IY3821: Final Year Project - Final Submission

IY3821: Modern Authentication - How Should Authentication Really Be Delegated?

Mohamed Yusuf Mohamed Javid

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science with Information Security

Department of Computer Science
Royal Holloway, University of London

April 7, 2024

بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

In the name of Allah, the most Gracious and the most Merciful.

Acknowledgments

This paper is dedicated to my dad, my mum and my sister. Sacrifices of whom have made the possibility of the completion of this 4 year undergraduate course even conceivable.

Table of Contents

1 Glossary . . . 2

2 Introduction . . . 3

2.1 Authentication . . . 4

2.2 Authorisation . . . 5

2.3 Delegated authentication and authorisation . . . 5

3 Understanding of Core Concepts of OAuth2.0 Authorisation Framework . . . 7

3.1 Authorization Code Grant . . . 10

3.2 Proof Key for Code Exchange . . . 10
3.2.1 Public Clients . . . 11
3.2.2 Confidential Clients . . . 11

3.3 Implicit Flow . . . 12

4 OAuth2.0 Pitfalls with Current Implementations . . . 14

4.1 Facebook OAuth2.0 Framework Vulnerability . . . 14

4.2 Improper Implementation of Implicit Grant Type . . . 15

4.3 Flawed Scope Validation . . . 15

4.4 Vulnerability to Session Fixation Attacks . . . 16

4.5 Token Leakage and Impersonation Risks . . . 16

4.6 Authorization Code Interception Attack . . . 17

4.7 Cross Site Request Forgery . . . 17

4.8 Absence of NONCE in OAuth2.0 Implementations . . . 18

5 Project Description and Objectives . . . 19

5.1 The Client Application . . . 19

5.2 User Authentication Flow . . . 19
5.2.1 User Authentication Initiation . . . 19
5.2.2 User Registration . . . 19
5.2.3 User Onboarding Flow . . . 21
5.2.4 Storage of User Details . . . 23

5.2.5 User Login . . . 23

5.3 Roles and Authorities . . . 23
5.3.1 Employee . . . 25
5.3.2 Manager . . . 25
5.3.3 Admin . . . 25

6 Project Structure and Code Review . . . 26

6.1 Configurations . . . 28
6.1.1 application-dev.yml for the Client . . . 28
6.1.2 application-dev.yml for the Authorisation Server . . . 30
6.1.3 application-dev.yml for the Resource Server . . . 31

6.2 Security Configurations . . . 31
6.2.1 Security Configuration for The Client . . . 31
6.2.1.1 CSRF Configuration . . . 32
6.2.1.2 CORS Configuration . . . 33
6.2.1.3 Authentication Success Handler . . . 34
6.2.1.4 Debugging Access Token Exchange between Client and AS . 35
6.2.1.5 Spring Boot Firewall Rule Exception - Path Traversal . . . 35
6.2.2 Security Configuration for The Authorisation Server . . . 36
6.2.2.1 Using Plaintext Passwords for Demonstration . . . 36
6.2.2.2 Facilitating JWT Token Management . . . 37

6.3 Spring Boot Repository Layer . . . 38
6.3.1 CompanyRepository . . . 38
6.3.2 Role Repository . . . 38

6.4 Spring Boot Service Layer . . . 39
6.4.1 Custom User Service Implementation for Local Authorisation Server . 40
6.4.2 Custom User Service Implementation for Google's Authorisation Server 42

6.5 Spring Boot Entity Layer . . . 43

6.6 Spring Boot Controller Layer . . . 48

6.7 Hurdles During Development . . . 52
6.7.1 Hostname Cookie Sharing Issue . . . 52
6.7.2 Postman Session Syncing Workaround . . . 53

6.8 Potential Improvements and Considerations . . . 53
6.8.1 Hiding The Authorisation and Resource Servers Inside a LAN . . . 53
6.8.2 Client Authentication with Private-Key JWT . . . 54
6.8.3 Utilising Azure Key Vaults . . . 55
6.8.4 Multi-Factor Authentication and Containerisation with Docker . . . 55

7 Professional Issues . . . 57

7.1 Responsible Disclosure of OAuth2.0 Vulnerabilities . . . 57

7.2 Ambiguity in Security Best Practices . . . 57

7.3 Unit Testing Methods and Test-Driven Development (TDD) . . . 58

7.4 Linguistic Inconsistencies . . . 58

8 Conclusion 60

Abstract

In the rapidly evolving landscape of web application development, ensuring secure user authentication and authorisation has become a critical concern for organisations worldwide. As the complexity of applications grows and the threat of cyber-attacks looms larger, it is imperative to adopt robust and reliable authentication mechanisms that safeguard user data and maintain the integrity of the system. This paper delves into the realm of modern authentication, focusing on the widely adopted OAuth 2.0 and OpenID Connect standards, and explores their implementation using the powerful Spring Boot Java framework.

The research presented in this paper aims to address the challenges faced by developers when implementing secure authentication systems, particularly in the context of Enterprise Resource Planning applications. By leveraging the capabilities of OAuth 2.0 and OpenID Connect, this study demonstrates how delegated authentication and authorisation can significantly enhance the security posture of web applications while simplifying the development process.

Through a meticulous analysis of common pitfalls and vulnerabilities associated with OAuth 2.0 and OpenID Connect implementations, such as the Facebook OAuth 2.0 Framework vulnerability, improper implementation of the Implicit Grant Type, and flawed scope validation, this paper highlights the importance of adhering to security best practices and provides practical solutions to mitigate these risks.

The project showcased in this paper involves the development of a simplified ERP HCM (Human Capital Management) web application using Spring Boot Java, which serves as a testament to the framework's ability to streamline the implementation of secure authentication and authorisation mechanisms.

Moreover, this research explores the benefits of delegated authentication and authorisation, which enable organisations to outsource these critical processes to trusted identity providers, such as Google, and custom-built authorisation servers. By leveraging the expertise of these providers, developers can focus on core application functionality while ensuring a robust security foundation.

1. Glossary

Authentication - A method used to verify the identity of an actor.

Authorisation - A Process determining access rights to resources based on policies and principles defined by the identity provider or resource provider.

Delegated Authentication/Authorisation - A security model where the authentication or authorisation process is handled by an external entity or service. This approach allows for a more centralised management of credentials and permissions.

ERP - Enterprise Resource Planning, a software built for business management. This could include services like employee appraisal, stock management etc.

HCM - Human Capital Management, a system that aims to deal with employee's acquisition, management, and optimisation.

Spring - A comprehensive framework for building Java applications.

Spring Boot - A project that makes it easy to create stand-alone, production-grade Spring based Applications.

Spring Bean - A Spring Bean is an object managed by the Spring IOC (Inversion of Control) container that is instantiated, assembled, and otherwise managed by the Spring framework.

MVC - An architectural pattern that separates an application into three main logical components: the Model (data and business logic), the View (user interface), and the Controller (interface between Model and View), facilitating a clear separation of concerns.

React - A JavaScript library for building user interfaces using single page applications.

Micro Services - An architectural style that structures an application as a collection of services, such that each service is responsible for only one family of tasks. The goal here is to achieve isolation between components of a system.

Monolithic - An application that is designed as a single and indivisible unit.

2. Introduction

The goal of this document is to conduct a detailed overview of authentication and authorisation methods in modern applications. This area of cybersecurity is vast and a topic of consistent and continuous research. Authentication methods and implementations vary from application to application, but the main decision makers regarding which implementation and/or which authentication principles to implement depend on the domain, the device used, the user base, the risks, and the affordability. This document goes over one such implementation of a widely adopted authentication and authorisation standard called OAuth2.0.

This document in essence is a project report for a project that aims to demonstrate a secure implementation of OAuth2.0 with the example of an extremely trivial **Enterprise Resource Planning Human Capital Management (ERP HCM)** web application. Again, the goal here is not to build a robust ERP HCM solution, rather, it is to demonstrate how any application with a similar domain should be built with a security first approach using Spring Boot Java. Although, the decision to use Spring Boot as a technology for back-end application development is entirely optional and even alternative to many other robust back-end tools like Node.js, Firebase, Express and many more. Spring Boot does come with rigorously tested libraries and packages that significantly reduce the need for re-inventing the wheel with certain aspects of authentication and authorisation, which make it an attractive choice.

Moreover, the project also contains a front-end aspect to it which is developed using ReactJS. Again, this decision is purely a matter of personal preference and adds no additional level of security to this domain exclusively. This however does not mean that the front-end is free from security issues and that attention to security when it comes to the front-end isn't of importance. It just means that front-end security isn't the scope of this project, except for CSRF prevention, which will be later discussed in this report. For the most part, the reason for the front-end aspect is for demonstration purposes only and hence, not much attention is given to graphic designing best practices. Perhaps, this would be a matter of consideration given a larger time frame and scope for this project.

In terms of the software engineering aspects of this project, Apache Maven is a tool used for dependency management and packaging the application, facilitating an efficient and secure development process by managing and updating library dependencies for the development environment. Docker is used for containerisation such that the examiner for this project is able to run and test the application without facing any environment related issues. Details on how to set this up is provided in the projects README.MD document. Finally, API testing in this project is conducted manually using Postman, which is chosen for its effectiveness in testing REST APIs, ensuring they meet the required functionality and security standards (Westerveld, 2021).

Authentication and authorisation are perhaps the most important elements of security in any sense. Be it cyber, physical or even national. The ability to verify the authenticity and the authority of an actor can determine if a system is robust or not. To understand how to implement these in the context of modern web-applications and the context of available adversarial methods, it is first important to understand authentication and authorisation in detail. The concept of 'authentication' and 'authorisation' is usually accompanied by 'identification' as described by Syed Idrus et al. (2013) but in the context of this paper, we will just merge identification with authentication since identification is more relevant when viewing from the perspective of the user. In the perspective of a back-end system, we will simply define 'identification' as an aspect of 'authentication' which answers the question 'who is the actor?'.

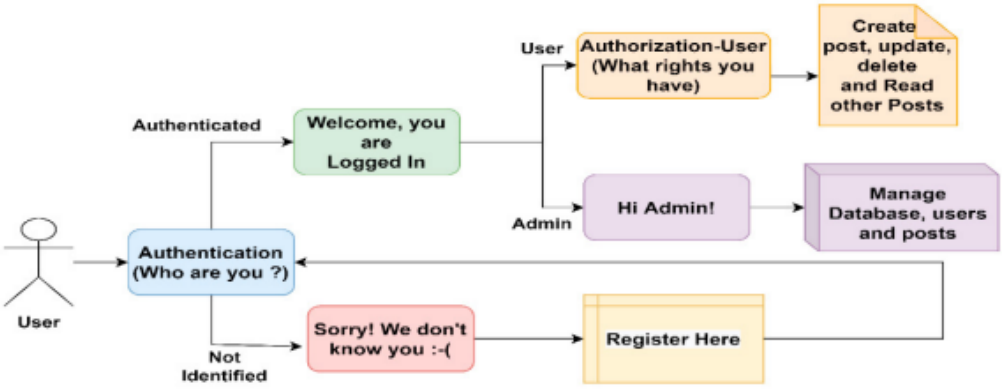

Figure 2.1: A high level overview of a generic authentication and authorisation system (Pant et al., 2022)

2.1 Authentication

The goal of authentication is to verify without a doubt that an actor is really what it says it is. In the case of an ERP HCM solution, this actor could be an employee or an admin. However, in other cases, actors could be systems attempting to access other systems (this concept of human-less authentication is more relevant in the context of delegated authentication). Furthermore, historically, the methods that are used in authentication have always revolved around 4 questions (Cankaya, 2011):

- Something an actor knows (passwords, passphrase, pins)
- Something an actor has (access cards)
- Something an actor is (fingerprint, face id)
- Something an actor produces (signature)

In an authentication system, these four aspects add the element of secrecy or confidentiality. For example: in the case of 'something an actor knows' a 4-digit pin-code would be the element of confidentiality and an authentication system's security posture would be evaluated by how it handles that 4-digit pin-code. The decision of which of these methods to use in an authentication system is entirely subjective and is based on factors like the domain, the user base, the affordability, the user experience and the ratio of acceptable false positive to false negatives. In most cases, frameworks and standards that have been rigorously tested have been developed such that when used properly will produce a secure authentication system free from automated attacks like SQL injections. However, these frameworks can not secure an authentication system against domain specific attacks that are caused by logic bugs in code. An example of a logic error in code would be improper error handling when a registration attempt to the system is made with an email (or any other unique identification) that already exists, this shown clearer in figure dash. Modern authentication systems use cryptographic methods like hashing and salting to protect the confidential element of the authentication system, like passwords or passphrases. Figure dash shows the data flow with regard to hashing.

2.2 Authorisation

The goal of authorisation is to ascertain an actors' authority in a system. This process essentially answers the following questions (Syed Idrus et al., 2013):

- Is the authenticated actor authorised to access the resource R?
- Is the authenticated actor authorised to perform the operation O?
- Is the authenticated actor authorised to perform the operation O on the resource R?

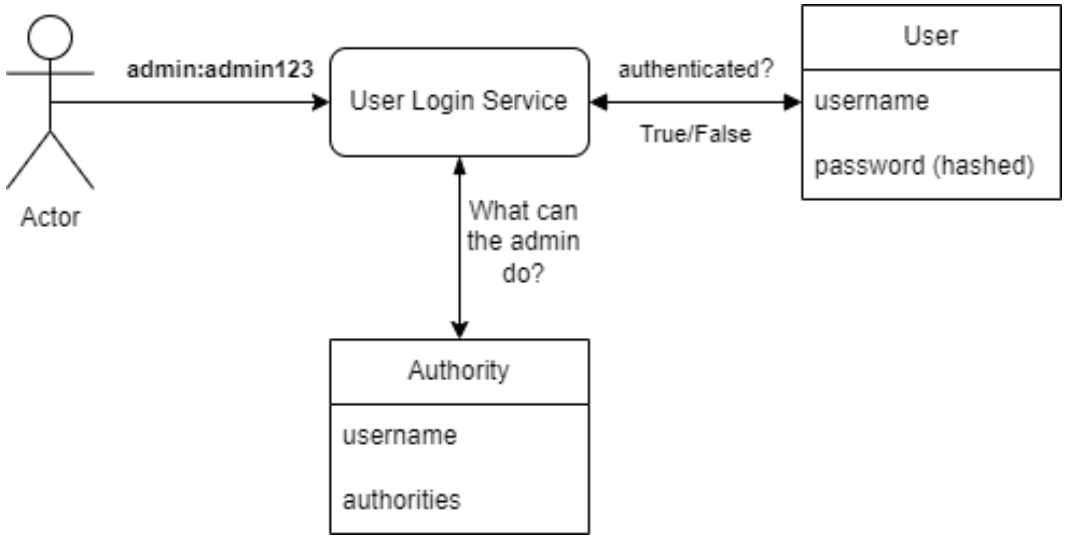

Figure 2.2: A generic flow for an authorisation request.

In modern systems, authorisation is a key element since it allows for users of systems to be segregated in different categories and even allow systems to return different applications depending on the authority of the user, this is sometimes known as Role Based Access Control (RBAC). Generally, authorities are stored in systems with READ, WRITE or EXECUTE features and followed by the description of the resource. For example: an admin with the authority to view first names of all users in the system would have READ_ALL_USERS authority. Although this convention of naming authorities is not a hard set rule, it seems to be consistent with most applications in the industry due to its descriptive nature. Figure 2.2 shows a generic data flow where a login service queries a database to figure out an admin user's authorities.

2.3 Delegated authentication and authorisation

Over the years, delegated authentication and authorisation have emerged as cornerstone practices for enhancing the security and efficiency of managing user identities and access permissions. This method involves outsourcing the authentication and authorisation process to a system that is more robust and secure. Identity providers can implement advanced security protocols, including multi-factor authentication and real-time threat detection, thereby enhancing the overall security posture without imposing additional burdens on application developers.

Furthermore, delegated authorisation complements the authentication process by enabling applications to act on behalf of the user, accessing third-party resources without directly

handling the user's credentials. This model is especially relevant when applications require access to external services or data, facilitating a permission-based access mechanism governed by the user's explicit consent. Although this authentication system may seem complex, it encourages the development of systems using a microservices architecture rather than a monolithic one. Moreover, dividing the system into specific areas of expertise: like a server dedicated solely to authentication excelling in its tasks, and another focused on data retrieval performing optimally— allows for a 'separation of concerns'. This design inherently enhances security (Chatterjee et al., 2022).

However, the adoption of delegated authentication and authorisation necessitates a thorough understanding of the underlying protocols and a vigilant approach to the configuration of trust relationships among the stakeholders—users, applications, and identity providers. Inadequacies in the implementation or misinterpretation of consent flows can inadvertently introduce vulnerabilities, highlighting the imperative of strict adherence to established protocols and best practices. These vulnerabilities are documented later in this document.

In sum, the shift towards delegated authentication and authorisation represents a significant evolution in the domain of digital identity management. By leveraging the expertise of identity providers like Google, applications can offer a secure, scalable, and user-centric authentication experience, in alignment with the overarching objective of a security-first approach in application development.

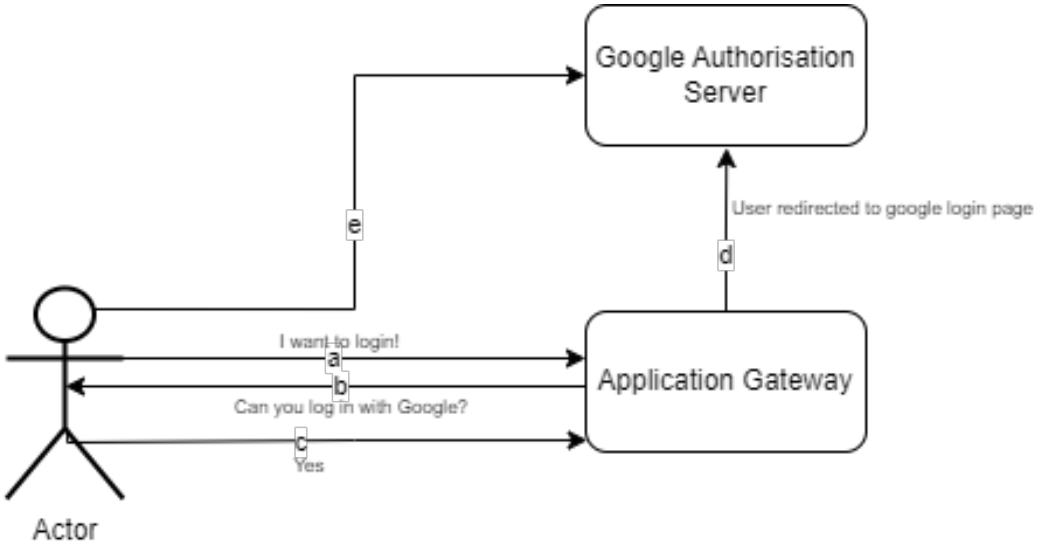

Figure 2.3: A simple abstraction of delegated authentication to Google's authorisation server.

Figure 2.3 presents a straightforward abstraction of a scenario where an actor accesses an application that lacks a local authentication system. The reasoning for not implementing a local authentication system has been previously discussed. This figure illustrates how the application redirects the user to an authorisation server when the user initiates a login request. It's important to note that this depiction is not meant to represent any specific delegated authentication technique, but rather to provide a high-level overview.

3. Understanding of Core Concepts of OAuth2.0 Authorisation Framework

OAuth2.0, as introduced in its official documentation (Hardt, 2012) and in this report, is a robust framework developed to facilitate secure and standardised authorisation in web-based applications and services. It is specifically designed to allow third-party access to server resources without exposing user credentials, thereby ensuring enhanced security in digital interactions. By granting specific tokens instead of direct access to user credentials, OAuth2.0 establishes a secure method for users to authorise third-party applications to access their server resources on their behalf (Shurdi et al., 2020). This authorisation mechanism is pivotal in today's interconnected digital environment, where security and privacy are of paramount importance. Through OAuth2.0, various web services can securely interact and exchange information, maintaining the integrity and confidentiality of user data.

In the OAuth2.0 framework, the roles are crucial in establishing a secure and efficient authorisation process. The framework categorises roles into four distinct entities: **the Resource Owner, Client, Authorisation Server, and Resource Server** (Hardt, 2012).

The **Resource Owner (RO)**, typically the end user, possesses control over the data or resources being requested. This role is pivotal in the initial stage of the authorisation process, where the owner grants or denies permission for resource access.

> *In a simple application like the ERP HCM solution, the RO is simply the end user (human) that attempts to qualify the client to access their information from an authorisation server. However, in more complex environments where server isolation is the theme and authentication between servers is required, the RO can be considered to be the entity that owns the resource. The reason for this distinction is to make it clear the protocol is not limited to RO being a human entity.*

The **Client (C)** refers to the application seeking access to the resource owner's data. This entity initiates the request for access and must be authorised by the resource owner and authenticated by the authorisation server to proceed.

> The statement, "The client is the entity the RO interacts with", often leads to the misconception that the client is synonymous with the frontend application. However, this interpretation oversimplifies the complexity and responsibilities of the client within the context of the OAuth 2.0 framework. In reality, the client refers to the application responsible for managing the logic associated with an OAuth 2.0 implementation. Typically, this involves backend systems rather than frontend interfaces.
>
> Positioning an OAuth 2.0 implementation solely within the frontend would be a severe security malpractice due to access tokens being dealt in the front channel. This will be clearer when the authorisation code flow is described in the later sections of the document.

The **Authorisation Server (AS)** plays a key role in the OAuth2.0 protocol. It is responsible for authenticating the resource owner's identity and granting the necessary authorities to the client. This server acts as the intermediary, ensuring that the process adheres to the prescribed security standards defined in the OAuth2.0 official documentation.

Lastly, the **Resource Server (RS)** hosts the protected resources, It is tasked with handling requests from the client. These requests must be accompanied by valid authorisation credentials, typically in the form of access tokens, to ensure that only authorised entities gain access to the requested resources.

Together, these roles form a cohesive system that underpins the OAuth2.0 authorisation framework. Each role is integral to the framework, functioning cohesively to ensure that resource access is secure, controlled, and consistent with the resource owner's permissions. This systematic approach to authorisation not only improves security but also streamlines the process of granting third-party applications access to server resources in a controlled manner.

A diagram of a classical implementation of OAuth2.0 Authorisation flow inspired by a talk given by Barbettini (2018) is shown. The diagram is not an exact replica, rather an inspiration such that can set a foundation for the code review section of this document.

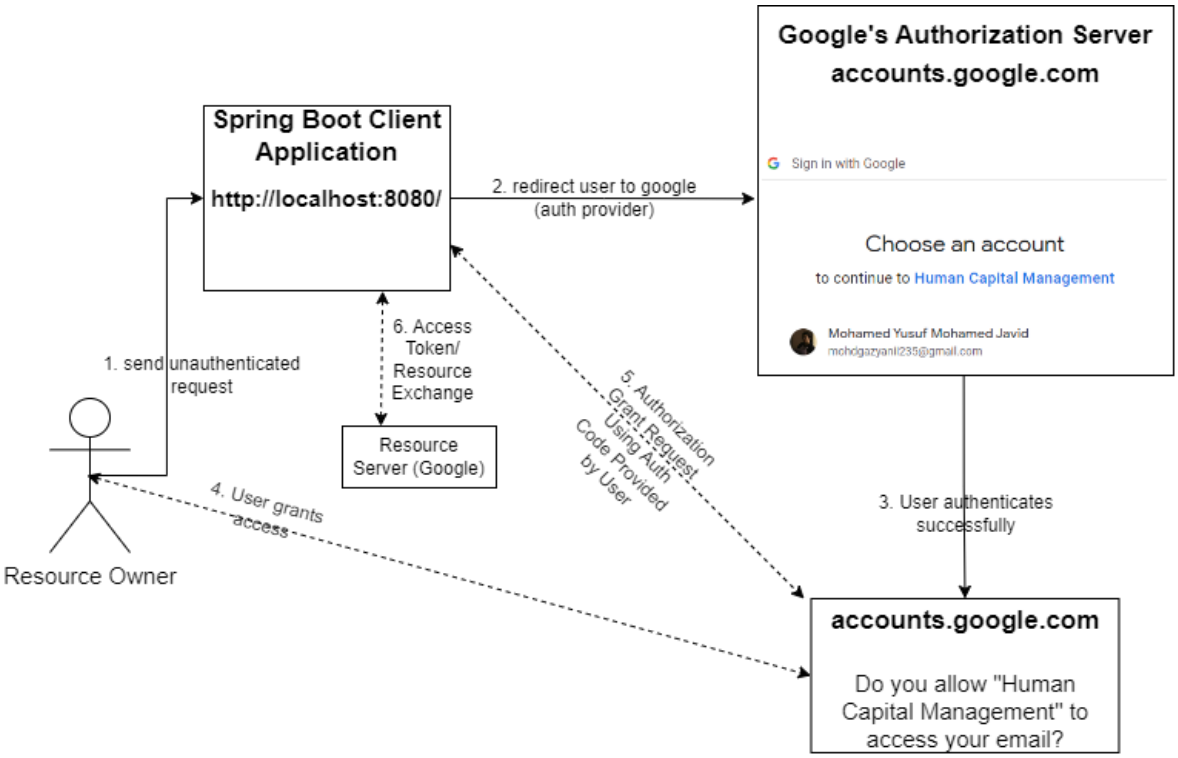

Figure 3.1: OAuth2.0 Data flow - specifically "Authorization Code Grant Type"

Figure 3.1 shows a complete implementation of OAuth2.0 with the integration between the different components of the system. The figure has numbered arrows that demonstrate the sequence of the request/response procedure. The data flow in OAuth2.0 starts when **RO** attempts to visit **C**. **C** validates the request using some authenticated session id and if **RO** hasn't previously authenticated, **RO** will be redirected to **AS** web-server. during this redirection, **RO** tells the **AS** some key information about **C**. This information includes the following things:

- *client_id* - A parameter used by **AS's** to identify **C**.
- *scope* - A parameter that classifies the range of access **C** can have approved by **RO** and **AS**.
- *response_type* - This is essentially what **RO** tries to ask **AS** to respond with. In most cases the value for this is "*code*" referring to "*authorization_code*".

Since this is a redirect, the information travels through a front channel, there is a potential risk of interception by malicious entities, particularly if the request is not encrypted with TLS. Furthermore, since this is typically an HTTP GET request, the details of the request are fully visible in the browser's address bar. As such, for developers, this environment is considered inherently insecure (Barbettini, 2018) as developers should always assume the worst-in this case, the worst would be a malicious party that is able to see **RO's** browser window. This means that one cannot assume any level of trust in the phase of the communication. It's crucial to apply stringent security measures, with the knowledge that any data transmitted in this manner could be exposed to unauthorised parties. So, the way OAuth2.0 circumvents this problem for the **AS** is by requiring security validation from both the **RO** and **C**. **RO** authorises itself by showing **C** the *authorization_code* (this way, **RO's** actual credentials are never shared with **C**), while **C** authorises itself to **AS** by showing the *authorization_code*, *client_id* & *client_secret* where the *authorization_code* is provided by **RO** and *client_id* & *client_secret* are provided by **AS** when the implementation is set up (Hardt, 2012). This process will be described later in this document.

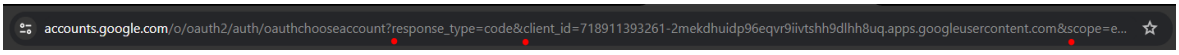

Figure 3.2: OAuth2.0 Dataflow

Figure 3.2 shows the browser-redirect GET request made on the front-channel of the **RO**.

When **C** is successfully able to authenticate itself to **AS** by providing it the correct *client_id* & *client_secret* as well as the correct *authorization_code* sent through the front-channel of the **RO**, **AS** returns back to **C** with something called an *access_token*. The *access_token* is a very sensitive piece of information in this entire system and must be stored by **C** for as long as the access is required. This *access_token* is used by **C** to authenticate to **RS**. Once authenticated to **RS**, **C** will receive from **RS** all the data that was defined in the **scope**.

In theory, an *access_token* can be implemented in multiple different formats, and methods. The decision for which method to be used depends on the specific security requirements of **RS** (Jones and Hardt, 2012).

Since the subject of this document is not only authorisation, the spring Boot client is also expecting authentication capabilities from **AS** as well. This is where the importance of OpenID Connect comes. OpenID Connect is an identity layer on top of OAuth2.0 (Barbettini, 2018) which facilitates **C** in authenticating an **RO**'s identity based on the authentication performed by **AS**, and obtaining an *id_token* which is usually a JWT token (Jones et al., 2015b) (JSON Web Token). In essence, a JWT token is simply a base64 encoded string of JSON data of

a user profile. This includes the user's first name, last name, email, profile picture etc. The *id_token* is never sent to the **RS** since it is not used as a method to authenticate or validate any component. The *id_token* is simply ready by the Client application to identify a user (Hardt, 2012).

3.1 Authorization Code Grant

The protocol described above in Figure 3.1 is a demonstration of something called the "Authorization Code Grant". The "Authorization Code Grant" is one of several methods of authentication/ authorisation defined in the OAuth2.0 framework. To reiterate, the authorisation code grant involves an exchange of something called an "authorization code" between the **AS** and the **Client**. This "authorization code" was initially provided to the **RO** from the **AS**, effectively creating an authentication triangle that automatically creates a level of confidence between all parties involved. Figure 3.3 describes again a more bespoke representation of the "Authorization Code Grant" flow. A good way of looking at the diagram is an anti clock wise data flow between the RO, Client and the AS.

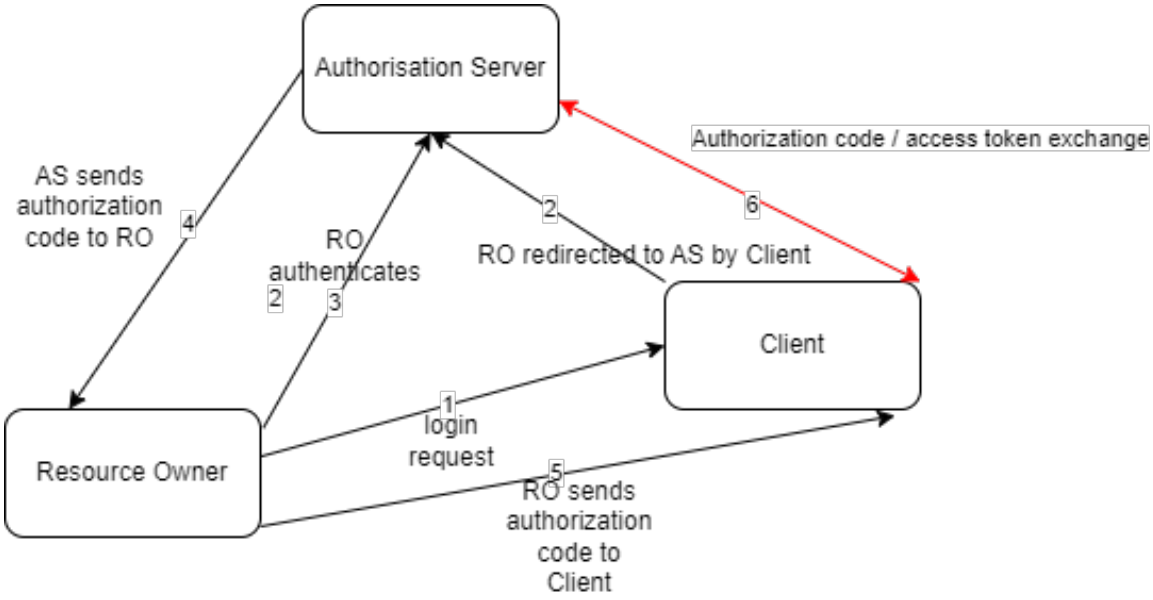

Figure 3.3: Authorization Code Grant

3.2 Proof Key for Code Exchange

The PKCE (Proof Key for Code Exchange) grant type is a specification defined after some attacks were discovered in the "Authorization Code Grant" flow, namely the "Authorization Code Interception Attack" (Preibisch, 2020). This will be described in more detail in a later section in this document. However, to understand the PKCE grant type, it is important to understand a distinction between public and confidential clients.

3.2.1 Public Clients

Public clients are those that operate in environments where the confidentiality of credentials cannot be securely maintained. This includes applications running on the user's device, such as mobile and desktop applications, as well as client-side web applications like Single Page Applications. In these environments, the client secret cannot be securely stored as the platform does not offer a secure method to do so, making it accessible to the user or potentially malicious entities (Auth0, 2024b).

3.2.2 Confidential Clients

Confidential clients are capable of securely storing credentials, including the client secret, making them suitable for server-side applications. These clients run in controlled environments where unauthorised access to client credentials is preventable. Examples include web applications with a back-end server, enterprise applications running on secure internal networks, and other applications where the client secret can be securely stored and used for authentication purposes. Confidential clients can use a wider range of OAuth grant types, such as the "Authorization Code Flow", "Client Credentials Flow", or "Resource Owner Password Credentials Flow", since they can securely authenticate themselves with the authorisation server using their stored credentials (Auth0, 2024b).

PKCE mitigates the risks associated with the interception and unauthorised exchange of authorization codes in public clients. It achieves this by introducing a dynamic secret creation mechanism that enhances security through a series of specific steps. This process is outlined below, accompanied by a diagram for a clearer understanding.

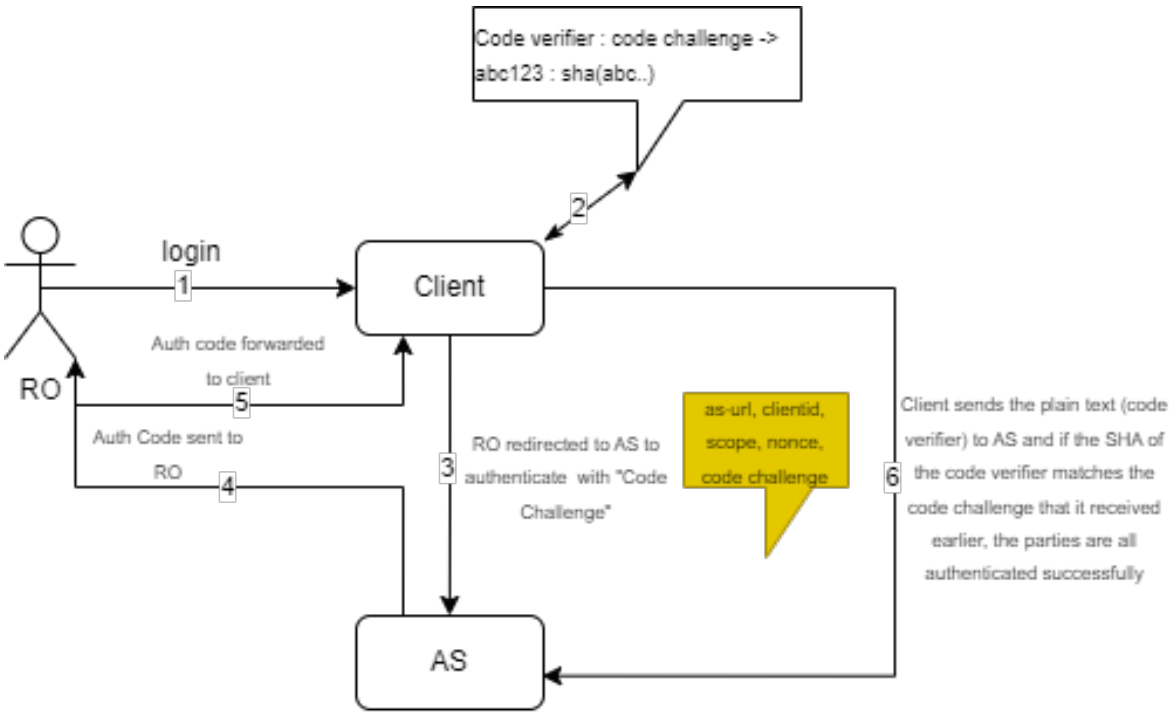

Figure 3.4: The Proof Key for Code Exchange Flow in OAuth2.0

- **1**: The RO tries to log in to the Client.

- **1a**: The client generates 2 things:
  - A code verifier: Think of this as a plain-text passphrase that must be kept entirely secret.
  - A code challenge: This a derived value from the code verifier using any kind of known transformation (known between AS and Client). This is generally a strong hashing scheme like SHA256.
- **2**: The client redirects the RO to the AS to authenticate. The client sends the RO the following things:
  - The URL of the AS where authentication can be done.
  - The client-id of the Client for Client identification.
  - The scope of the requests. For example: email, profile, contacts.
  - The code challenge — The transformed value from the code verifier.
- **3**: The RO Authenticates to the AS using the above items and personal authentication information.
- **4**: When authentication is successful, the AS sends the RO an *Authorization_Code* and a redirect back to the client.
- **5**: The client receives *Authorization_Code* from RO and sends it to AS with **the code verifier**.
- **6**: The AS transforms the code verifier and matches it with the code challenge. If this is successful, all parties have been authorised and the PKCE flow is complete.

To conclude, the key differences between the PKCE extension and the standard Authorization Code Flow lie in their treatment of client authentication and security mechanisms designed to protect against specific attack vectors. PKCE was introduced specifically to safeguard public clients from the Authorization Code Interception Attack, a vulnerability that public clients, by their nature, are particularly susceptible to due to their inability to securely store client secrets (Okta, 2022). An important consideration when implementing security measures like PKCE is finding the right balance between enhancing security and maintaining system efficiency. PKCE significantly elevates security in the client authentication process, particularly beneficial in potentially hostile environments such as mobile and front-end application where securely storing credentials is challenging.

However, efficiency is a critical factor, especially for applications expected to handle a high volume of authentication requests (An application like our implementation of ERP HCM will in production have to expect a high volume of authentication requests). Implementing PKCE in such an environment would introduce a considerable overhead. This is because of PKCE's requirement to generate a unique code verifier and challenge for each user login attempt, which could potentially strain the system resources if there are numerous concurrent authentication requests.

3.3 Implicit Flow

The Implicit Grant Type in OAuth2.0 was originally designed for client-side applications, where the application interacts directly with the user's browser without the need for a server-side component to handle the authentication. This grant type simplifies the flow by allowing

the application to receive an access token directly after the user's authorisation, without an intermediate code exchange step (DocuSign, 2024). In most cases, this can be seen as a clear security flaw, since the access token is simply provided to the end-user without assuming a hostile environment.

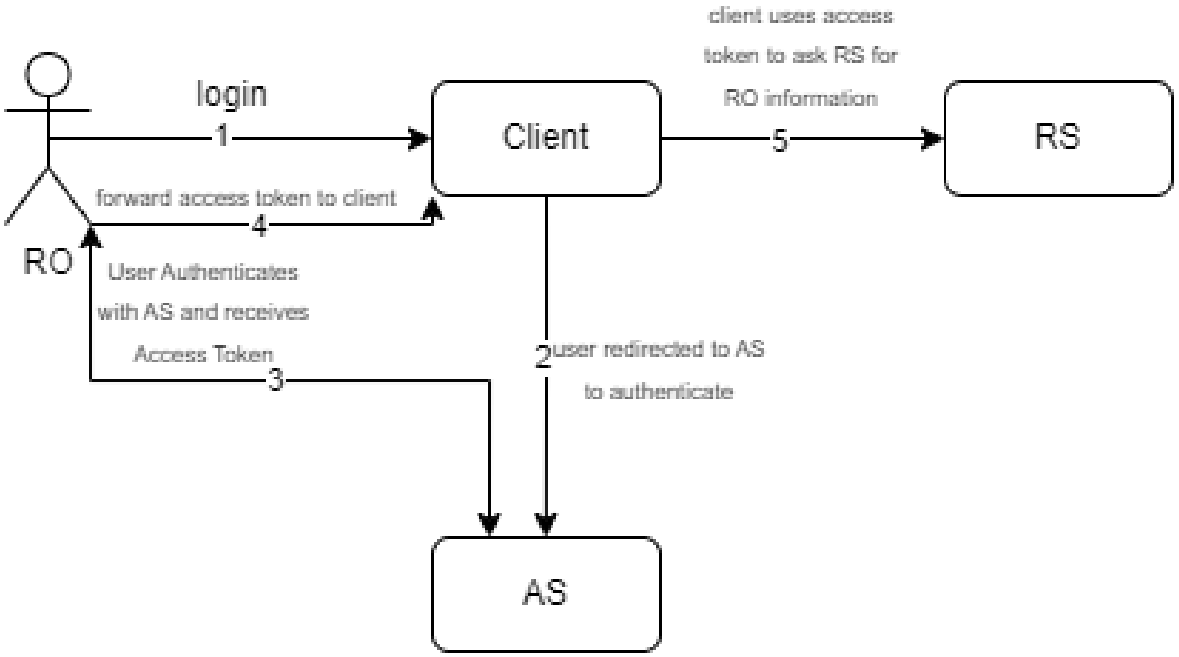

Figure 3.5: The Implicit Grant Type flow in OAuth2.0

The Implicit Grant Type was considered a good option for applications unable to securely store client secrets (client-id and client secret), such as single-page applications running entirely in the user's browser. However, its simplicity comes at the cost of security. Since the access token is passed directly through the browser (like the authorization code in the Authorization Code Grant Type), it is more susceptible to being exposed or intercepted. Additionally, the Implicit Grant does not support this issuance of refresh tokens, which means short-lived access tokens must be frequently reissued through repeated user authorisations (Parecki, 2018) (mic, 2023) (Auth0, 2024a).

Compared to other grant types, like the Authorization Code Grant, which requires the application to exchange an authorization code for an access token via a server-side request, the Implicit Grant Type is less secure. This realisation has led to a shift in recommendations, with many now advocating for the use of the Authorization Code Grant Type with PKCE for applications previously suited to the Implicit Grant. The reason for this is clear, the lack of trust within the environments of the "client" and the "end-user" which in most cases where Implicit Grant Type was suitable would be the same environments.

In summary, while the Implicit Grant Type offers simplicity and was a viable option for certain types of applications, evolving standards and a deeper understanding of web security have led to the preference for more secure protocols within OAuth2.0, especially for public clients like mobile and single-page applications.

Now that a fundamental understanding of OAuth2.0 is established, the next section of this document will describe in detail potential configurations that cause OAuth2.0 implementations to be flawed.

4. OAuth2.0 Pitfalls with Current Implementations

The widespread adoption of OAuth 2.0 and OpenID Connect in modern web applications has significantly improved the overall security and user experience in authentication and authorisation processes. However, the complexity and flexibility of these frameworks have also introduced a new set of challenges and potential vulnerabilities that developers must address. This section provides an in-depth analysis of the common pitfalls, misconfigurations, and attack vectors associated with the implementation of OAuth 2.0 and OpenID Connect.

4.1 Facebook OAuth2.0 Framework Vulnerability

The Facebook OAuth Framework vulnerability, 2019 (Baikar, 2020), exposes a critical flaw in Facebook's OAuth 2.0 implementation that allowed attackers to hijack the OAuth flow and steal access tokens. This vulnerability is particularly concerning as it could enable attackers to gain unauthorised access to user accounts across various clients that rely on Facebook's OAuth 2.0 for authentication and authorisation. This can be considered a "supply-chain attack".

In the case of the Facebook OAuth Framework vulnerability, the flaw resided in the "/connect/ping" endpoint, which was responsible for issuing user access tokens, similar to the "/token" endpoint that will be present in the code section of this document. This endpoint used a redirect URI, "XD_Arbiter", which was whitelisted by default for all clients. The vulnerability was exploited through a proxy resource called "page_proxy", which contained insecure code that allowed the leakage of access tokens to any origin via the postMessage() API.

The attacker could manipulate the "XD_Arbiter" URL by appending the "page_proxy" resource, effectively bypassing Facebook's security measures. This bypass was possible due to several factors, including the absence of the `X-Frame-Options` header, which made the flow completely frameable.

> *The `X-Frame-Options` header is a security measure that is used to prevent a web-page from being embedded or framed within another web-page. This header is an important defence against clickjacking attacks, where an attacker tricks a user into interacting with a hidden or disguised element on a web-page, potentially leading to unintended actions or disclosure of sensitive information (Buchanan et al., 2018).*
>
> *By making the OAuth flow "completely frameable", the attacker could easily trick users into entering their credentials or granting permissions to the malicious application without realising that they were not interacting with the genuine Facebook OAuth pages. This framing capability played a crucial role in the exploit, as it allowed the attacker to seamlessly integrate the vulnerable OAuth flow into their malicious website, increasing the likelihood of a successful attack.*

By exploiting this vulnerability, attackers could steal access tokens without the victim's knowledge. These stolen tokens could then be used to compromise user accounts across a

wide range of third-party applications that relied on Facebook's OAuth 2.0 implementation, such as Instagram, Netflix, Spotify, and others.

The discovery and mitigation of the Facebook OAuth Framework vulnerability highlight the importance of thoroughly testing and auditing OAuth 2.0 implementations to identify and address potential security flaws. It also underscores the need for developers to adhere to best practices and follow security guidelines when implementing OAuth 2.0 to ensure the protection of user data and prevent unauthorised access. Furthermore, Developers should also implement strict whitelist-based validation for redirect URIs and avoid using wildcard or overly permissive patterns. Additionally, enabling the `X-Frame-Options` header with the appropriate value (e.g., "DENY" or "SAMEORIGIN") can help prevent clickjacking attacks by restricting the framing of OAuth pages. By adhering to these guidelines and maintaining a security-focused approach, developers can build more robust and secure OAuth 2.0 applications that protect user data and prevent unauthorised access.

4.2 Improper Implementation of Implicit Grant Type

Improper implementation of the implicit grant type is another prevalent pitfall in OAuth 2.0 deployments. The OAuth 2.0 implicit grant flow was initially designed to cater to the needs of public clients as defined in the earlier section, where the entire client application runs within the user's browser. However, the implicit flow has inherent security risks, as it exposes access tokens in the browser's URL, making them vulnerable to interception and unauthorised access by malicious actors (Parecki, 2018). Attackers can exploit these exposed tokens to gain unauthorised access to user resources and perform actions on their behalf. To mitigate these risks, the OAuth 2.0 Security Best Current Practice (OAuth 2.0 Security BCP) recommends using the $authorization\_code\_flow$ in combination with PKCE for single page applications, providing a more secure alternative to the implicit flow (T. Lodderstedt and Authlete, 2024).

4.3 Flawed Scope Validation

Flawed scope validation is another critical area of concern in OAuth 2.0 implementations. Scopes are used to define the permissions and access rights that are granted to the client application. They play a crucial role in controlling the level of access that an application has to the RS. Inadequate validation of scopes during the token issuance process can lead to unauthorised access and potential data breaches. The demonstration of how access validation should be done at the RS level is shown in **section 6.6**. For example, if a client application requests a scope that the user has not explicitly consented to, and the AS fails to properly validate the requested scope, it may erroneously issue an $access\_token$ with elevated privileges. This flaw can allow the client application to access sensitive user data or perform actions that the user has not authorised. To prevent such issues, it is essential to implement robust scope validation mechanisms, ensuring that the authorisation server only grants scopes that have been explicitly approved by the user and adhere to the principle of least privilege.

> *The principle of least privilege is a fundamental security concept that states that a user, application, or system should be granted the minimum level of access rights and permissions necessary to perform its intended functions. This approach minimises the potential damage that can be caused by accidental or malicious*

> *misuse of elevated privileges, reducing the attack surface and limiting the impact of security breaches. By adhering to the principle of least privilege, organisations can enhance their overall security posture and protect sensitive data and resources from unauthorised access (Ma et al., 2011).*

4.4 Vulnerability to Session Fixation Attacks

In a session fixation attack, an attacker establishes a session with a web application and obtains a valid session identifier. The attacker then tricks the victim into using this session identifier, typically by crafting a malicious link or embedding the session identifier in a web page. If the victim authenticates with the web application using the attacker-provided session identifier, the attacker can hijack the session and gain unauthorised access to the victim's account and data (Sosa, 2019).

In the context of the ERP HCM solution, since the application is Java based a unique "session id" key called `JSSESSIONID` is created by Spring Boot for the browser, if an attacker can obtain a valid `JSSESSIONID`, they can potentially use it to fixate the user's session and hijack it once the user authenticates with the AS.

This vulnerability is particularly concerning because the `JSSESSIONID` is often stored in the browser's local storage or session storage, which can be vulnerable to cross-site scripting attacks. If an attacker can inject malicious scripts into the clients' web page, they could potentially steal the `JSSESSIONID` and use it to fixate and hijack the user's session. Although this is not a vulnerability that is specifically caused by OAuth2.0 or OpenID Connect the vulnerability itself can jeopardise the entire integrity of the application such that the whole point of implementing robust authentication and authorisation mechanisms like OAuth2.0 and OpenID Connect is voided.

To mitigate the risk of session fixation attacks in applications, it is essential to implement secure session management practices. This includes generating new session identifiers after authentication, validating the authenticity and integrity of the `JSSESSIONID`, and using secure storage mechanisms, such as HTTP-only cookies or secure browser storage.

Additionally, implementing strong XSS protection measures, such as input validation and output encoding, can help prevent attackers from injecting malicious scripts and stealing the `JSSESSIONID`. In the case of the ERP HCM solution, no parameters where XSS payloads can be placed are available on the frontend.

4.5 Token Leakage and Impersonation Risks

If an attacker manages to obtain a valid *access_token* or *id_token*, they can exploit this vulnerability to gain unauthorised access to the user's data or perform actions on their behalf without the user's knowledge or consent. This risk is particularly concerning because these tokens are often stored in the browser's local storage or session storage. This is obviously only the case in the OAuth2.0 Implicit Grant Type. This further demonstrates 2 things, the need for PKCE and the need for using HTTP only cookies, such that these tokens can not be leaked even after an XSS attack.

The paper by Mitchell et al. (2019) suggests that the risk of token leakage is inherent to

the design of OAuth 2.0 and OpenID Connect protocols, as they heavily rely on the use of bearer tokens. The authors emphasise the need for proper security measures to protect these tokens from being leaked or stolen. Again, this emphasises the need for abiding by security best practices when designing frontend applications. This security best practices include protection against XSS and using HTTP only cookies.

4.6 Authorization Code Interception Attack

The Authorization Code Interception Attack is a security vulnerability in OAuth 2.0 that can occur when the *authorization_code*, which is exchanged for an access token, is intercepted by an attacker during the authorisation process. This attack is particularly relevant to the $Authorization\_Code\_Grant\_flow$, one of the most commonly used OAuth 2.0 flows.

In this attack scenario, the attacker intercepts the *authorization_code* as it is being sent from the AS to the client application via the user's browser. The attacker can then use the stolen *authorization_code* to request an access token from the AS, effectively gaining unauthorised access to the user's protected resources (OWASP, 2023).

To prevent this type of attack, OAuth2.0 recommends using the PKCE extension. PKCE introduces an additional layer of security (as mentioned in **section 3.2** of this document) by transferring the responsibility of singularising the RO. Furthermore, it is crucial to ensure that all communication between the client application, AS, and the RS occurs over secure, encrypted channels using HTTPS. This helps protect the *authorization_code* and other sensitive data from being intercepted by attackers during transit.

4.7 Cross Site Request Forgery

Cross-Site Request Forgery (CSRF) is a type of attack that exploits the established session between a user's browser and the client. After the user successfully authenticates with an AS, the client creates a local pair containing information about the user account and the AS-managed account. This pairing is facilitated by a unique, randomly generated value called the "state" parameter, which is produced by the AS (Argyriou et al., 2017).

In a CSRF attack, both the user's browser and the client are targeted by the attacker. The attack is initiated when the attacker forces the user's browser to send a request to the client, with the goal of making the client execute actions without the user's knowledge or consent. This can occur when the user is lured into visiting a malicious site or by following a link on the targeted client site, in this case the ERP HCM web application.

The attacker's browser then sends a request to the client, containing a new binding value that is valid for correlating the attacker's binding with their own AS. If this request passes the client's validation process, the attacker gains the ability to log in to the victim's account on the RP.

The vulnerability that enables CSRF attacks often stems from defective validation of the "state" parameter produced by the AS. This error can be present in both the client and the AS. Many client implementations fail to consider the "state" value altogether, relying solely on its presence to ensure the validity of the action. On the other hand, some AS use easily guessable, fixed-length strings to represent the "state" value, making it predictable and

susceptible to attack. In the case of the ERP HCM solution, the CSRF token creation and validation is done by Spring Boot and shown in the code section of this document (Argyriou et al., 2017).

To mitigate the risk of CSRF attacks, it is crucial to use hash functions that produce random-length values for the "state" parameter, making it difficult for attackers to guess or predict. Additionally, clients must always verify the correctness of the "state" value received from the AS, rather than simply checking for its presence. These security measures are more important and should be given more attention when using frameworks that do not come with support for OAuth2.0 and OpenID Connect. In the case of Spring Boot, these mechanisms are pre-existing.

4.8 Absence of NONCE in OAuth2.0 Implementations

The use of a nonce (number used once) is a crucial security measure in OAuth 2.0 and OpenID Connect to prevent replay attacks. A nonce is a unique, randomly generated string that is included in the authentication request and is used to associate a client session with an id_token, ensuring the authenticity and freshness of the token (OpenID Foundation, 2014a).

Replay attacks occur when an attacker intercepts a valid request or response and maliciously repeats or delays it to gain unauthorised access or impersonate a legitimate user (OWASP, 2008). In the context of OAuth 2.0 and OpenID Connect, an attacker could capture a valid id_token or access token and attempt to reuse it in subsequent requests to gain access to protected resources (Jones et al., 2015a).

The use of a nonce is particularly important in the Implicit Flow of OAuth2.0 and OpenID Connect, where the *id_token* is returned directly to the client application via the browser's URL fragment. In these flows, the nonce helps to ensure that the *id_token* is not intercepted and reused by an attacker (OpenID Foundation, 2014b).

To effectively prevent replay attacks, it is essential to generate a unique nonce for each authentication request and to securely store the nonce value on the client-side until it can be verified against the received *id_token*. The nonce should be sufficiently random and long enough to make it difficult for attackers to guess or predict (T. Lodderstedt and Authlete, 2024). In the case of our implementation, the nonce is handled entirely by Spring Boot Java.

5. Project Description and Objectives

As previously articulated, the primary objective of this document is to showcase the use of Spring Boot in the development of a sophisticated authentication system, rather than delivering a comprehensive ERP HCM solution. Consequently, the focus is squarely on the underlying technical framework and the security aspects it enhances, rather than the encompassing business logic, product functionalities, or the user interface and user experience (UI/UX) design of the application. This section methodically details, on a feature-by-feature bases, which components of the ERP HCP solution are incorporated within this project and elucidates their significance in demonstrating the capabilities of the authentication system underpinned by Spring Boot.

The software requirements for this project are divided into three applications; the front-end application, the client application, the authorisation server, and finally the resource server. The goal of the client application is to demonstrate the back-end workflows of a secure user onboarding process, which involves allowing the user to choose their preferred authentication mechanism.

5.1 The Client Application

As previously outlined in this document, the client application serves as a crucial component within the OAuth 2.0 framework. It functions by authenticating both itself and the user to the authorisation server, representing the interface through which end users engage with the entirety of the project. This interaction model underscores the client application's pivotal role in facilitating secure access and ensuring a streamlined user experience in accordance with OAuth 2.0 standards.

5.2 User Authentication Flow

5.2.1 User Authentication Initiation

The user authentication flow constitutes a key feature of the client application, designed to manage user authentication requests seamlessly. This process initiates when the user accesses the application and is presented with the option to select and authorisation provider for login purposes. Upon making their selection, the user is then redirected to the login page of the chosen authorisation provider to authenticate themselves. This choice can be seen in Figure 5.1.

Figures 5.2 and 5.3 provide a detailed visual representation of the underlying mechanisms activated once a user selects their preferred authorisation provider.

5.2.2 User Registration

After the choice of which authorisation server to authenticate with is made by the user, the client application receives the user information. In the case of this application, a conscious

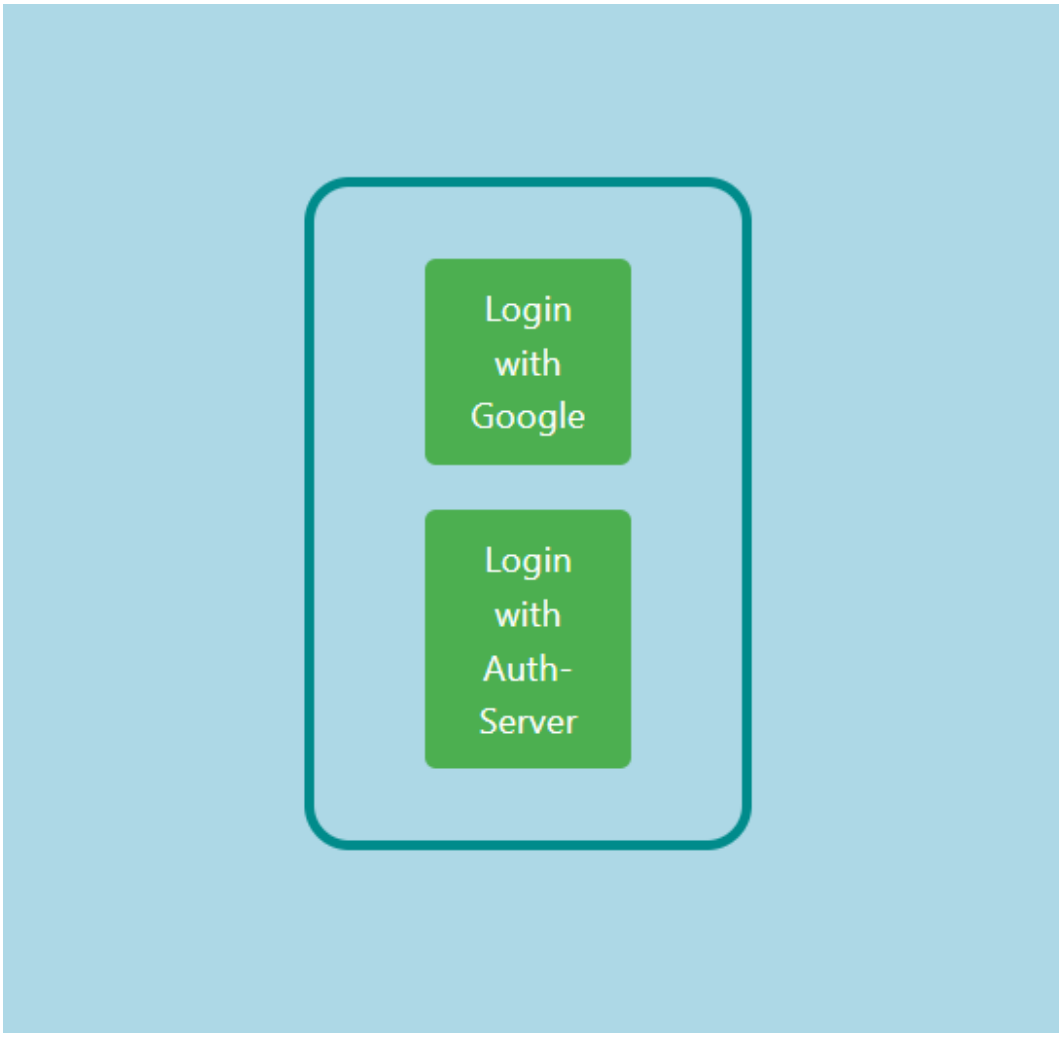

Figure 5.1: User given an option to chose which authorisation server they would like to log in with.

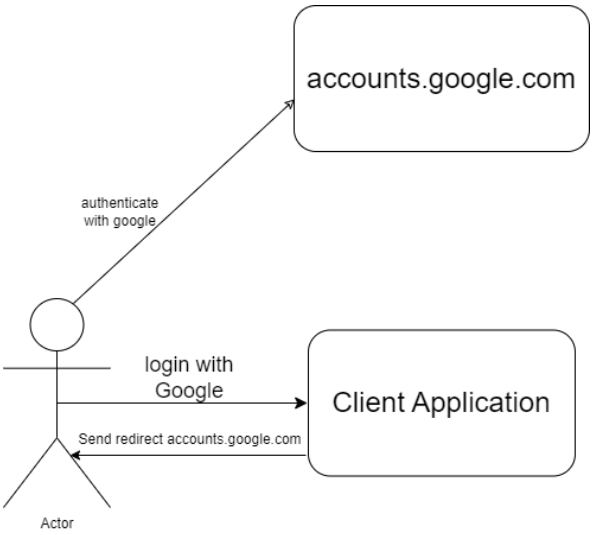

Figure 5.2: When user clicks 'Login with Google'.

security decision has been made to move the user information endpoint from the authorisation server to the resource server. The rationale behind this decision is to further double down on the idea of 'separation of concerns' in this microservices architecture. This configuration

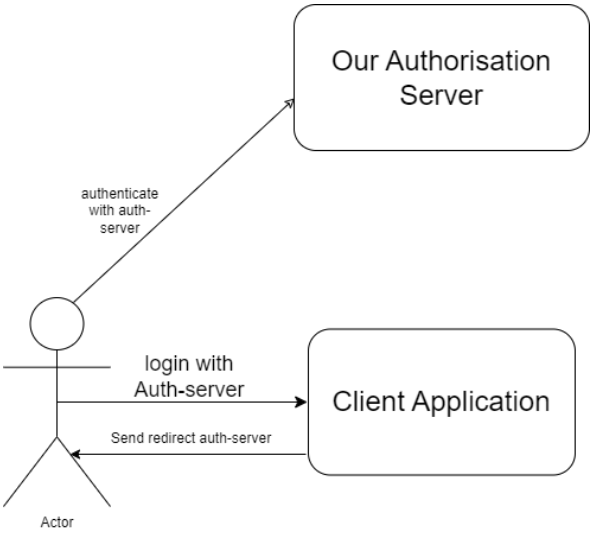

Figure 5.3: When user clicks 'Login with Auth-Server'.

allows for two things:

- **Flexible storage of user information** — If user information is stored in the resource server, other servers can use this information more seamlessly provided they have an access token. This allows for multiple authorisation servers to be set up connected to a single resource server for some applications in the context of a load balancing system where multiple active servers are up and running to balance out the load of the system (Aruba Networks, 2023).
- **Isolation** — Isolation is a very important concept of security, which involves segregating computing processes such that each element is only responsible for one task. In this case, an authorisation server should only be responsible for authenticating a user and any other tasks should be delegated to another system. As such, the resource server.

Upon receipt of user information from the resource server, the client application initiates a verification process to ascertain whether the user is already registered within the system. If the user's credentials are recognised, indicating prior registration, they are seamlessly navigated to their dashboard. Conversely, if this is the user's fist authentication with the system, they are redirected through an onboarding process.

5.2.3 User Onboarding Flow

This section describes the case where the user authenticating is indeed a new user and the application decides to redirect the user to the user onboarding process. The onboarding flow incorporates Role-Based Access Control (RBAC) as a crucial component. The primary objective of integrating RBAC in this context is to ensure that once users are onboarded, they are prevented from re-initiating the onboarding process. Additionally, RBAC plays a vital role in controlling user access to various stages of the onboarding process. For instance, users in the initial phase of onboarding are restricted from accessing or making requests to advance stages, such as the third stage. This might appear overly cautious, but it serves as a demonstrative example of how RBAC can be effectively implemented in a system.

Figure 5.4 Shows the decision-making process taken by the application when deciding which endpoint to direct the user to.

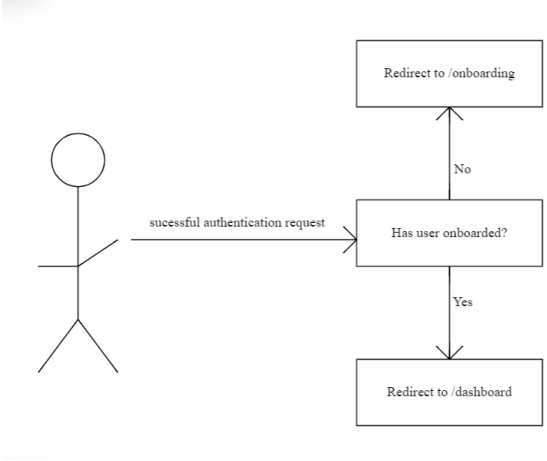

Figure 5.4: User request to Spring Boot Application after Authenticating with Google

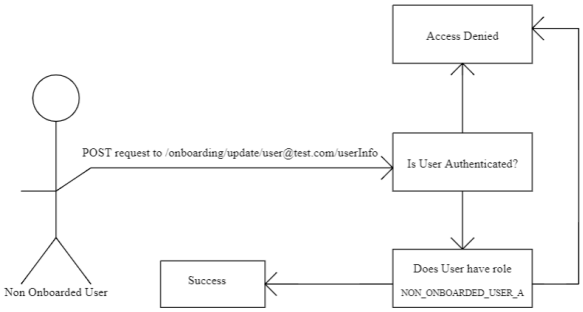

Figure 5.5: First onboarding phase using RBAC

In **Figure 5.5** notice the reference to "NON_ONBOARDED_USER_A" as the checked role when the user tries to update their user information. In this part of the user onboarding process, the user updates information provided by the authorisation server about them. For example, first name and last name.

It is important to note that any user with the "NON_ONBOARDED_USER_*" will not have any other roles in the system, since that would defeat the purpose of the RBAC. This also means these non onboarded roles are exclusive for certain endpoints and can only be accessed by these roles and these roles can not access any other endpoint.

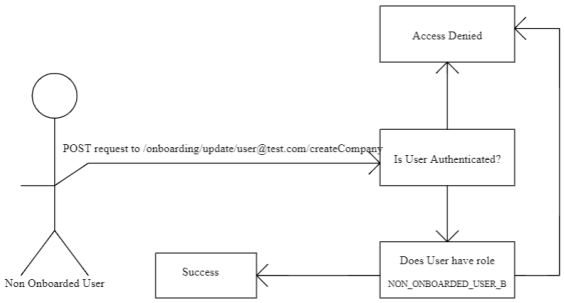

Figure 5.6: First onboarding phase using RBAC

Figure 5.6 shows another phase of the onboarding process where the user needs to have another role. Only users with the role "NON_ONBOARDED_USER_B" can access the endpoint, any other requests will be denied.

The onboarding process is made up of more phases, including the "join company" and the "complete onboarding" phases. The data flow diagrams for those are shown below.

5.2.4 Storage of User Details

In the context of this project, the process of user authentication is facilitated through a chosen authentication provider. Upon successful authentication, the client application will request required information from what is known as a "userinfo" endpoint. This endpoint usually sits either on the authorisation server or the resource server.

For the purposes of this project, the scope must be meticulously specified to encompass the acquisition of the following user information:

- Email
- First name
- Last name
- Profile Picture: A URL linking to an image of the user, hosted on perhaps an external resource server.

5.2.5 User Login

In scenarios where a user is revisiting the system after initial registration, an authentication request is indicative of their intention to log in. In this case, the user authentication service verifies the following things:

- The user has successfully completed all stages of the onboarding process.
- The user account is enabled.
- The user account is not locked.

Upon satisfying all these conditions, the user is automatically redirected to the ERP HCM dashboard.

5.3 Roles and Authorities

Access control alone cannot comprehensively ensure the security of a system; it necessitates complementing with robust auditing mechanisms. Effective auditing, in turn, relies heavily on the establishment of strong authentication protocols. In response to this requirement, a method of RBAC called AARBAC (Authentication and Auditing Enabled Role Based Access Control) is introduced, (Kashyap, 2012) which seamlessly integrates RBAC with authentication and auditing functionalities.

Following are the features of AARBAC that are important to this project and how they are resolved.

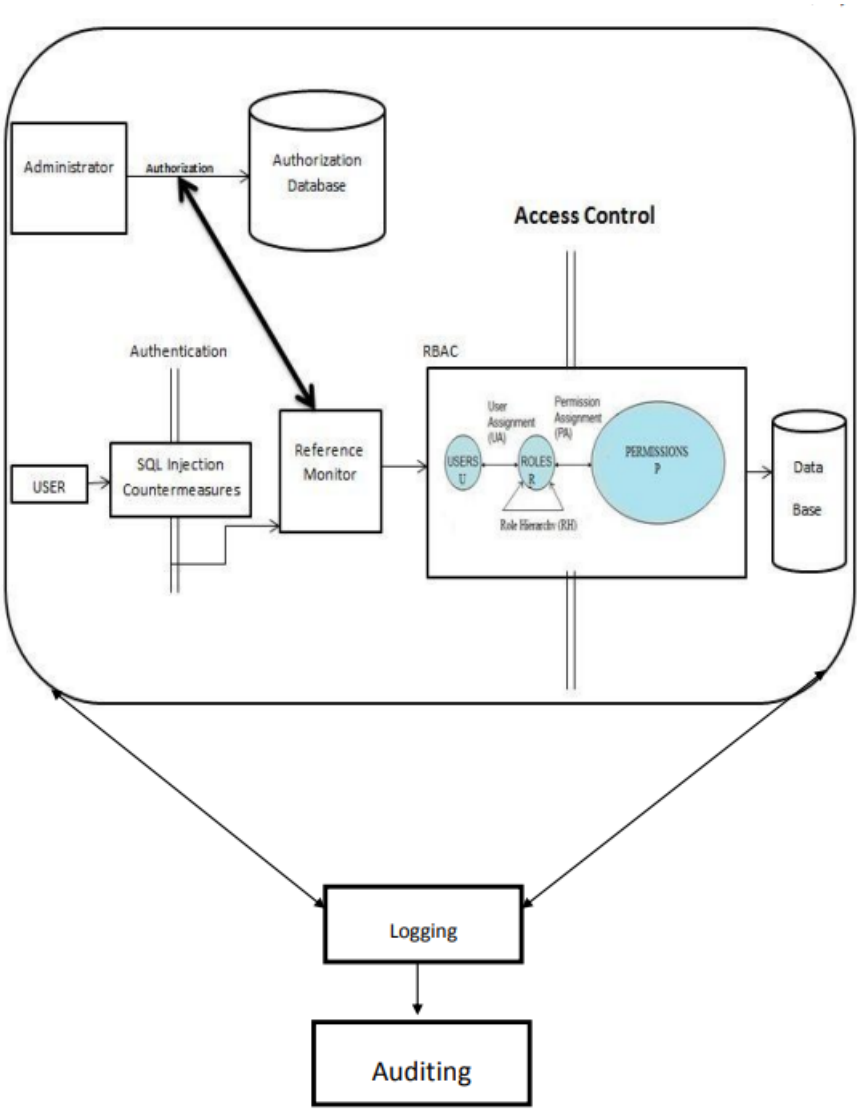

Figure 5.7: AARBAC dataflow as shown by Kashyap (2012)

- Authentication — This is entirely delegated to Authorisation servers.
- Auditing — This is done in the client application as a proof of concept.
- SQL Injection prevention — This is also handled by the authorisation server. In the implementation of the local authorisation server, SQL injection prevention is handled by Spring Data JPA, since no manual SQL queries are written (Philippe Sevestre, 2021).

A simpler version of what is shown in the Figure 5.7 is implemented within the client application using Roles and Authorities as programmatically distinct things. The reason for the distinction is to allow more flexibility in RBAC. Roles and authorities in the system are two completely different tables that share a Many-to-Many relationship. This means that one admin can have completely different authorities and multiple of them, and one authority can be shared by multiple admins. This system naturally allows for more flexibility in the future as the applications of the ERP HCM solution increase and roles and authorities are defined as a spectrum rather than distinct and rigid types. Following are all the roles and their authorities that are implemented in the client application.

5.3.1 Employee

Employees have the following authorities:

- Account creation using Google SSO (Single Sign-On).
- Login via Google credentials.
- Automatic retrieval of full name, email, and profile photo from Google's resource server.
- Updating personal details like first name, last name, address, and profile photo.

5.3.2 Manager

Managers can perform employee tasks and additionally:

- Approve holiday requests for team members.
- Upload payslips for team members.
- Add or remove employees in their team.

This requirement although not implemented in the application, an explanation of how this can easily be done is described in the code review section of this document.

5.3.3 Admin

The admin, who initiates the company or is assigned the role, has all the above authorities plus:

- Edit personal details of all company members.
- Add or remove any user in the company.
- Create new roles and authorities.
- Assign roles and authorities to users.

In a more robust ERP HCM solution, these roles and authorities would be more elaborate and the application would also consist of many more roles within the company that demonstrate the capabilities of different teams.

6. Project Structure and Code Review

This section of the document delves into the software engineering principles and practices adopted in the development of the ERP HCM solution. It covers the structure of the project on GitLab, the challenges encountered and how they were addressed, and the implementation of the Spring MVC design pattern. Special focus is given to the security configuration within Spring Boot, detailing the roles of controllers, service layers, and repositories. It also explores the intricacies of entity management and the functionality of various handlers. A key element of this section is the demonstration of how Spring Security can be used to implement application layer security.

TDD (Test Driven Development), renowned for its effectiveness in ensuring code quality and reducing bugs, was initially planned for this project. However, due to time constraints, JUnit5 test cases for REST API controllers were not implemented in time, leading to a reliance on manual testing using PostMan as described in the Introduction of this document. Recognising the value of integrating bot automated and manual testing methods, the aim is to establish a robust TDD setup in future phases of the project, enhancing development efficiency and reliability (George and Williams, 2004).

To enable the Spring Boot client's communication with Google's **AS**, a configuration in the Google API Console needs to be made (goo). Here, two critical elements: *client_id* & *client_id*, and *callback_url* must be considered. The *client_id* & *client_id* are key to identifying and securing the client application. The *callback_url*, which is necessary for redirecting users after authentication. These components are crucial for authenticating the Spring Boot application and establishing OAuth2.0 connection with Google, ensuring that the authorisation process is both reliable and secure. Below is a series of images demonstrating how this setup can be done from the side of the Google API Console.

The figures above **6.1, 6.2, 6.3** do a clear demonstration of how the two tokens can be established from the Google API console. **Figure 6.3** shows the call back URL. The callback URL is simply the endpoint on the Spring Boot application that will accept the *authorization_code* form of the user. The reason this is important is so that Google is able to automatically redirect the user back to the application once authentication and/or authorisation is done (Hardt, 2012).

Furthermore, since our ERP HCM solution aims to allow the delegation of authentication to not only Google but also to another locally hosted authorisation server, the configuration for this is more manual and bespoke. All the configuration that is shown in Google's case is done in the application properties of the authorisation server implementation in Spring Boot. The application properties for the authorisation server that demonstrate the *client_id*, *client_secret*, *available_scopes*, *available_grant_types*, *redirect_uris* and *access_token_duration* is shown in **section 6.1.2**

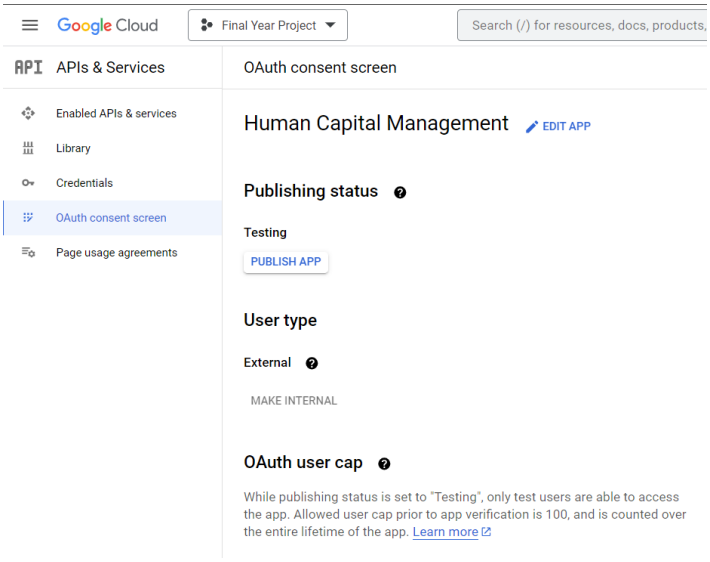

Figure 6.1: Google API Dashboard

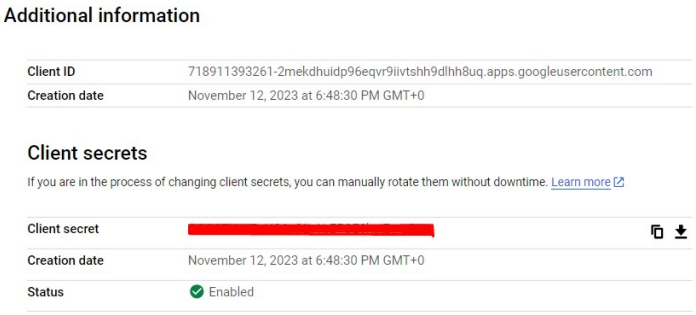

Figure 6.2: Access to client_id and client_secret from the dashboard

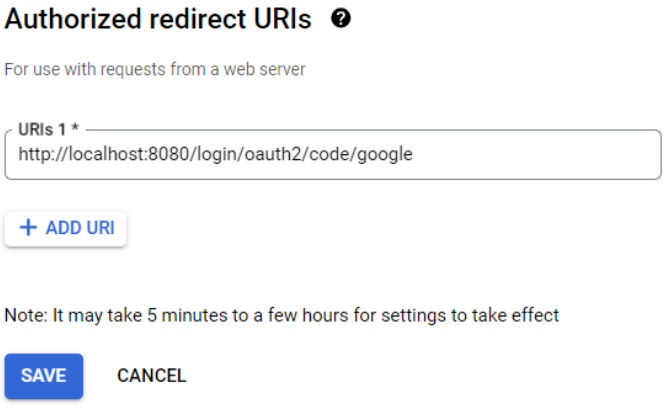

Figure 6.3: Access to client_id and client_secret from the dashboard

6.1 Configurations

As referenced in the earlier section, Spring Boot comes with a file called application.yml. This is a top-level configuration file specifically used to set up the main configuration for the application. Generally, the application.yml file will hold global variables that need to be accessed everywhere in the application. In the case of the ERP HCM solution, the original application.yml file refers to application-dev.yml as the active configuration. This is the case for the Client, the Authorisation Server as well as the Resource Server. This is a good software engineering practice, as it creates a blatant isolation between development and production environments. Some of the configurations that are put in the application-dev.yml should never be used in production. For example, the use of database credentials directly from environment variables isn't a good practice (Truth, 2023).

> *While using environment variables for database credentials in a development application is somewhat acceptable, it is far from optimal. A superior approach involves utilising secret management services like Microsoft Azure Key Vault, offering a secure repository for keys, tokens, and certificates. This method elevates security, especially in contexts where codebases might be exposed, such as open-source projects on GitLab. Opting for environment variables, albeit a conscious security compromise, is markedly better than embedding plain-text secrets directly in code, which could significantly amplify risks if unauthorised individuals access the codebase.*

6.1.1 application-dev.yml for the Client

```yaml
server:
  port: 8080

spring:
  security:
    oauth2:
      client:
        registration:
          google:
            clientId: ${CLIENT_ID}
            clientSecret: ${CLIENT_SECRET}
            redirectUri: "{baseUrl}/login/oauth2/code/google"
            scope:
              - profile
              - openid
              - email
          auth-server:
            client-id: "erp-api"
            client-secret: ${AUTH_SERVER_SECRET}
            redirect-uri: "{baseUrl}/login/oauth2/code/auth-server"
            authorization-grant-type: "authorization_code"
            client-authentication-method: "client_secret_post"
            scope:
              - openid
              - profile
```

```
                        - user.read
                    provider:
                      google:
                        authorizationUri:
                            https://accounts.google.com/o/oauth2/auth
                        tokenUri: https://oauth2.googleapis.com/token
                        userInfoUri: https://www.googleapis.com/oauth2/v3/userinfo
                        userNameAttribute: sub
                        jwkSetUri: https://www.googleapis.com/oauth2/v3/certs
                      auth-server:
                        issuer-uri: http://127.0.0.1:8081
                        user-info-uri: http://localhost:8082/userinfo


                datasource:
                  url: jdbc:mysql://localhost:3306/human_capital_management_dev
                  username: root
                  password: root
                  driver-class-name: com.mysql.cj.jdbc.Driver
                jpa:
                  hibernate:
                    ddl-auto: create
              logging:
                level:
                  org:
                    springframework:
                      web: DEBUG
                      security: DEBUG
```

The application-dev.yml establishes the necessary variables needed for the OAuth2.0 authentication with Google. It includes 'client_id' and 'client_secret'. These are sent to the **AS** when conducting the authorisation grant request. The the configuration also implements the 'redirection_uri' variable to let Spring Security know which endpoint to expect authorisation codes to. The application-dev.yml file also provides the 'scope' parameter which is used to communicate to the **AS** the resources needed by **C**. It can be seen clearly the fact that configuration for both Google and "auth-server" is present in the application-dev.yml file. Furthermore, the configuration for auth-server shows exactly the grant-type the client would like to use for the application which is *authorization_code_grant*.

Moreover, a key called $client - authentication - method$ is set to *client_secret_post* which essentially tells the client to authenticate with the AS using a post request with the *client_id* and the *client_secret*. This value is upto the discretion of the configuration at the server side. The decision as to which method to use should be based on the level to hostility to assume within the environment relating to the communication between the client and the AS. Since these request are done through a back-channel, and in the case where the authorisation sever instance and the client instance are both placed in a secure LAN the communication happens using a simple HTTP POST request. Below is a list of potential methods that could be used (Spring Security Team, 2023).

- *client_secret_basic* - The client uses its *client_id* and *client_secret* through the HTTP Basic authentication scheme.

- *client_secret_post* - similar to *client_secret_basic*, but the credentials are sent in the request body.
- *client_secret_jwt* - The client signs a JWT using the *client_secret* as the key.
- *private_key_jwt* - The client signs a JWT with its private key, which the server can verify with the corresponding public key.
- **none** - Indicates that no authentication method is used, suitable for clients that cannot keep a secret secure.

6.1.2 application-dev.yml for the Authorisation Server

```
spring:
  security:
    oauth2:
      authorizationserver:
        client:
          erp-api:
            registration:
              client-id: ${AUTH_SERVER_CLIENT_ID}
              client-secret: ${AUTH_SERVER_SECRET}
              client-authentication-methods: "client_secret_post"
              authorization-grant-types:
                - "client_credentials"
                - "authorization_code"
                - "refresh_token"
              scopes:
                - "openid"
                - "user.read"
                - "user.write"
                - "profile"
              redirect-uris:
                  "http://localhost:8080/login/oauth2/code/auth-server"
              post-logout-redirect-uris:
                - "http://localhost:8080/"
            require-authorization-consent: true
            token:
              access-token-time-to-live: 3600s

    ... database credentials, logging configurations etc.
```

The ‘application.yml‘ on the AS outlines the registration of a client for OAuth2.0, using the client’s name, ”erp-api”, as the key for its registration. Given the sensitive nature of client credentials, they are stored in environment variables. As detailed in section 6.1.1, the client’s preferred grant type is *authorization_code*, which is supported by the AS alongside *client_credentials* and $refresh\_token$ grant types. The *client_credentials* grant type focuses on the client’s authentication rather than the RO’s. In contrast, the $refresh\_token$ grant type allows the AS to refresh the access token granted to the client, necessary when an access token expires. For demonstration, an $access-token-time-to-live$ key is also presented in the application-dev.yml for the AS, demonstrating how to configure the token lifespan.

Finally, the application-dev.yml showcases the list of scopes that the AS supports. While a platform like Google may offer an extensive list of scopes, the ERP HCM solution narrows its focus to only the scopes required for demonstrating the OAuth2.0 implementation. This targeted approach ensures clarity and relevance to the specific use case at hand.

6.1.3 application-dev.yml for the Resource Server

```
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://127.0.0.1:8081

... database credentials, loggin configurations etc.
```

As described earlier in this document the role of RS is simply to hold the resources of the RO. This makes the landscape of configurations for a RS quite flexible. This flexibility means that the configurations tied to the RS are fundamentally designed to meet operational needs, with the RS's security being reliant on the validation of incoming requests. The application-dev.yml file notably includes the $issuer - uri$, linking it to the AS. This configuration is pivotal for establishing common methods to authenticate access tokens presented by clients. The reason for the $issuer-uri$ to be an $ip-address$ rather that "localhost" as used previously is described in **section 6.7.1.**

6.2 Security Configurations

A Spring Boot security configuration file has been made for this project to configure certain non-default security parameters. These configurations set the following things in the project:

- Type of authentication used
- Post authentication mechanism
- Login success handling
- Error handling

6.2.1 Security Configuration for The Client

Most of Spring Security's implementations are based on "Security Filters" which are layers on top of the application layer through which HTTP requests are 'filtered'. In light of this, defining a "Security Filter Chain" is quite important. The security filter chain method defined for the client is shown below with its explanation.

```
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http) throws
    Exception {
```

```
3                   return http
4                           .csrf((csrf) -> csrf
5                                   .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse(
6                                   .csrfTokenRequestHandler(new
                                        CsrfTokenRequestAttributeHandler())
7                           )
8                           .addFilterAfter(new CsrfCookieFilter(),
                                BasicAuthenticationFilter.class)
9                           .cors(cors ->
                                cors.configurationSource(this.myCorsConfiguration()))
10                          .oauth2Login(oauth2 -> oauth2
11                                  .tokenEndpoint(tokenEndpointConfig ->
                                        tokenEndpointConfig
12                                          .accessTokenResponseClient(new
                                                CustomAuthorizationTokenResponseClientForDebugging()))
13                                  .redirectionEndpoint(redirection -> redirection
14                                          .baseUri("/login/oauth2/code/{registrationId}"))
15                                  .userInfoEndpoint(userInfo -> userInfo
16                                          .oidcUserService(this.getOIDCUserService()))
17                                  .successHandler(new
                                        CustomAuthenticationSuccessHandler()))
18                          .authorizeHttpRequests(authorize -> authorize
19                                  .anyRequest().authenticated())
20                          .exceptionHandling(exception -> exception
21                                  .accessDeniedHandler((request, response,
                                        accessDeniedException) -> {
22                                      log.info("Access denied: " +
                                            accessDeniedException.getMessage());
23                                      response.sendRedirect("/access-denied");
24                                  }))
25                          .build();
26          }
```

The above code shows the `securityFilterChain()` method which aims to determine the chain of filtering any HTTP request must go through in this application before the request is processed by the necessary REST API endpoints/ Controllers.

6.2.1.1 CSRF Configuration

For the most part of developing the ERP HCP application CSRF protection was disabled and that could be seen in security filter chain. This, as described in an earlier section is a blatant security issue and should never be persisted in a production application. The configuration for CSRF can be seen in the security filter chain as a CSRF filter is added after the value of the CSRF token is extracted from the user's request. the class $CsrfCookieFilter$ is a public class defined below, which extends the $OncePerRequestFilter$ class defined by the spring framework. The class has a "doFilterInternal" method that runs to extract the CSRF tokens from the users request. This can be seen in the code below:

```
1                   @Override
2                    protected void doFilterInternal(HttpServletRequest request,
```

```
                    HttpServletResponse response, FilterChain filterChain)
3                       throws ServletException, IOException {
4                   CsrfToken csrfToken = (CsrfToken)
                        request.getAttribute(CsrfToken.class.getName());
5                   if (csrfToken != null) {
6                       String csrfTokenValue = null;
7                       Cookie[] cookies = request.getCookies();
8                       if (cookies != null) {
9                           for (Cookie cookie : cookies) {
10                              if ("XSRF-TOKEN".equals(cookie.getName())) {
11                                  csrfTokenValue = cookie.getValue();
12                              }
13                          }
14                      }
15                  }
16                  filterChain.doFilter(request, response);
17              }
```

Additionally, "User Services" classes have been defined as classes that will handle user operations for the endpoints. Defining Custom User Classes that implement standard Spring Boot interfaces allows for a great level of standardisation throughout the project and that philosophy itself mitigates a plethora of application layer vulnerabilities and bugs (**?**). What these Services do will be described later in this document.

6.2.1.2 CORS Configuration

Line 9 in the code defined above in **section 6.2.1** shows something called the CORS configuration. CORS (Cross-Origin Resource Sharing) is a security feature implemented in web browsers to restrict how resources on a web page can be requested from another domain outside the domain from which the first resource was served. This mechanism is crucial for preventing malicious websites from accessing resources and data from another domain without permission. The code in the `securityFilterChain()` method simply pulls in the configuration defined in the `this.myCorsConfiguration()` method. This can be seen below:

```
1           @Bean
2           CorsConfigurationSource myCorsConfiguration() {
3               CorsConfiguration configuration = new CorsConfiguration();
4               configuration.setAllowedOrigins(List.of("http://localhost:3000"));
5               configuration.addAllowedHeader("*");
6               configuration.addAllowedMethod("*");
7               configuration.setAllowCredentials(true);
8               urlBasedCorsConfigurationSource.registerCorsConfiguration("/**",
                    configuration);
9               return urlBasedCorsConfigurationSource;
10          }
```

In this code the my CORS configuration is particularly restrictive as it allows requests from only one fron-end application "http://localhost:3000" (This is the host and port running the React front-end application). Furthermore, The configuration allows all HTTP methods to be used from the allowed domain. The ERP solution simply only uses GET and POST

requests but in future more requirements may arise.

> *Misconfiguration in CORS can significantly impact the security of an OAuth 2.0 application by allowing unauthorised cross-origin requests that could lead to sensitive information exposure or account takeover. One common vulnerability arises from the improper validation of the Origin header, leading to a scenario where an attacker's website can make requests to a vulnerable application and receive sensitive data in response.*
>
> *An example exploit could involve an attacker convincing a user to visit a malicious page, which then uses JavaScript to send a request to the vulnerable application. Due to the misconfigured CORS policy, the attacker's page can read the response, potentially accessing sensitive information that should have been restricted (Polop, 2021).*

Moreover, **line 19** of the above code defined in **section 6.2.1** sets a security filter configuration to treat every endpoint in the application as an authenticated endpoint. This means that when a **RO** visits the application unauthenticated, they will be asked to authenticate. The method of authentication is defined on **line 10** (**section 6.2.1**) "oauth2Login". This configuration sets more OAuth2.0 details such as the the *redirect_uri* and the `AuthenticationSuccessHandler`. The `AuthenticationSuccessHandler` does exactly what it says on the tin- "run this on a successful authentication attempt".

6.2.1.3 Authentication Success Handler

At the moment, the handler simply redirects the user to either "/" after which the front-end decides whether the user needs to complete onboarding or reach the "dashboard".

```
1   @AllArgsConstructor
2   public class CustomAuthenticationSuccessHandler implements
        AuthenticationSuccessHandler {
3
4       @Override
5       public void onAuthenticationSuccess(HttpServletRequest request,
            HttpServletResponse response, Authentication authentication)
            throws IOException, ServletException {
6           OidcUser oidcUser = (OidcUser) authentication.getPrincipal();
7           String oidcUserEmail = oidcUser.getAttribute("email");
8
9           UserProfileDataDTO userProfileDataDTO = new UserProfileDataDTO();
10
11          userProfileDataDTO.setEmail(oidcUserEmail);
12          System.out.println(DataConversionUtil.convertToBase64(userProfileDataDTO));
13          Cookie userInfoCookie = new Cookie("inf",
                DataConversionUtil.convertToBase64(userProfileDataDTO));
14          userInfoCookie.setPath("/");
15          response.addCookie(userInfoCookie);
16          response.sendRedirect("http://localhost:3000/");
17      }
18  }
```

The method `onAuthenticationSuccess()` is run by some security filter in Spring Boot when **RO's** *authentication_code* is validated by the **AS**. The code here simply base64 encodes the user information of the successfully authenticated user and sends it back to the front-end in a cookie called "inf" (short for information). This cookie allows the front-end to identify the user for subsequent requests.

> *Since an instance of this can be seen on **line 1**, it may be a good opportunity to introduce **Project Lombok**. Lombok is a Java Library that plugs into the IDE and build tools, simplifying the code by reducing boilerplate code. It does this by automatically generating getters, setters, constructors and other common methods using Java annotations. Lombok helps in making the Java code more readable.*

6.2.1.4 Debugging Access Token Exchange between Client and AS

In **section 6.2.1**, **Line 12** introduces a custom debugging class, made specifically to understand the data flow within Spring Boot's OAuth2.0 Client implementation. By integrating a breakpoint within the method that simply invokes its superclass's functionality, it becomes possible to trace the sequence of requests exchanged between the OAuth2.0 client and the AS. This process facilitates a granular examination of how the *authorization_code* is exchanged for the *access_token*. This deliberate step-by-step analysis enhanced the comprehension of the underlying communication patterns in OAuth2.0.

6.2.1.5 Spring Boot Firewall Rule Exception - Path Traversal

```
1               @Bean
2               public HttpFirewall allowUrlEncodedSlashHttpFireWall() {
3                   StrictHttpFirewall firewall = new StrictHttpFirewall();
4                   firewall.setAllowUrlEncodedDoubleSlash(true);
5                   firewall.setAllowUrlEncodedPercent(true);
6                   return firewall;
7               }
```

During the manual testing and debugging of this application a problem was faced during the client-AS *authorization_code-access_token* exchange. By default Spring Security blocked any post requests between the client and the AS that had a "double slash" in them. Given that the request was going to "http://127.0.0.1/.." the request gets blocked. The reason for this blockage is a firewall rule implemented by spring security by default to prevent any from of path traversal attacks.

> *A path traversal attack aims to access files and directories that are stored outside the web root folder. By manipulating variables that reference files with dot-dot-slash sequence, an attacker can trick the application into accessing unauthorised directories (OWASP Foundation, 2024).*

The decision to permit URL-encoded double slashes and percent signs in requests made by the client is not inherently insecure even thought it may seem so at first glance. The reason for this is the same reason we chose to use the $authorization_code_flow$ without the need for PKCE. The fact that the communication channel between the client and the AS is entirely trusted. Furthermore, the AS does not hold any files that are accessible publicly via the URL for a path traversal attack to be possible.

6.2.2 Security Configuration for The Authorisation Server

Since one of the main goals in this project is to demonstrate isolation using the micro-services architecture promoted by OAuth2.0, the only task the AS should be responsible for is "Authentication". In this sense the name "Authorisation Server" could be considered a misnomer, a more sensible name for its task should be "Authentication Server".

Since that is the goal, and given that our AS implementation lacks advanced features like multi-factor authentication or email verification, the database schema requires only a simple user table. This table should adequately store two essential pieces of user information: the user identification value (email) and the user's secret (password).

Hence, the security configuration for the AS has the following requirements:

- Set up password encoding.
- Configure a JWT Decoder for validation and verification of JWT tokens.

Everything else with regards to the user management is taken care of by default by Spring Security.

6.2.2.1 Using Plaintext Passwords for Demonstration

The security configuration for the AS includes a method named `passwordEncoder()`, desinged to return a Spring Bean of type `PasswordEncoder`. This Bean specifies the password encoding scheme utilised for storing passwords in the database. In the present scenario, passwords are stored in plain text, with any form of encoding. Storing passwords in plan text poses a severe security risk and is considered a significant vulnerability. It is a practice that should categorically be avoided in any production environment.

For a robust security implementation, passwords must undergo a hashing process, complemented by the use of a salt, before being stored in the database. Hashing and salting are essential practices that ensure even if an unauthorised party gains access to the database, the compromised data is entirely unintelligible. This methodology drastically mitigates the risk associated with data breaches.

The decision to store passwords in plain text during the development phase of the AS was driven by the aim to streamline the development process. Given that the AS lacks a user-facing registration system, with data being manually preloaded into the database via hard-coded insert queries found in the `data.sql` file within the server's resources, incorporating hashed passwords would introduce an additional development step. This step was deemed to provide minimal incremental security value for the project's objectives at this stage.

```
1                   @Bean
2                   public PasswordEncoder passwordEncoder() {
3                       return new PasswordEncoder() {
4                           @Override
5                           public String encode(CharSequence rawPassword) {
6                               return rawPassword.toString();
7                           }
8
9                           @Override
```

```
10                      public boolean matches(CharSequence rawPassword, String
                            encodedPassword) {
11                          return rawPassword.toString().equals(encodedPassword);
12                      }
13                  };
14              }
```

6.2.2.2 Facilitating JWT Token Management

```
1               @Bean
2               public JwtDecoder jwtDecoder(JWKSource<SecurityContext>
                    jwkSource) {
3                   return
                        OAuth2AuthorizationServerConfiguration.jwtDecoder(jwkSource);
4               }
5
6
7               @Bean
8               public JWKSource<SecurityContext> jwkSource() {
9                   KeyPair keyPair = generateRsaKey();
10                  RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
11                  RSAPrivateKey privateKey = (RSAPrivateKey)
                        keyPair.getPrivate();
12                  RSAKey rsaKey = new RSAKey.Builder(publicKey)
13                          .privateKey(privateKey)
14                          .keyID(UUID.randomUUID().toString())
15                          .build();
16                  JWKSet jwkSet = new JWKSet(rsaKey);
17                  return new ImmutableJWKSet<>(jwkSet);
18              }
19
20              private static KeyPair generateRsaKey() {
21                  KeyPair keyPair;
22                  try {
23                      KeyPairGenerator keyPairGenerator =
                            KeyPairGenerator.getInstance("RSA");
24                      keyPairGenerator.initialize(2048);
25                      keyPair = keyPairGenerator.generateKeyPair();
26                  } catch (Exception e) {
27                      throw new IllegalStateException(e);
28                  }
29                  return keyPair;
30              }
```

The security configuration for the AS also includes a `JwtDecoder` Bean which is crucial for decoding and verifying JWT tokens, ensuring that only tokens with valid signatures are processed. This component is supported by Spring's `spring-security-oauth2-jose` library, which is necessary to enable resource server support for JWT-encoded Bearer Tokens

(access tokens). The presense of the decoder is important to validate the JWTs against the keys provided by a JSON Web Key (JWK) source, thereby enforcing token integrity and authenticity (Team, 2023)

The `JWKSource` Bean generates a source of JWKs, which are used for signing JWTs or for other services to validate the signatures of JWTs issued by the server. In our configuration, RSA keys are generated, emphasising the use of public and private keys in the signing and validation process. This setup underscores the secure handling of tokens by ensuring that JWTs can be authenticated effectively, enhancing the security of our OAuth 2.0 flow. Further more, this configuration also allows for custom implementation of cryptographic protocols for signing JWTs even thought that is not usually recommended.

6.3 Spring Boot Repository Layer

In Spring Boot, the Repository layer plays a pivotal role in abstracting the complexities of data access within the application. By utilising Java Persistence API (JPA) repositories, it offers a streamlined, efficient approach to handling database operations. This layer stands out due to its ability to simplify CRUD (Create, Read, Update, Delete) operations. The integration with JPA repository is particularly beneficial, as it not only provides built-in methods for common data access tasks but also supports the addition of custom methods, enhancing flexibility. The code below from some examples of the Repositories can show how this works. Furthermore, as spoken about earlier in the document, the spring data JPA creates a perfect barrier between the application and the database such that vulnerabilities like SQL injections can be prevented as securely tested code is used to make JPA functions.

6.3.1 CompanyRepository

```
1  @Repository
2  @Transactional
3  public interface CompanyRepository extends JpaRepository<Company,
       Long> {
4      Boolean existsByName(String name);
5
6      Company findByName(String name);
7
8      @Query("SELECT c.name FROM Company c WHERE c.id > 0")
9      Set<String> getAllCompanyNames();
10 }
```

The company repository as shown above is a simple JPA repository with methods that are used in the service layer to find companies by name, checking whether a company exists or getting a set of all companies.

6.3.2 Role Repository

```
1  @Repository
2  public interface RoleRepository extends JpaRepository<Role, Long> {
```

```
3                   Optional<Role> findByName(String name);
4               }
```

The job of the role repository is to communicate simply with the Role Entity (Entities are defined in the next section). The only method present here is the `findByName(String)` method which fetches a "role" object within the database based on the name provided. The Optional¡Role¿ allows the database to return null if no Role objects are found in the database table.

The client application has 2 more JPA repositories that are responsible for communicating to their respective tables in the database. The `UserRepository` communicating with the "user" table at the client level and the `AuthorityRepository` communicating with the "authority" table at the client level. It is important to note that these tables are all positioned at the client and are solely responsible for the functionality of the ERP HCM solution while not interfering with any functionality of the OAuth2.0 implementation.

6.4 Spring Boot Service Layer

In this project, the Service layer is positioned above the Repository layer, serving as an intermediary that refines and abstracts data fetched from the repositories to the controller. This layer is essential for processing and transforming data, ensuring it meets the application's business requirements. It plays a critical role in handling exceptions and throwing errors to the controller with the aim of maintaining the readability of the application. By delegating these responsibilities to the service layer, the application's architecture remains clean and uncluttered, promoting better organisation and separation of concerns.

In the security configuration shown in **section 6.2.1** a reference to the method `this.getOIDCUserService()` is made, this method simply returns the service class that is responsible for handling the authentication from then on. If the user choses to login with "Google" the method will return the `GoogleOIDCUserService` class and if the user chose "Auth-server" which is our local implementation of an AS, the `AuthServerOIDCUserService` class will be returned.

```
1           private OAuth2UserService<OidcUserRequest, OidcUser>
                getOIDCUserService() {
2               return (userRequest) -> {
3                   if
                        (userRequest.getClientRegistration().getRegistrationId().equals(GOOGLE))
                        {
4                       return new GoogleOIDCUserService(userRepository,
                            userService).loadUser(userRequest);
5                   } else if
                        (userRequest.getClientRegistration().getRegistrationId().equals(AUTH_SERVER)
                        {
6                       return new AuthServerOIDCUserService(userRepository,
                            userService, restTemplate()).loadUser(userRequest);
7                   } else {
8                       throw new UnknownRegistrationIdException("Unknown
                            Registration ID");
9                   }
10              };
```

```
11          }
```

It can be seen in the above code how the method returns `GoogleOIDCUserService(userRepository, userService).loadUser(userRequest)` when the `registrationId` is "Google".

The custom OIDC user service classes (`GoogleOIDCUserService` and `AuthServerOIDCUserService`) are perfect examples to demonstrate both, how Spring Security works as well as how OAuth2.0 works. The goal of these classes is to manage what happens when a successful authentication happens. In this case, if a user logs in with an AS, the services check to see what roles, authorities and information the application has from the **RS**. Once this is done, Spring Security will load this information onto the **SecurityContextHolder**.

> *In Spring Boot, the Security Context Holder is simply an object that holds session related information regarding the authenticated user. This is the stage at which session management is handled by Spring Boot. The Security Context Holder is managed per thread which means there is no room for confusion between multiple sessions. Below is a simple listing that shows how to return the "currently authenticated user".*

```
1           User user = (User)
                SecurityContextHolder.getContext().getAuthentication().getPrincipal();
```

6.4.1 Custom User Service Implementation for Local Authorisation Server

```
1               @AllArgsConstructor
2               public class AuthServerOIDCUserService extends OidcUserService {
3
4
5                   private final UserRepository userRepository;
6                   private final UserService userService;
7                   private final RestTemplate restTemplate;
8                   private static final String USER_INFO_ENDPOINT =
                        "http://localhost:8082/userinfo";
9
10                  private OidcUserInfo getUserDetails(String accessToken) {
11                      HttpHeaders headers = new HttpHeaders();
12                      headers.setBearerAuth(accessToken);
13                      HttpEntity<?> entity = new HttpEntity<>(headers);
14
15                      ResponseEntity<Object> response =
                            restTemplate.exchange(USER_INFO_ENDPOINT, HttpMethod.GET,
                            entity, Object.class);
16                      if (response.getStatusCode() == HttpStatus.OK) {
17                          HashMap<String, Object> userDetails = (HashMap<String,
                                Object>) response.getBody();
18                          Assert.notNull(userDetails, "Resource Server Responded
                                With Empty Body.");
```

```
19                          String email = (String) userDetails.get("email");
20                          // .. additional user details
21                          Map<String, Object> claims = new HashMap<>();
22                          claims.put("sub", email);
23                          // ... additional claims
24                          return new OidcUserInfo(claims);
25                      } else {
26                          throw new UsernameNotFoundException("User not found on
                                resource server.");
27                      }
28                  }
29
30                  @Override
31                  public OidcUser loadUser(OidcUserRequest userRequest) {
32                      Optional<User> userOptional =
                            userRepository.getRegisteredUserByEmailAndIssuer((String)
                            userRequest.getIdToken().getClaims().get("sub"),
                            SSOIssuer.AUTH_SERVER);
33                      OidcUserInfo oidcUserInfo =
                            getUserDetails(userRequest.getAccessToken().getTokenValue());
34                      if (userOptional.isEmpty()) {
35                          OIDCUserInformationDTO oidcUserInformationDTO = new
                                OIDCUserInformationDTO(oidcUserInfo.getClaims());
36                          try {
37                              return
                                    userService.registerUser(oidcUserInformationDTO,
                                    userRequest.getIdToken(), oidcUserInfo,
                                    SSOIssuer.AUTH_SERVER);
38                          } catch (NoSuchRoleException | AlreadyExistsException e) {
39                              throw new RuntimeException(e);
40                          }
41                      } else {
42                          User user = userOptional.get();
43                          List<GrantedAuthority> mappedAuthorities =
                                (List<GrantedAuthority>) user.getAuthorities();
44
45                          return new DefaultOidcUser(mappedAuthorities,
                                userRequest.getIdToken(), oidcUserInfo);
46                      }
47                  }
48              }
```

The `AuthServerOIDCUserService` class extends the `OidcUserService` provided by Spring Security's OAuth2.0 implementation. This class is specifically designed to handle the user registration / user loading by fetching user information from something called the "userinfo" endpoint. The "userInfo" endpoint is supposed to provide the claims of the authenticated user (Spring Security Team, 2018) ('claims' is the word used by OAuth2.0 official documentation to describe user information parameters like firstname, lastname, etc.). In most applications, the "userinfo" endpoint is positioned at the AS as a convention. This convention is entirely arbitrary since OAuth2.0 does not have any stringent policies against it. However, in our

implementation, a novel decision to place the userinfo endpoint at the RS is made to pursue a more cohesive micro-services architecture within the application.

> *This decision creates a level of isolation such that the table that stores the password of the user is entirely different from the table that stores the information of the user. This means that an attacker with access to one would not have any access to the other entirely.*

The variable `USER_INFO_ENDPOINT` refers to the URL at which a get request with the appropriate headers would return information of the user. The method `getUserDetails(String accessToken)` returns `OidcUserInfo` which is just a wrapper for the user information that is received from the RS. The method clearly demonstrates a REST API call being sent to the resource server at **line 15**, this API call is a simple get request with the *access_token* as the header value and the response is a JSON object with user detail of the authenticated user in them.

Finally, a `loadUser(OidcUserRequest)` method, which is overridden that aims to return an `OidcUser`. The goal of this method is to simply load an `OidcUser` into the Spring security context, allowing for future processing. This method executes a function to ascertain the presence of a user within the system based on a specific email address. It accomplishes this by invoking the `getRegisteredUserByEmailAndIssuer(email, issuerName)` function, which queries the database for a user associated with the given email and issuer name. Should the user already exist within the system, they are directly returned by the method. However, in the absence of the user, the `registerUser()` method in the `UserService` class is called to initiate the creation of a new user within the system. Following successful registration, the function then returns this new user.

6.4.2 Custom User Service Implementation for Google's Authorisation Server

```
1               @Override
2               public OidcUser loadUser(OidcUserRequest userRequest) throws
                    OAuth2AuthenticationException {
3                   OidcUser oidcUser = new OidcUserService().loadUser(userRequest);
4                   OIDCUserInformationDTO oidcUserInformationDTO = new
                        OIDCUserInformationDTO(oidcUser.getAttributes());
5                   Optional<User> userOptional =
                        userRepository.getRegisteredUserByEmailAndIssuer(oidcUserInformationDTO.getE
                        SSOIssuer.GOOGLE);
6                   if (userOptional.isEmpty()) {
7                       try {
8                           return userService.registerUser(oidcUserInformationDTO,
                                oidcUser.getIdToken(), oidcUser.getUserInfo(),
                                SSOIssuer.GOOGLE);
9                       } catch (NoSuchRoleException | AlreadyExistsException e) {
10                          throw new RuntimeException(e);
11                      }
12                  } else {
13                      User user = userOptional.get();
```

```
                    List<GrantedAuthority> mappedAuthorities =
                        (List<GrantedAuthority>) user.getAuthorities();
                    return new DefaultOidcUser(mappedAuthorities,
                        oidcUser.getIdToken(), oidcUser.getUserInfo());
                }
            }
```

The only difference between the configuration for Google and our local AS is the `loadUser` method in the OIDC service class. In this case, the simplicity arises from the fact that we do not deal with making any HTTP requests to the resource server, as this is done in the backend entirely by Spring. The way the Spring configuration knows the userinfo endpoint is through the `application-dev.yml` configuration. This is shown below. Other than that, there are no other differences between the implementation of the service between google and our local AS.

```
        provider:
          google:
            authorizationUri: https://accounts.google.com/o/oauth2/auth
            tokenUri: https://oauth2.googleapis.com/token
            userInfoUri: https://www.googleapis.com/oauth2/v3/userinfo
            userNameAttribute: sub
            jwkSetUri: https://www.googleapis.com/oauth2/v3/certs
```

The above YAML configuration shows clearly the reference to the `userInfoUri` where the user information is stored.

Similar to the OIDC user services, there are multiple service implementations in the project with discrete tasks and use cases. For example, the AdminService will have tasks that are specific to the authorities and roles of an Admin. Similarly, the RoleService has methods that interact with the Role table in the database and perform actions specific to the manipulation of roles in the application.

6.5 Spring Boot Entity Layer

The job of the Service layer as well as the Repository layer is simply to perform manipulation on the database. Data flows as requests from the Controller layer to the Service layer, and finally on to the database through the Repository layer. However, there is one more layer of code that sits below all the other layers, this is called the entity layer. At the entity layer, the goal is to represent the structure of the database. Spring Boot provides a great level of abstraction at this layer, as annotations like 'ManyToMany can be used to describe a many-to-many relationship between entities (tables). Below is the code for all entities that are relevant to the context of this report.

```
        @Entity
        @Getter
        @Setter
        @Table(name = "user")
        public class User implements UserDetails {

```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    private String email;
    // .. more parameters

    @ManyToMany(fetch = FetchType.EAGER)
    private Set<Role> roles;

    @ManyToOne(fetch = FetchType.LAZY)
    private Company company;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        List<GrantedAuthority> grantedAuthorityList = new ArrayList<>();
        for (Role role : this.roles) {
            grantedAuthorityList.add(new CustomGrantedAuthority("ROLE_" +
                role.getName()));
            for (Authority authority : role.getAuthorities()) {
                grantedAuthorityList.add(new
                    CustomGrantedAuthority(authority.getName()));
            }
        }
        return grantedAuthorityList;
    }

    public boolean hasAuthority(String authority) {
        for (GrantedAuthority grantedAuthority : this.getAuthorities()) {
            CustomGrantedAuthority customGrantedAuthority =
                (CustomGrantedAuthority) grantedAuthority;
            if (customGrantedAuthority.getAuthority().equals(authority)) {
                return true;
            }
        }
        return false;
    }

    public boolean hasRole(String roleName) {
        if (this.roles == null || this.roles.isEmpty()) {
            return false;
        }
        return this.roles.stream()
                .anyMatch(role -> role.getName().equals(roleName));
    }

    // .. public getter and setter methods

    @Override
```

```
54              public boolean equals(Object o) {
55                  return ((o instanceof User)
56                          && (Objects.equals(this.id, ((User) o).getId()))
57                          && (Objects.equals(this.email, ((User) o).getEmail()))
58                          && (this.getAuthorities().equals(((User)
                                o).getAuthorities())));
59              }
60
61              public boolean isAdmin() {
62                  return (this.getAuthorities().contains(new
                        SimpleGrantedAuthority("ROLE_ADMIN")));
63              }
64          }
```

The code above demonstrates the definition of the user entity at the client level. The purpose of the entity is emphasised more by the `Table` annotation at the top, where it defines the name of the table that Spring Boot should create in the database. The `getAuthorities` method is particularly noteworthy as it dynamically constructs a collection of granted authorities based on the user's roles and specific permissions, enabling fine-grained access control. This means that at any point at the service level when a comparison of authorities needs to be made, the `getAuthorities` method will be called. By leveraging Spring Security's `User Details` interface, this entity not only serves as a data model but also plays a crucial role in the security framework, enabling the system to make authorisation decisions based on the user's granted authorities. Furthermore, the fact that this entity is a `UserDetails` object, spring security is able to use it in default authentication mechanisms, further preventing us from re-inventing the wheel. Moreover, the entity shows mappings with other entities like "role" and "company" that demonstrate the structure of the schema better. For example, one user can have multiple roles and one user can only be a part of one company.

> A reference was made earlier in this document to the structure of "roles" and "authorities", which were characterised as a spectrum rather than as rigid types. This approach is quite novel, as most applications typically define these elements as fixed categories. While both methodologies have their advantages and disadvantages—the spectrum approach offering greater flexibility, and the rigid approach requiring more storage—this distinction provides a prime opportunity to demonstrate its practical implementation within the codebase. The listings below show how the mapping between "roles" and "authorities" has been made.

Roles Entity

```
1               @Entity
2               @Getter
3               @Setter
4               @Table(name = "role")
5               @NoArgsConstructor
6               public class Role {
7
8                   @Id
9                   @GeneratedValue(strategy = GenerationType.IDENTITY)
10                  private Long id;
11
```

```
12                  private String name;
13
14                  @ManyToMany(fetch = FetchType.EAGER)
15                  @JoinTable(
16                          name = "role_authorities",
17                          joinColumns = @JoinColumn(name = "role_id"), // Column
                                name from the role table
18                          inverseJoinColumns = @JoinColumn(name = "authority_id")
                                // Column name from the authority table
19                  )
20                  private Set<Authority> authorities;
21
22                  public void addAuthority(Authority authority) {
23                      if (this.authorities == null) {
24                          this.authorities = new HashSet<>();
25                      }
26                      this.authorities.add(authority);
27                  }
28              }
```

Authorities Entity

```
1           @Entity
2           @Getter
3           @Setter
4           @Table(name = "authority")
5           @NoArgsConstructor
6           public class Authority {
7               @Id
8               @GeneratedValue(strategy = GenerationType.IDENTITY)
9               private Long id;
10
11              private String name;
12
13              public Authority(String name) {
14                  this.name = name;
15              }
16          }
```

The mapping shows `Many-to-many` relationship between roles and authorities where a new table called `role_authorities` is created such one role can have multiple authorities and one authority can be shared by multiple roles. This creates a spectrum such that admins of different companies can have entirely different authorities.

The user entity in the client, as shown in the listings above, has a `Many-to-many` mapping with roles, this means that one user can have multiple roles. This again

Figure 6.4: User Roles Relationship

means that one user can have multiple roles, and that one role can be shared by multiple users. The reason for having this `Many-to-many` mapping is to stay consistent with the philosophy that was used in the mapping between the roles and authorities. This mapping creates a table called `user_roles`, which is shown in **figure 6.4** as a screenshot of the database visualiser.

Just like the client, our local implementation of the AS and the RS both have user entities but with entirely different purposes. The AS has a user entity which only has two parameters, the user's email and the users' password. This can be shown in the listing below:

User Entity in the AS

```
1           @Entity
2           @Getter
3           @Setter
4           @Table(name = "user")
5           public class User implements UserDetails {
6
7               @Id
8               @GeneratedValue(strategy = GenerationType.IDENTITY)
9               private Long id;
10
11              private String email;
12
13              private String password;
14
15              //.. public getter and setter methods.
16          }
```

Similarly, the RS also has a user entity, serving a distinct purpose. The primary objective of having this user entity is to store user information, uniquely identified by their email. The code for the user entity in the RS can be seen below:

```
1           @Entity
2           @AllArgsConstructor
3           @NoArgsConstructor
```

```
4           @Getter
5           @Setter
6           public class UserInformation {
7
8               @Id
9               @GeneratedValue(strategy = GenerationType.IDENTITY)
10              private Long id;
11
12              private String firstName;
13
14              private String lastName;
15
16              private String email;
17
18              private String profilePicture;
19
20          }
```

Unlike the user entities in the AS and the client, the RS does not implement `UserDetails`. This is because the RS does not require handling any aspects of security or authentication, further illustrating our concept of isolation. The RS simply stores the user information of the user, which will be later queried by the controller at the userinfo endpoint.

Similar to the "User" entity, the project incorporates several other entities such as "Company" which serve as templates for database tables. While a comprehensive ERP HCM solution would necessitate a detailed focus on the structure and coding of these entities, our primary objective in this context is to demonstrate authentication and authorisation capabilities. Hence, emphasising the "User" entities suffices for this purpose.

6.6 Spring Boot Controller Layer

In Spring Boot, Controllers are a fundamental part of the MVC (Model-View-Controller) architecture, responsible for handling incoming HTTP requests and returning responses to the client. They act as an intermediary between the View, which presents data, and the Model, which processes data. Controllers interpret user inputs converted into models, which are rendered into views. They are annotated with '@Controller' or '@RestController', the latter implying a combination of '@Controller' and '@ResponseBody', where the return value of the methods are directly bound to the web response body (Varanasi and Belida, 2015). In the case of the ERP HCM solution, a REST Controller is being used. Below is an example of a controller that is being used in the application, the code below demonstrates method level security as a proof-of-concept.

```
@RestController
@RequestMapping("/onboarding")
@AllArgsConstructor
public class OnboardingController {

    private final UserService userService;
    private final AdminService adminService;

    @PreAuthorize("#email == authentication.principal.attributes['email'] &&
        hasRole('NON_ONBOARDED_USER_A')")
    @PostMapping(path = "/{email}/update/userInfo", consumes =
        "application/json", produces = "application/json")
    public ResponseEntity<?> updateUserInfo(@RequestBody UserInfoDTO
        userInfoDTO, @PathVariable String email) {
        String firstName = userInfoDTO.getFirstName();
        String lastName = userInfoDTO.getLastName();
        try {
            if (firstName != null && !firstName.isEmpty() &&
                !firstName.equals(this.userService.getFirstName(email))) {
                this.userService.updateFirstName(email, firstName);
            }
            if (lastName != null && !lastName.isEmpty() &&
                !lastName.equals(this.userService.getLastName(email))) {
                this.userService.updateLastName(email, lastName);
            }

            this.userService.updateRole(email, "NON_ONBOARDED_USER_B");

            return ResponseEntity.ok().build();
        } catch (Exception e) {
            return ResponseEntity.badRequest().body(e.getMessage());
        }
    }

    // more code...

}
```

The above controller is a controller that is being used by the onboarding portal of the application. The idea here is to have an overly secure suite of endpoints to prove method level security for RBAC. In the above code, the '@PreAuthorize' annotation is used to check if the endpoint should pass the filter or not.

> *In this system, when a user logs in, they are automatically assigned a role. This role is called "NON_ONBOARDED_USER_A". Any user with this role will not have any other roles in the system. A user with the role "NON_ONBOARDED_USER_A" can only perform one query to the API. If*

this query is successful, the users' role will change to "NON_ONBOARDED_USER_B". This process will repeat until all onboarding steps have been completed, and the user has been assigned a proper role.

Hence, the '@PreAuthrorize' annotation allows Spring Security to check if the user has the required roles and that the action the user is trying to conduct is one of the users authorities. If not, the request will be redirected to "/access-denied" — as seen in the Security Configuration at first — **section 6.2.1**.

One of the other controllers which is of particular importance to the scope of this project is the `AdminController`. The goal of the admin controller is to define endpoints that allow the admin to do privileged tasks. These tasks as defined in **section 5.3.3**. The endpoints in the `AdminController` are annotated with `AdminActionAuthorize(email, actionType)` and `PreAuthorize` annotations to provide method level security to the endpoints. These can be thought of as filters that a request will go through before the code in the endpoint function is run. Below is the code for the `AdminActionAuthorize(email, actionType)` aspect that works as an annotation for the endpoints:

```
1           @Around("@annotation(adminActionAuthorize)")
2           public Object authorizeAdminActionOnSubject(ProceedingJoinPoint
                joinPoint, AdminActionAuthorize adminActionAuthorize) throws
                Throwable {
3
4               String actionSubjectEmail =
                    adminActionAuthorize.actionSubjectEmail();
5               User actionSubject = (User)
                    userService.loadUserByUsername(actionSubjectEmail);
6               User admin = (User)
                    SecurityContextHolder.getContext().getAuthentication().getPrincipal();
7               if
                    (adminActionAuthorize.actionType().equals(AdminActionType.WRITE_ROLES))
                    {
8                   if (admin.hasAuthority("WRITE_" +
                        actionSubject.getCompany().getName().toUpperCase() +
                        "_ROLES")){
9                       return joinPoint.proceed();
10                  } else {
11                      throw new AccessDeniedException("Admin doesn't have authority
                            to do that");
12                  }
13              } else if
                    (adminActionAuthorize.actionType().equals(AdminActionType.WRITE_AUTHORITIES))
                    {
14                  if (admin.hasAuthority("WRITE_" +
                        actionSubject.getCompany().getName().toUpperCase() +
                        "_AUTHORITIES")){
15                      return joinPoint.proceed();
16                  } else {
```

```
                    throw new AccessDeniedException("Admin doesn't have authority
                        to do that");
                }
            }
            throw new AccessDeniedException("Admin doesn't have authority to do
                that");
        }
```

The method above employs the `UserService` class to check the user's authorities, verifying if the user has the authority required by the endpoint. This required authority is delineated by an `enum` named `AdminActionType`. Upon confirming that the user holds the necessary authority, the method executes `return joinPoint.proceed()`, effectively serving as an authorisation to proceed with the intended endpoint method. Conversely, if the user lacks the required authority, it triggers `throw new AccessDeniedException(\Admi doesn't have authority to do that")`, thereby denying access. The existence of the `@AdminActionAuthorize` annotation, alongside the `@PreAuthorize` annotation, serves not only to demonstrate the implementation of method-level security through annotations in Spring Boot but also introduces a layer of flexibility. This flexibility allows administrators of different companies to possess distinct authorities, thereby facilitating a tailored and secure approach to managing user permissions.

AdminActionAuthorize Interface

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface AdminActionAuthorize {

    String actionSubjectEmail();

    AdminActionType actionType();
}
```

AdminActionType Enum

```
public enum AdminActionType {

    WRITE_ROLES,
    WRITE_AUTHORITIES
}
```

The client application has many more controllers that are specific to the features defined in the software requirements section of this document. While all those features implement some form of RBAC they do not add anything new to the already established information this document has provided. However, as discussed before, the novelty in the implementation of the userinfo endpoint can be seen in the controller method in the RS. The AS implementation by default places the userinfo endpoint on the AS

however in our implementation, the userinfo endpoint is placed in a controller within the RS called the `UserInfoController`. This can be seen below:

UserInfoController in The Resource Server

```
@RestController
@AllArgsConstructor
public class UserInfoController {

    private final UserInformationService userInformationService;

    @GetMapping("/userinfo")
    @PreAuthorize("hasAuthority('SCOPE_user.read') &&
        hasAuthority('SCOPE_openid')")
    public ResponseEntity<?> getUserInfo(@AuthenticationPrincipal Jwt
        jwt) {
        return
            ResponseEntity.ok().body(this.userInformationService.loadUserByEmail(jwt.get
    }
}
```

The userinfo endpoint executes a `@PreAuthorize` check to determine whether the client requesting the resource, with the provided access token, has the required authorities. In this case, the authorities being checked are the scopes defined by the AS.

6.7 Hurdles During Development

6.7.1 Hostname Cookie Sharing Issue

During the development of the ERP HCM solution, a perplexing issue was encountered while configuring the OAuth2.0 authentication flow with the local AS. The application was developed on a local machine, with the client running on `http://localhost:8080`, the AS running on `http://localhost:8081`, and the RS running on `http://localhost:8082`. Despite following the standard setup procedure, the AS consistently failed to authenticate the user, leading to a frustrating and prolonged debugging process.

After extensive investigation, it was discovered that the root cause of the problem lay in the way web browsers handle cookies when dealing with different ports on the same domain. By default, web browsers treat different ports on the same domain as part of the same origin, which means that cookies set by one port can be accessed by another port on the same domain. In the case of the ERP HCM solution, the client application running on `http://localhost:8080` was setting cookies that were also accessible to the AS running on `http://localhost:8081`. This unintended cookie sharing led to conflicts in the authentication process, causing the AS to misinterpret

the client's cookies as its own and resulting in authentication failures.

To resolve this issue, a simple yet effective solution was implemented. By changing the AS's location from `http://localhost:8081` to `http://127.0.0.1:8081`, the problem of cookie sharing was eliminated. Although `localhost` and `127.0.0.1` both refer to the local machine, web browsers treat them as different domains. Consequently, cookies set by the client on `localhost` were no longer accessible to the AS on AS, allowing the authentication process to proceed successfully (Josh Long, Steve Riesenberg, 2023).

6.7.2 Postman Session Syncing Workaround

One of the most significant hurdles encountered during the earlier stages of this project was related to testing the application using Postman, especially in the context of OAuth2.0 authentication. Since authentication in this project was handled by Google and not directly through the Spring Boot application, establishing a method to log in for testing purposes posed an unexpectedly complex challenge.

Typically, the solution for such scenarios involves leveraging Postman's built-in OAuth2.0 integration features. However, in this situation the application functioned flawlessly within a web browser environment, the same process repeatedly failed when attempted through Postman. This discrepancy was puzzling and led to an extensive period of research and trial-and-error, yet the root cause remained unknown.

In response to this challenge, a creative workaround was applied. The strategy involved performing the authentication step with a browser, where it operated successfully, and then transferring the session information from the browser to Postman. This approach effectively bypassed the need for direct authentication in Postman and allowed continued testing of the application. A demonstration of how that works is described thoroughly in the video submission of the interim review.

6.8 Potential Improvements and Considerations

6.8.1 Hiding The Authorisation and Resource Servers Inside a LAN

In scenarios where delegated authentication is not the primary goal, and an organisation wants to build an entire OAuth2.0 system from scratch, including the AS, Client, and RS, it is recommended to explore the possibility of hiding the AS and RS from direct internet access. This approach enhances security by reducing the attack surface and minimising the exposure of sensitive components to potential threats.

One potential area of research is the implementation of secure network architectures that isolate the AS and RS within a Local Area Network (LAN) while allowing con-

trolled access through the Client. This can be achieved using techniques such as SSH tunnelling or Network Address Translation (NAT).

By employing SSH tunnelling, the Client can establish a secure, encrypted connection to the AS and RS, which are hosted within the LAN. This ensures that all communication between the Client and the backend servers is protected from unauthorised access or interception. Additionally, SSH tunnelling allows for the AS and RS to be accessible only through the Client, effectively hiding them from direct internet exposure.

Similarly, NAT can be utilised to map internal LAN IP addresses to external, publicly accessible IP addresses. In this setup, the Client would act as the gateway, receiving requests from the internet and forwarding them to the appropriate internal servers (AS or RS) based on predefined routing rules. This approach provides an additional layer of security by obscuring the internal network structure and preventing direct access to the AS and RS from external entities.

Again, this is entirely an area of research and has not been implemented as a software engineering practice. In truth, this system may be a very computing intense system such that the level of security gained would not account for the resources used. However, research in developing a system like this would nevertheless enhance our understanding of limitations with security within authentication system. **Figure 6.5** below is a demonstration of the proposed solution.

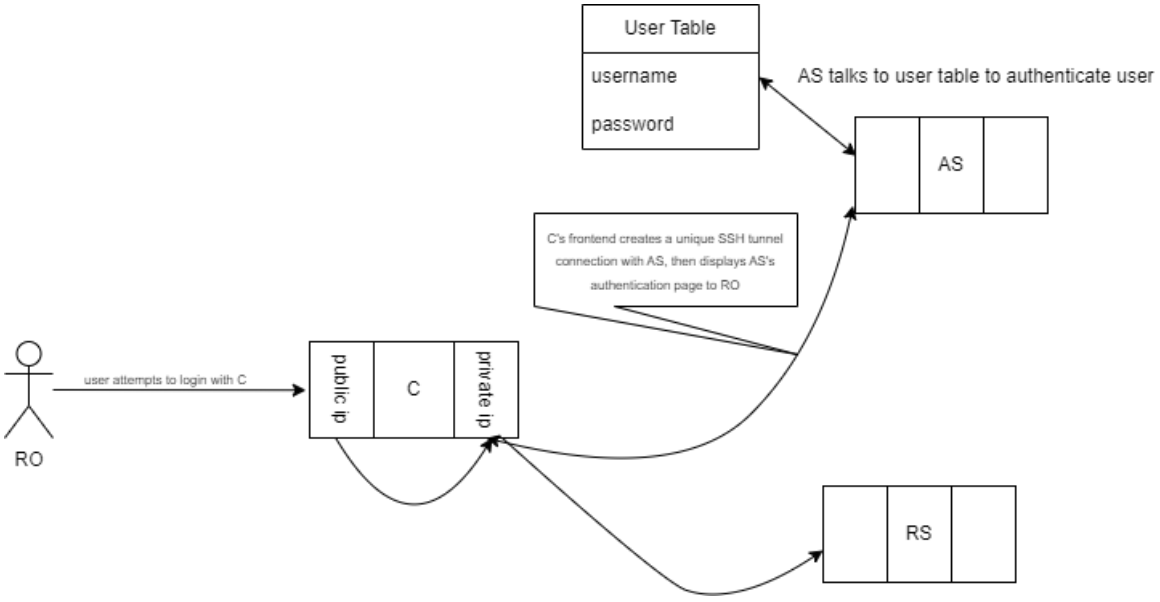

Figure 6.5: AS and RS in a Local Area Network

6.8.2 Client Authentication with Private-Key JWT

Another potential improvement to enhance the security of the OAuth2.0 implementation in the ERP HCM solution is to explore the use of Private-Key JWT for client authentication with the AS. Currently, the implementation relies on a simple client

credentials sent to the AS in a HTTP POST request (briefly described in **section 6.1.1**) for client authentication, while functional, may not provide the highest level of security.

Private-Key JWTs offer a more secure alternative by leveraging asymmetric cryptography for client authentication (Jones et al., 2015b). In this approach, the client generates a public-private key pair and shares the public key with the AS during the client registration process. When the client needs to authenticate with the AS, it creates a JWT and signs it using its private key. The AS can then verify the authenticity of the JWT using the client's public key, ensuring that the request originates from a legitimate client.

In conclusion, allowing client authentication with the AS using Private-Key JWTs is a potential improvement worth exploring in the ERP HCM solution. By leveraging asymmetric cryptography and eliminating the need for shared secrets, Private-Key JWTs offer enhanced security for client authentication.

6.8.3 Utilising Azure Key Vaults

In the current implementation of the ERP HCM solution, sensitive information such as database credentials and OAuth2.0 client secrets are stored in environment variables. While this approach is relatively secure compared to hardcoding secrets in the codebase as described in the code review section of this document, it still poses potential risks. Environment variables can be accidentally exposed or accessed by unauthorised parties, compromising the security of the application.

To enhance the security of sensitive information, it is recommended to explore the use of a dedicated key management system like Azure Key Vault. Azure Key Vault is a cloud-based service that provides secure storage and management of secrets, keys, and certificates (Microsoft, 2023).

6.8.4 Multi-Factor Authentication and Containerisation with Docker

During the development of the ERP HCM solution, there were several additional features and improvements that were planned but could not be implemented due to time constraints. One of these potential enhancements was the integration of Multi-Factor Authentication (MFA) in the AS. MFA is a crucial security practice in modern authentication systems, providing an extra layer of protection beyond traditional username and password authentication (Abhishek et al., 2013). By requiring users to provide an additional form of authentication, such as a one-time password or a biometric factor, MFA significantly reduces the risk of unauthorised access, even if a user's credentials are compromised. Implementing MFA in the AS would have showcased a best practice for enhancing the security of the OAuth2.0 flow and protecting user accounts from

potential threats.

Another planned improvement was the containerisation of the different components of the ERP HCM solution using Docker. Docker is a popular platform that enables the packaging of applications and their dependencies into lightweight, portable containers (Merkel, 2014). By containerising the client, AS, RS, it would have guaranteed that the markers of this final year project would have working copy without any dependency based issues.

While the integration of MFA and containerisation was not feasible within the scope of this project, they remain valuable potential improvements for future iterations, such that they will be implemented in this solution after the completion of this degree.

7. Professional Issues

7.1 Responsible Disclosure of OAuth2.0 Vulnerabilities

Responsible disclosure is a crucial ethical principle that should guide the actions of security researchers and developers when identifying and addressing vulnerabilities in software systems, including those related to OAuth 2.0 and OpenID Connect implementations (Pupillo et al., 2018). When a vulnerability is discovered, it is essential to follow a responsible disclosure process, which involves notifying the affected parties, such as the software vendor or the organisation maintaining the system, and providing them with sufficient time to develop and deploy a patch before publicly disclosing the vulnerability (Schrittwieser et al., 2012).

In the context of the vulnerabilities discussed in this project, such as the Facebook OAuth 2.0 Framework vulnerability, it is important to emphasise that these vulnerabilities should not be exploited for malicious purposes. Instead, they should be reported to the relevant parties through established channels, allowing them to address the issues and protect their users. Exploiting vulnerabilities without proper disclosure can lead to significant harm, compromising user privacy, data security, and system integrity.

The ethical concerns surrounding the exploitation of OAuth 2.0 vulnerabilities extend beyond the direct impact on the affected systems and users. It is crucial to consider the broader implications of such actions, as they can erode public trust in the security of web applications and undermine the efforts of the security community to build a safer online environment. By adhering to responsible disclosure practices, security researchers and developers can contribute to the overall improvement of OAuth 2.0 and OpenID Connect implementations, promoting a more secure and trustworthy ecosystem.

7.2 Ambiguity in Security Best Practices

OAuth 2.0 and OpenID Connect are flexible and extensible frameworks that provide guidelines for secure authentication and authorisation processes. However, this flexibility can sometimes lead to ambiguity in the interpretation and implementation of security best practices (T. Lodderstedt and Authlete, 2024). The documentation for these frameworks often focuses on the desired outcomes and general principles rather than providing detailed, prescriptive implementation guidelines (T. Lodderstedt and Authlete, 2024).

This ambiguity can result in inconsistencies among different implementations, as devel-

opers may interpret and apply the guidelines differently based on their understanding and specific use cases. Consequently, this has led to the repetition of references to similar security practices throughout the project, as developers strive to ensure compliance with the overall security objectives outlined in the OAuth 2.0 and OpenID Connect specifications.

To mitigate the risks associated with ambiguous security best practices, it is essential for developers to actively engage with the OAuth 2.0 and OpenID Connect communities, staying informed about the latest security recommendations, and seeking clarification when needed. Additionally, referring to well-established libraries and frameworks that encapsulate best practices, such as the official Spring Security OAuth2 page, can help ensure a more consistent and secure implementation of these standards.

7.3 Unit Testing Methods and Test-Driven Development (TDD)

Test-Driven Development (TDD) is a software development approach that emphasises writing unit tests before implementing the actual code. This approach helps ensure that the code is testable, maintainable, and meets the desired functionality (George and Williams, 2004). However, due to time constraints and the primary focus on demonstrating the implementation of OAuth 2.0 and OpenID Connect in a Spring Boot application, TDD was not used in this project.

While the absence of comprehensive unit tests may result in potential bugs or issues, it is essential to recognise that the primary goal of this project was to showcase the integration of secure authentication and authorisation mechanisms using OAuth 2.0 and OpenID Connect. The project aimed to provide a foundational understanding of these protocols and their implementation within the context of a Spring Boot application, rather than delivering a production-ready solution.

By adopting TDD practices in future iterations of the project, potential issues can be addressed early on in the development process, code quality can be improved, and the application will remain maintainable and adaptable to changing requirements.

7.4 Linguistic Inconsistencies

Throughout this project report, there may be minor linguistic inconsistencies in the spelling of certain terms. The American English spelling “authorization” is used when discussing OAuth 2.0 and OpenID Connect specific terminology, aligning with the conventions used in the specifications and the software development industry. However,

in other parts of the report, the British English spelling "authorisation" is employed, reflecting the linguistic norms of the United Kingdom, where this project was conducted. These inconsistencies should not impact the clarity or technical accuracy of the report, but they exist to prevent confusion if this document is used as a tutorial of development using Spring Boot.

8. Conclusion

In conclusion, this project has successfully demonstrated the implementation of a secure and modern authentication system using OAuth 2.0 and OpenID Connect with Spring Boot Java. The primary objective was to showcase how these widely adopted authentication and authorisation standards can be effectively utilised to build a robust and secure application, using the example of a simplified Enterprise Resource Planning Human Capital Management web application.

Throughout the project, a strong emphasis was placed on adhering to security best practices and addressing common pitfalls associated with OAuth 2.0 and OpenID Connect implementations. By conducting a thorough analysis of these pitfalls, such as the Facebook OAuth 2.0 Framework vulnerability, improper implementation of the Implicit Grant Type, flawed scope validation, and vulnerability to session fixation attacks, the project highlighted the importance of understanding and mitigating these risks to ensure a secure authentication system.

The project also demonstrated the benefits of delegated authentication and authorisation, which allow for the outsourcing of these processes to more robust and secure systems. By leveraging the expertise of identity providers like Google and a custom-built authorisation server, the application was able to enhance its overall security posture without imposing additional burdens onto developers.

The implementation of the ERP HCM solution using Spring Boot Java showcased the framework's capabilities in simplifying the development process while maintaining a high level of security. The project structure, which included the client application, authorisation server, and resource server, demonstrated the principles of separation of concerns and isolation, further enhancing the system's security.

The code review section of the document provided an in-depth look at the various components of the Spring Boot application, including configurations, security configurations, repository layer, service layer, entity layer, and controller layer. By examining these components in detail, the project illustrated how Spring Boot's features and abstractions can be effectively utilised to build a secure and maintainable application.

Moreover, the project addressed various issues encountered during the development process, such as the hostname cookie sharing issue, Postman session syncing workaround, and HTTP-only cookie requirement. By documenting these challenges and their respective solutions, the project serves as a valuable resource for developers facing similar issues in their own OAuth 2.0 and OpenID Connect implementations.

While the project successfully achieved its goals of demonstrating a secure authentication system using OAuth 2.0 and OpenID Connect with Spring Boot Java, it also acknowledged potential improvements that could be made in future iterations. These

improvements include hiding the authorisation server and resource server in a LAN, enforcing bespoke network security requirements, utilising Azure Key Vault for storing secrets, implementing Docker for containerisation, and integrating multi-factor authentication in the local authorisation server.

In summary, this project has provided a comprehensive overview of modern authentication and authorisation methods, focusing on the secure implementation of OAuth 2.0 and OpenID Connect using Spring Boot Java. By addressing common pitfalls, adhering to security best practices, and demonstrating the benefits of delegated authentication and authorisation, the project serves as a valuable resource for developers looking to build secure and scalable applications. The lessons learned, and the potential improvements identified throughout the project, will undoubtedly contribute to the ongoing evolution of secure authentication and authorisation practices in the rapidly changing landscape of web application development.

Bibliography

Setting up oauth 2.0. Technical report, Google.

Microsoft identity platform and oauth 2.0 implicit grant flow. https://learn.microsoft.com/en-us/entra/identity-platform/v2-oauth2-implicit-grant-flow, October 2023.

Kumar Abhishek, Sahana Roshan, Prabhat Kumar, and Rajeev Ranjan. A comprehensive study on multifactor authentication schemes. In Natarajan Meghanathan, Dhinaharan Nagamalai, and Nabendu Chaki, editors, *Advances in Computing and Information Technology*, pages 561–568, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-31552-7.

Marios Argyriou, Nicola Dragoni, and Angelo Spognardi. Security flows in oauth 2.0 framework: a case study. In *Computer Safety, Reliability, and Security: SAFECOMP 2017 Workshops, ASSURE, DECSoS, SASSUR, TELERISE, and TIPS, Trento, Italy, September 12, 2017, Proceedings 36*, pages 396–406. Springer, 2017.

Aruba Networks. Authentication server load balancing. https://www.arubanetworks.com/techdocs/ArubaOS_87_Web_Help/Content/arubaos-solutions/auth-servers/auth-serv-load-bala.htm, 2023. Accessed: 2023-03-23.

Auth0. Application grant types. https://auth0.com/docs/get-started/applications/application-grant-types, 2024a.

Auth0. Confidential and public applications. https://auth0.com/docs/get-started/applications/confidential-and-public-applications, 2024b.

Amol Baikar. Facebook OAuth Framework Vulnerability. https://www.amolbaikar.com/facebook-oauth-framework-vulnerability/, 2020.

Nate Barbettini. Oauth 2.0 and openid connect (in plain english), 2018. URL https://www.youtube.com/watch?v=996OiexHze0.

William J Buchanan, Scott Helme, and Alan Woodward. Analysis of the adoption of security headers in http. *IET Information Security*, 12(2):118–126, 2018.

EbruCelikel Cankaya. *Authentication*, pages 61–62. Springer US, Boston, MA, 2011. ISBN 978-1-4419-5906-5. doi: 10.1007/978-1-4419-5906-5_772. URL https://doi.org/10.1007/978-1-4419-5906-5_772.

Ayan Chatterjee, Martin W Gerdes, Pankaj Khatiwada, and Andreas Prinz. Sftsdh: Applying spring security framework with tsd-based oauth2 to protect microservice architecture apis. *IEEE Access*, 10:41914–41934, 2022.

DocuSign. Implicit grant authentication. https://developers.docusign.com/platform/auth/implicit, 2024.

Boby George and Laurie Williams. A structured experiment of test-driven development. *Information and software Technology*, 46(5):337–342, 2004.

Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012. URL `https://www.rfc-editor.org/info/rfc6749`.

M. Jones, J. Bradley, and N. Sakimura. JSON Web Token (JWT). RFC 7519, IETF, 2015a.

Michael Jones and Dick Hardt. The oauth 2.0 authorization framework: Bearer token usage. Technical report, 2012.

Michael Jones, John Bradley, and Nat Sakimura. Json web token (jwt), 2015b.

Josh Long, Steve Riesenberg. Going Full OAuth with the new Spring Authorization Server in Spring Boot 3.1! oauth2 oauth. `https://www.youtube.com/watch?v=7zm3mxaAFWk`, 2023.

Indu Kashyap. Enhanced role based access control: integrating auditing and authentication. *Int J Comput Appl*, 72(2), 2012.

Xiaopu Ma, Ruixuan Li, Zhengding Lu, Jianfeng Lu, and Meng Dong. Specifying and enforcing the principle of least privilege in role-based access control. *Concurrency and Computation: Practice and Experience*, 23(12):1313–1331, 2011.

Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), mar 2014. ISSN 1075-3583.

Microsoft. Overview - Azure Key Vault. `https://learn.microsoft.com/en-us/azure/key-vault/general/overview`, 2023.

Chris J Mitchell, Wanpeng Li, 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*, 2019.

Okta. Oauth public client identity, 2022. URL `https://developer.okta.com/blog/2022/06/01/oauth-public-client-identity`.

OpenID Foundation. OpenID Connect Core 1.0. `https://openid.net/specs/openid-connect-core-1_0.html`, 2014a. [Online; accessed 4-April-2024].

OpenID Foundation. OpenID Connect Implicit Flow 1.0. `https://openid.net/specs/openid-connect-implicit-1_0.html`, 2014b.

OWASP. Demystifying Authentication Attacks. `https://owasp.org/www-pdf-archive/Demystifying_Authentication_Attacks.pdf`, 2008.

OWASP. OAuth 2.0 Cheat Sheet. `https://cheatsheetseries.owasp.org/cheatsheets/OAuth2_Cheat_Sheet.html`, 2023. [Online; accessed 4-April-2024].

OWASP Foundation. Path traversal. https://owasp.org/www-community/attacks/Path_Traversal, 2024.

Piyush Pant, Anand Singh Rajawat, S.B. Goyal, Pradeep Bedi, Chaman Verma, Maria Simona Raboaca, and Florentina Magda Enescu. Authentication and authorization in modern web apps for data security using nodejs and role of dark web. *Procedia Computer Science*, 215:781–790, 2022. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2022.12.080. URL https://www.sciencedirect.com/science/article/pii/S1877050922021512. 4th International Conference on Innovative Data Communication Technology and Application.

Aaron Parecki. What is the oauth 2.0 implicit grant type? *Okta Developer Blog*, May 2018. URL https://developer.okta.com/blog/2018/05/24/what-is-the-oauth2-implicit-grant-type.

Michal Aibin Philippe Sevestre. Sql injection and how to prevent it?, 2021. URL https://www.baeldung.com/sql-injection. Accessed: 2024-03-29.

Carlos Polop. Cors bypass - pentesting web. https://book.hacktricks.xyz/pentesting-web/cors-bypass, 2021.

Sascha Preibisch. Oauth 2.0 - pkce. https://www.youtube.com/watch?v=Gtbm5Fut-j8, 2020.

Lorenzo Pupillo, Afonso Ferreira, and Gianluca Varisco. Software vulnerability disclosure in europe: Technology, policies and legal challenges, 2018.

Sebastian Schrittwieser, Peter Frühwirt, Peter Kieseberg, Manuel Leithner, Martin Mulazzani, Markus Huber, and Edgar Weippl. Guess who is texting you? evaluating the security of smartphone messaging applications. 2012.

Olimpion Shurdi, Aleksander Biberaj, Igli Tafa, and Genci Mesi. Oauth2. 0 in securing apis. 2020.

Ana Valentina Rodriguez Sosa. *An Authentication mechanism for stateless communication*. Rochester Institute of Technology, 2019.

Spring Security Team. Oauth 2.0 login: Advanced topics. https://docs.spring.io/spring-security/site/docs/5.0.7.RELEASE/reference/html/oauth2login-advanced.html, 2018.

Spring Security Team. Oauth 2.0 client authentication. https://docs.spring.io/spring-security/reference/servlet/oauth2/client/client-authentication.html, 2023.

Syed Zulkarnain Syed Idrus, Estelle Cherrier, Christophe Rosenberger, and Jean-Jacques Schwartzmann. A Review on Authentication Methods. *Australian Journal of Basic and Applied Sciences*, 7(5):95–107, March 2013. URL https://hal.science/hal-00912435.

Yubico A. Labunets D. Fett T. Lodderstedt, J. Bradley and Authlete. OAuth 2.0 Security Best Current Practice. `https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics`, 2024.

Spring Security Team. OAuth 2.0 Resource Server JWT - Spring Security. `https://docs.spring.io/spring-security/site/docs/current/reference/html5/#oauth2resourceserver-jwt`, 2023. Accessed: 2023-09-30.

Cloud Truth. The pitfalls of using environment variables for config and secrets, 2023. URL `https://www.cloudtruth.com/blog/the-pitfalls-of-using-environment-variables-for-config-and-secrets`.

Balaji Varanasi and Sudha Belida. *Spring Rest*. Apress, 2015.

Dave Westerveld. *API Testing and Development with Postman: A practical guide to creating, testing, and managing APIs for automated software testing*. Packt Publishing Ltd, 2021.