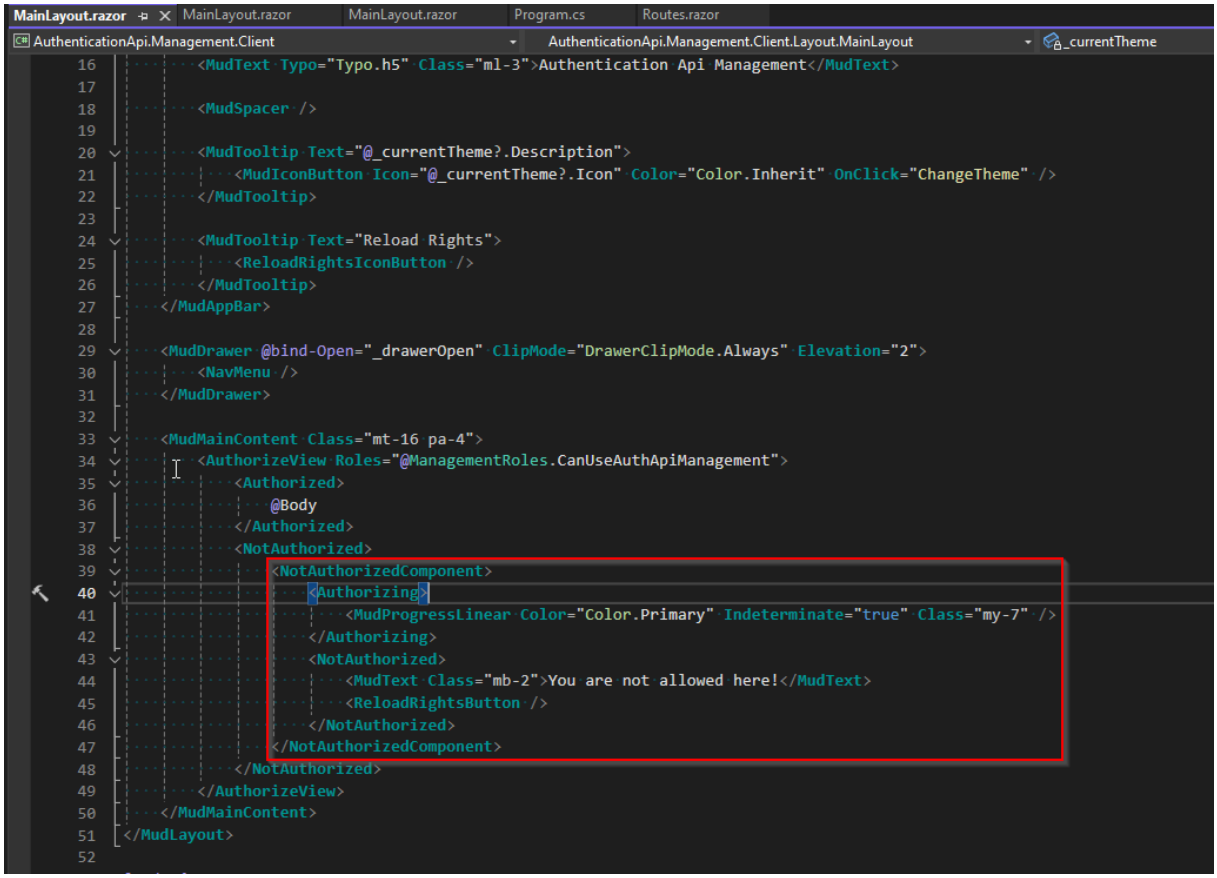


Figure C.1: NotAuthorizedComponent example

- Authorizing tag - Not mandatory, if nothing is provided, the default Authorizing behaviour is a <p>tag saying "Logging you in...". In this example we use MudBlazor to provide a loading bar for a better UI experience (Inside the Authorizing tag),
- NotAuthorized tag - Not Mandatory, but highly recommended. Here, we put a custom message when user is not authenticated with a reload rights button so the user can just click it in case something went wrong.

C.2.2 Custom Reload Rights Button

1. Create a razor class that inherits class ReloadRightsButton from the Frontend package
2. Now you can customize this button as you wish, it will handle all the authentication part for you. Optionally you can override the parent methods OnAfterLogin or

OnAfterLoginAsync (use only one) to do something after authentication, such as showing a pop up.

```
@using AuthenticationApi.Sso.Frontend.SsoClient
@inherits AuthenticationApi.Sso.Frontend.Components.ReloadRightsButton;

@inject ISnackbar Snackbar

<MudButton Color="Color.Primary"
           Variant="Variant.Filled"
           OnClick="RefreshRights"
           ClickPropagation="false"
           Disabled="@LoggingIn">
    Reload Rights
</MudButton>

@code {
    public override void OnAfterLogin(ELoginType loginType)
    {
        Snackbar.ShowRefreshRightsSnackbar(loginType);
    }
}
```

Listing C.13: Frontend Library Custom Reload Rights Button Example

OnAfterLogin and OnAfterLoginAsync methods receive the authentication state that tells you what type of authentication was performed:

- Sso: means the user will be redirected to AuthenticationApi to do the login. In this scenario your app will be exited and user will come back to it after authentication is over. That is why LoggingIn remains false after the method has completed, because we are waiting for the redirect to happen.
- RefreshToken: This login type is almost instant and your app will not be exited to do it. So, if you receive this state you are sure that the user is now authenticated.
- None: Not used by this ReloadRightsButton component.