BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

SINGLE SIGN-ON WITH OPENID CONNECT AND KEYCLOAK
JEDNOTNÉ PŘIHLAŠOVÁNÍ POMOCÍ OPENID CONNECT A KEYCLOAK

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR   **MAKSYM KOVAL**
AUTOR PRÁCE

SUPERVISOR   **Mgr. KAMIL MALINKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

Bachelor's Thesis Assignment

161250

Institut:	Department of Intelligent Systems (DITS)	
Student:	**Koval Maksym**	
Programme:	Information Technology	
Title:	**Cloud-Native Single Sign-On with OpenID Connect and Keycloak**	
Category:	Security	
Academic year:	2023/24	

Assignment:

1. Get familiar with OpenID Connect, an identity layer for the OAuth 2.0 protocol. Focus on different kinds of clients (public/confidential), different authentication flows, and the scopes and claims mechanisms.
2. Investigate Keycloak, an open-source identity and access management solution. Focus on the authorization mechanism of Keycloak.
3. Propose a demo application showcasing various features of Keycloak integrated with OpenID Connect/OAuth2.
4. Implement a demo application and design a set of tests to evaluate the correctness of your implementation and perform the testing.
5. Summarize and discuss the best practices discovered while creating the demo applications.

Literature:
- Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, https://www.rfc-editor.org/info/rfc6749>.
- Sakimura, Nat, et al. "OpenID Connect Core 1.0 incorporating errata set 1, 2014."

Requirements for the semestral defence:
Items 1 to 4

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Malinka Kamil, Mgr., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	16.5.2024
Approval date:	8.4.2024

Abstract

This thesis delves into the principles of OAuth 2.0 and OpenID Connect protocols and explains how they should be implemented in a microservice architecture. Two Angular web clients and two Spring Boot servers were developed as applications for the demonstration. The paper also explains how to use Keycloak as an identity provider for the above applications. The result is centralized authentication of all applications as well as implementation of Single Sign On mechanism in a cloud-native environment.

Abstrakt

Cílem této práce je prozkoumat principy protokolů OAuth 2.0 a OpenID Connect a vysvětlit, jak by tyto protokoly měly být implementovány v architektuře mikroslužeb. Jako demonstrační aplikace byly navrhnuty dva weboví klienty Angular a dva servery Spring Boot. Práce také vysvětluje připojení Keycloak jako poskytovatele identit pro výše uvedené aplikace. Výsledkem je centralizované autentizace všech aplikací a také implementace mechanismu jednotného přihlašování v cloudovém prostředí.

Keywords

OAuth 1.0, OAuth 2.0, OpenID Connect, Keycloak, Single sign-on, Angular, Spring Boot, Access Token, ID Token, Authorization, Authentication, Identity

Klíčová slova

OAuth 1.0, OAuth 2.0, OpenID Connect, Keycloak, Jednotné přihlašování, Angular, Spring Boot, Autorizace, Autentifikace, Identita

Reference

Rozšířený abstrakt

Počet uživatelů roste, stejně jako počet aplikací a uživatelských účtů v těchto aplikacích. Systémy jednotného přihlášení s možností využívat účty z jiných aplikací, jako je Google, Facebook nebo Twitter, si proto nejen získávají zvýšenou potřebu uživatelů, ale stávají se v dnešním světě standardem. S rozvojem digitálních ekosystémů a zvyšující se potřebou jednotného uživatelského prostředí napříč aplikacemi se potřeba spolehlivých řešení jednotného přihlašování stává nezbytnou. Poptávka po mechanismu jednotného přihlášení a centralizované správě identit roste také mezi velkými a středními společnostmi, protože každá společnost využívá velké množství nástrojů, služeb a integrací, které zaměstnanec během pracovního dne používá. V kontextu firem je nutná jasná regulace a přizpůsobení práv pracovníků, proto se rychlá a jednoduchá autorizace stává nezbytnou. Cílem tohoto výzkumu je ukázat, jak může integrace OAuth 2.0 a OpenID Connect se správou identit Keycloak poskytnout bezpečný, rozšiřitelný, efektivní a centralizovaný systém ověřování a autorizace vhodný pro prostředí nativního cloudu.

Metodika výzkumu zahrnuje srovnávací analýzu autentizačních a autorizačních protokolů založených na tokenech, které jsou podporovány standardy OAuth 2.0 a OpenID Connect jako standardem pro soudobé systémy. Analyzuje také způsob centralizované správy identit a způsoby, jak dosáhnout správné implementace systému jednotného přihlašování v architektuře mikroslužeb. Keycloak byl zkoumán jako komplexní řešení pro správu identit a přístupu a jeho schopnost zjednodušit nasazení jednotného přihlášení a zvýšit bezpečnostní opatření.

Po nastudování teoretické části byla vytvořena architektura demonstračního systému, který zahrnuje dva webové klienty napsané na frameworku Angular, z nichž každý komunikuje s vlastním serverem napsaným na frameworku Spring Boot. Weboví klienty byly nakonfigurovány s autentizačními a autorizačními mechanismy pomocí knihovny 'angular-oauth2-oidc' a připojeny k systému Keycloak pro správu identit a přístupu. Klienti jsou schopni přijímat ID tokeny a přístupové tokeny vydané serverem Keycloak a načítat z nich informace o sezení a uživateli a také ověřovat jejich platnost pomocí ověřování podpisů. Rovněž byly napsány dva servery Spring Boot obsahující rozhraní REST API pro komunikaci s klienty. Servery jsou schopny kontrolovat platnost přístupových tokenů přijatých během komunikace s klienty a také získávat informace o autorizaci uživatele. Klienti i servery, stejně jako server Keycloak, byly spuštěny v cloudovém prostředí s využitím Minikube jako prostředku pro provoz lokálního clusteru Kubernetes.

Konečným výsledkem práce jsou weboví klienti podporující mechanismus jednotného přihlašování schopný autentizované interakce s backendovými servery na základě tokenů. V průběhu práce byly aplikace testovány a jejich zápis byl dokumentován.

Single sign-on with OpenID Connect and Keycloak

Declaration

I hereby declare that I have prepared this bachelor thesis independently under the supervision of Mr. Kamil Malinka. The supplementary information was provided by Mr. Olivier Rivat (RedHat). I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

<div align="right">

. . . .

Maksym Koval

May 16, 2024

</div>

Acknowledgements

I would like to thank my supervisor Mr. Kamil Malinka for his support, motivation and immediate and accurate feedback. I would also like to mention my RedHat supervisor for this thesis Mr. Olivier Rivat. He helped me in selecting resources, information to be used in the work and also provided feedback on the written work and motivation.

Contents

List of Figures

Chapter 1

Introduction

Accounts, authorization, authentication are an integral part of the modern world of information technology. A set of data about a user required for his/her identification (authentication) has become an industry standard for web applications. Along with authentication, a certain person must be granted the right to perform certain actions - authorization. Authorization also occurs during the interaction between two web applications. The purpose of these two basic mechanisms is to provide the user and the application he is using with protection, immutability, authenticity, integrity, verifiability.

Traditional authentication and authorization methods are cumbersome for users and difficult to manage for administrators. As the number of users, application usage time, and user requirements for comfort and security of applications grows, new solutions are needed. Also, nowadays the number of collaborations and partnerships of different companies is increasing and the user needs a better solution other than creating an account for each of them.

For secure authentication, there are protocols such as OAuth 2.0, SAML 2.0, Kerberos, OpenID Connect (OIDC). These protocols have different working principles, technology chains for user authentication and different characteristics such as security, speed, implementation complexity, usability and reliability.

The implementation of this SSO solution aims to enhance user experience by eliminating multiple logins, improve security through centralized authentication and authorization, and demonstrate the effectiveness of open source tools such as Keycloak for cloud-based identity management.

The paper starts with chapter two which is a comprehensive review of key authentication and authorization protocols that underpin modern security systems, including SAML 2.0, OAuth 1.0 and 2.0 Protocols and OpenID Connect Protocol.

Chapter three then delves further into the realm of identity management and detailed analysis of Keycloak's features.

Chapter four lists the functional requirements that the thesis must fulfill to achieve the goals of the work, setting the stage for system design and implementation. The remaining chapters are mandatory for almost every work are Implementation, Testing and Conclusion.

This paper details the development of an Angular client application that uses OIDC protocol for authentication and OAuth 2.0 for authorization, a Spring Boot server that provides resource security and token validation, and the configuration of Keycloak to manage user accounts and issue tokens.

This research provides valuable guidance to developers looking to implement secure and user-friendly SSO solutions for their cloud applications.

Chapter 2

Overview of Authentication and Authorization Protocols

This section will cover basics of the protocols that are used between applications to authenticate and authorise user, such as SAML 2.0, OAuth and OAuth 2.0 and OpenID Connect based on them. The concept of the OAuth protocol as the basis for OAuth 2.0 and its extension - OpenID Connect will also be considered.

2.1 SAML 2.0

The Security Assertion Markup Language (SAML) standard defines an XML-based framework for describing and exchanging security information applications. This security information is expressed in the form of portable SAML assertions that applications working across security domain boundaries can trust. The SAML standard defines precise syntax and rules for requesting, creating, communicating, and using these SAML assertions [16].

SAML consists of building-block components that, when put together, allow a number of use cases to be supported. The components primarily permit transfer of identity, authentication, attribute, and authorization information between autonomous organizations that have an established trust relationship. The core SAML specification defines the structure and content of both assertions and protocol messages used to transfer this information [4].

Key features of SAML:

- **Assertions**. The core of SAML, assertions are XML statements that assert one or more statements about a subject (a user). SAML assertions carry statements about a principal that an asserting party claims to be true. The valid structure and contents of an assertion are defined by the SAML assertion XML schema.

- **Protocols**. SAML defines a set of request-response protocols for securely exchanging assertions between parties. *Authentication Request Protocol* is used by a service to request authentication from an identity provider. *Logout Protocol* facilitates the process of logging out users across multiple systems simultaneously.

- **Bindings**. These are mechanisms that define how SAML protocol messages are transported within SOAP, HTTP, and other protocols. For example *HTTP Redirect Binding* that sends SAML messages through HTTP redirects or *HTTP POST Binding* that transmits SAML messages within the body of an HTTP POST request [2].

Security mechanisms

SAML implements several mechanisms to ensure the security of authentication and authorization data exchanged between the identity provider and the service.

Encryption   Sensitive data in SAML assertions can be encrypted to prevent unauthorized access. Typically, the identity provider encrypts the assertion using the service provider's public key.

Transport-Level Encryption (TSL)   SAML messages are often sent over HTTPS to ensure encryption during transmission.

Digital Signatures   Assertions are signed digitally using the private key of the identity provider to ensure their integrity and authenticity. The service provider verifies this signature using the corresponding public key.

Summary

SAML was primarily designed for web-based applications where the exchange of SAML messages occurs between the user's browser and web servers. Using various security mechanisms SAML ensures the secure exchange of identity information across trusted services. Using SAML for public clients, such as mobile applications or single-page applications (SPAs), presents certain challenges and limitations. It requires handling XML-based SAML assertions and responses, which can be heavy and challenging to parse in client-side environments.

2.2   OAuth 1.0 protocol

OAuth (Open Authorization) protocol was originally created by a small community of web developers from a variety of websites and other Internet services who wanted to solve the common problem of enabling delegated access to protected resources. The resulting OAuth protocol was stabilized at version 1.0 in October 2007, and revised in June 2009.

OAuth is an open standard for access delegation. In a simple words: user is granting application A access his own information from application B without sharing his credentials from B with application A. In general this protocol offers secure method for Internet users to allow third-party access to their resources without exposing their credentials [20].

Specification

Accordingly to the RFC 5849 specification[1], OAuth 1.0 allows making authenticated HTTP requests using token-based exchange. This protocol defines five terms that participate in this exhange:

- **Client** – HTTP client that is capable of making OAuth-authenticated requests.

- **Server** – an HTTP server capable of accepting OAuth-authenticated requests credentials - Credentials are a pair of a unique identifier and a matching shared secret. OAuth defines three classes of credentials: client, temporary, and token, used to identify and authenticate the client making the request, the authorization request, and the access grant, respectively.

- **Protected Resource** – An access-restricted resource that can be obtained from the server using an OAuth-authenticated request.

- **Token** – A unique identifier issued by the server and used by the client to associate authenticated requests with the resource owner whose authorization is requested or has been obtained by the client. Tokens have a matching shared-secret that is used by the client to establish its ownership of the token, and its authority to represent the resource owner [5].

Security

The security mechanisms proposed by the OAuth 1.0 standard:

Digital Signature   OAuth 1.0 requires digital signing of all requests, using a shared secret client key and token key. The HMAC-SHA1 signature algorithm is used to ensure that data has not been altered.

Nonce and Timestamp   Nonce, a random number generated by the client for each request, prevents requests from being reused. The timestamp ensures the relevance of the request, reducing the risk of a replay attack.

Transport-Level Encryption (TSL)   As with other authorization protocols, a secure HTTPS connection must be used to protect data and prevent man-in-the-middle attacks.

Summary

OAuth 1.0 was the first version of the OAuth protocol, designed to give third-party applications limited access to a user's resources without giving them a password. The protocol depended on the signature format, which made it difficult to adapt to different services. Authentication and authorization are not separated, leading to confusion in their management as well as the possibility of token transfer to a third party. Despite its limitations, OAuth 1.0 played an important role in the development of authorization standards, but its use today is not recommended due to the availability of more modern and reliable protocols.

---

[1] https://www.rfc-editor.org/info/rfc5849

2.3   OAuth 2.0 protocol

Because of some limitations and complexities in OAuth 1.0 there was a need to develop a new and more flexible protocol OAuth 2.0 that preserves OAuth 1.0 motifs. Second version simplifies client development and provides different authorization flows for different clients such as web applications, living room devices and mobile phones. The OAuth 2.0 protocol is not backward compatible with OAuth 1.0.
Improvements include:

- **Flexibility** – OAuth 2.0 supports multiple flows (authorization code, implicit, resource owner credentials, and client credentials), that makes it come in handy for different types of applications.

- **Security** – Obtaining of the access tokens was standartized, making it more adaptive for modern security.

Security

OAuth 2.0 includes several security mechanisms to protect data and provide secure delegation of access. It retains the security mechanisms used in OAuth 1.0, but also adds several new ones.

- **PKCE (Proof of Key for Code Exchange)** – security enhancement for providing an authorization code that mitigates code interception attacks. The client generates a secret (code verifier) and a hashed version (code challenge), providing both to prove their identity.

- **Scopes** – Scopes limit the level of access provided by the token. The client requests only the necessary access scopes, following the principle of least privilege.

- **Verifying token validity** – Resource servers can check token validity via introspection endpoints.

- **Audience Restriction** – Access tokens should be audience-restricted, ensuring that they are only used by the intended resource server.

- **Flows** – The exchange of tokens between a service and an authorization server can occur under different scenarios. The choice of scenario depends on the security level of the client as well as the authorization server's level of trust to the client.

Client types

OAuth 2.0 has two types of clients: *confidential* and *public*. The client type depends on the client's ability to keep credentials secure.

Confidential clients are able to handle their registered client secret in safe. Client implemented on a secure server with restricted access to the client credentials and can securely authenticate user with the authorization server .

Public clients are not able to handle their registered client secret/user's credentials in safe. Examples of public clients are mobile or single-page applications, JavaScript-based web applications, and native application [6].

Accordingly to the RFC 6749[2] this specification has been designed for the following client profiles:

- **Web Application** – A web application is a confidential client running on a web server. Resource owners access the client via an HTML user interface rendered in a user-agent on the device used by the resource owner.

- **User-Agent-Based Application** – A user-agent-based application is a public client in which the client code is downloaded from a web server and executes within a user-agent (e.g., web browser) on the device used by the resource owner. Protocol data and credentials are easily accessible (and often visible) to the resource owner.

- **Native Application** – A native application is a public client installed and executed on the device used by the resource owner. Protocol data and credentials are accessible to the resource owner. It is assumed that any client authentication credentials included in the application can be extracted.

Token types

OAuth 2.0 standard uses three types of tokens. They are used in communication between the client, server and authorization server.

- **Authorization code grant** – The client will get the authorization code from the authorization server during 'Authorization Code Grant' flow. Client exchanges an authorization code for an access token then.

- **Access token** - String that is being used by client to make requests to the protected resource server.

- **Refresh token** - String that is being used by client to get a new access token.

Authorization code   The authorization code does not contain any information and cannot be decrypted by the client. It is generated by the authorization server and sent in response to the client after the user has allowed the client access. The generation of the code depends entirely on the authorization server, in most cases it is a random string of ASCII characters that the server stores in its database.

Access token

An access token is a type of credentials that is issued by the server in response to an authorization code and must be used to access protected resources. An access token may have an expiration time that is controlled by the authorization server. When the token expires, the client can request the next one using a refresh token 2.3. Access tokens may be either *bearer tokens* or *sender-constrained* tokens. A Bearer Token is an opaque string, not intended to have any meaning to clients using it [11].

Sender-constrained tokens require the OAuth client to prove possession of a private key in some way in order to use the access token, such that the access token by itself would not be usable [10].

---

[2]https://www.rfc-editor.org/info/rfc6749

Refresh token

A refresh token is very similar to an access token. The only difference between them is the purpose - the refresh token is used to get a new access token. Accordingly, when the access token expires, the client should not repeat the authorization procedure again, but use the refresh token and get a new access token.

Flows

To configure our client and server applications, we need to consider the types of flows and select the one we need.

Authorization Code Grant with Proof Key for Code Exchange

One of the Flows that can be implemented is the Authorization Code Grant Flow. It is well suited for confidential clients. But within the scope of this paper we will consider its improvement - Authorization Code Grant with Proof Key for Code Exchange (PKCE). This flow is suitable for public clients and can be used in an insecure environment.

The PKCE-enhanced Authorization Code Flow introduces a secret created by the calling application that can be verified by the authorization server; this secret is called the Code Verifier. Additionally, the calling app creates a transform value of the Code Verifier called the Code Challenge and sends this value over HTTPS to retrieve an Authorization Code. This way, a malicious attacker can only intercept the Authorization Code, and they cannot exchange it for a token without the Code Verifier.

Client should be capable of interacting with the resource owner's user-agent and receiving incoming requests from authorization server [8]. Authorization Code Grant diagram can be found in appendix A.

The following are the steps involved in the Authorization Code Grant with PKCE flow:

1. User proceeds to login.

2. The client creates a cryptographically-random `code_verifier` and from this generates a `code_challenge`.

3. Client redirects the user to the identity provider (authorization server) along with the `code_challenge`.

4. Identity provider redirects user to the login and authorization prompt.

5. The user authenticates using one of the configured login options and may see a consent page listing the permissions identity provider will give to the application.

6. Identity provider stores the `code_challenge` and redirects the user back to the application with an authorization code, which is good for one use.

7. Clint sends this code and the `code_verifier` (created in step 2) to the Identity Provider.

8. Identity Provider verifies the `code_challenge` and `code_verifier`.

9. Identity provider responds with access token (and optionally, a refresh token).

10. Client can use the access token to call an API to access information about the user.

The authorization code grant with PKCE is one of the most common and secure flows in OAuth2.0 protocol specification.

Resource Owner Password Credentials Grant

The resource owner password credentials grant type can be applied in cases where resource owner trusts the client and can give him his username and password [6].
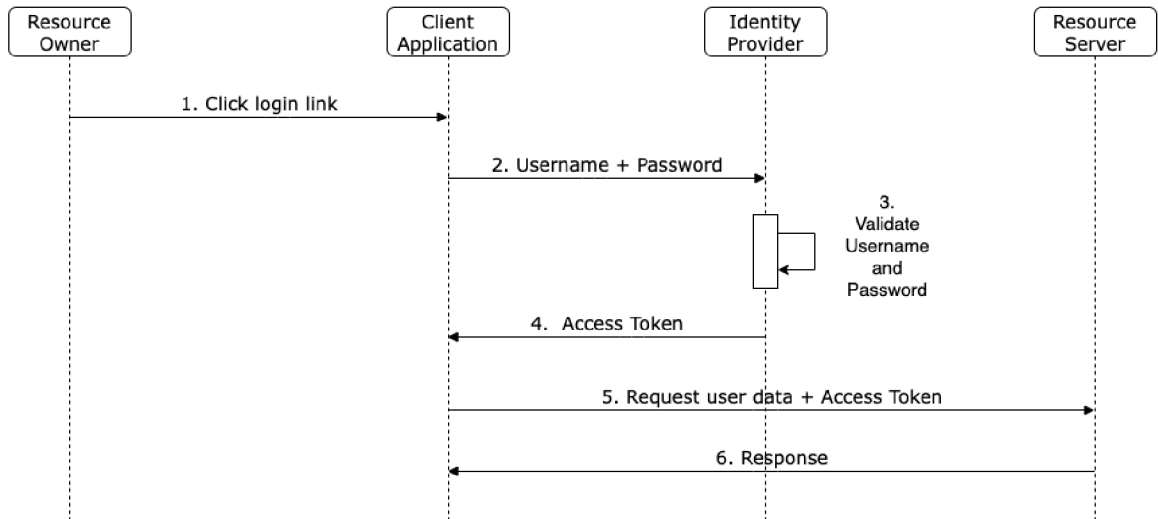
Figure 2.1: Resource Owner Password Credentials Grant.

1. User proceeds to login.

2. Client requests an access token from the authorization server using user's credentials.

3. Authorization server authenticates the client, validates user's credentials.

4. Authorization server sends access (and optionally, a refresh token) back to the client.

5. Client can use the access token to call an API to access information about the user.

6. Resource Server responds with requested data.

Client Credentials Grant

The client credential grant flow can be used when the client obtains a token using its own credentials. Credentials in this flow is a secret that must be known to both the authentication server and the client.

1. The client authenticates with the authorization server and requests an access token from the token endpoint.

2. Authorization server authenticates the client.

3. Authorization server issues an access token.

4. Client can use the access token to call an API to access information about the user.

5. Resource Server responds with requested data.

11
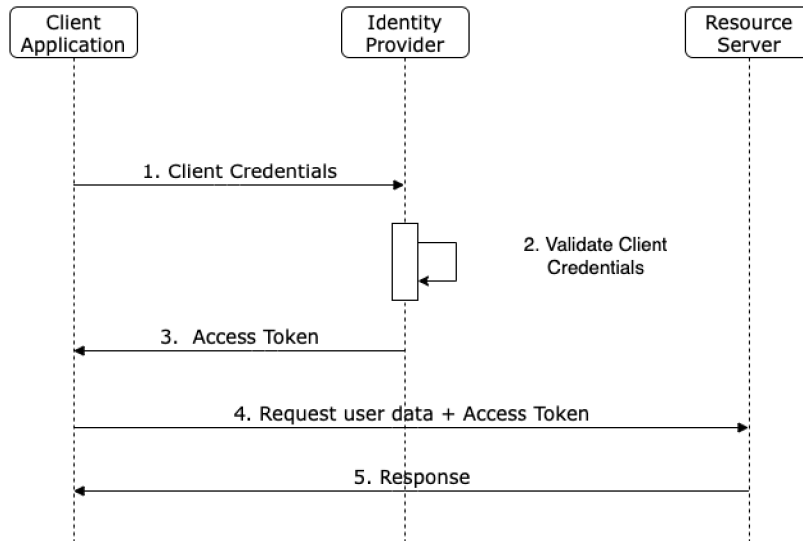
Figure 2.2: Client Credentials Grant.

Claims

In case an Access Token is encoded as JSON Web Tokens (JWTs) [10], it may contain a set of claims. Claims are pieces of information that are contained in the access token payload and may contain details about the user, session, authorization server or permissions. The number of claims in a token depends on the configuration of the authorization server and the Scopes that the client wants to receive.

Often an Access Token contains a set of standard claims such as:

- **Issuer (iss)** – Specifies the issuer of the access token. It represents the authorization server that issued the token.

- **Expiration Time (exp)** – indicates the expiration time of the access token. After this time has passed, the token should no longer be considered valid.

- **Token Identifier (jti)** – unique identifier for the access token.

In addition to the standard tokens, a token may contain custom tokens. As mentioned earlier, the authorization server can configure the content of the tokens and create new ones. For example, the authorization server can add a custom token that contains the role of the user who received the token. It can name it `role` and use a mapper that will add the `role` value [3].

Claims example:

```
{
  "iss": "https://authorization-server.com/",
  "exp": 1637344572,
  "jti": "1637337372.2051.620f5a3dc0ebaa097312",
  "role": "admin"
}
```

Scopes

Scopes in OAuth 2.0 is a mechanism for the client to request specific claims to be included in the access token by the authorization server. Users must understand what level of access they are granting to the client. In turn, clients should only access specific user and session information, not all of it. The client sets the Scopes it needs before it begins authorizing the user. After the user is authorized, the user sees a list of scopes that the client requires from the authorization server. The client can then accept them and access will be granted, or refuse [11].

Problems

The main drawback of this protocol is that it does not authenticate the user. It gains delegated access to the user's information. The motivations of authentication and authorization participants are different. In the case of authorization, the client is entrusted with an access token, as he does not intend to share it with anyone. But the client can still share the access token with a third party and give them access to the information (protected resource). The user and the authorization server cannot naively assume that the client will not share information with a third party after the token is issued. Thus any site to which a user logs in with a Google or Facebook account can impersonate that user on any other site that accepts Google or Facebook logins.

2.4   OpenID Connect Protocol

OpenID Connect is a simple identity layer on top of the OAuth 2.0 protocol. It enables clients to verify the identity of the user based on the authentication performed by identity provider, as well as well as to obtain basic user profile information [17]. This protocol extends OAuth 2.0 by adding another token to the process - ID Token. It is encrypted in JSON Web Token (JWT)[10] format and contains information about the user's authentication status and profile information.

Core components of OIDC:

- **ID Token** - contains encoded information about the user's authentication. such as the time of his authentication, who issued the token, and user information.

- **Discovery** - A provider supporting the OIDC protocol publishes an endpoint with the configuration. This in turn facilitates client configuration. The client can for example verify the token using the published information.

- **UserInfo Endpoint** - The provider also publishes an endpoint that returns stigmas about the authenticated user.

Security

The use of OIDC also includes defense mechanisms, such as the use of a `nonce` parameter to prevent a replay attack. It is also possible to use an `at_hash` parameter in the ID token, which is needed to validate the access token. Both parameters are stored in the token as claims 2.3.

Single Sign On

Since the user session is centralized and its information is stored in the identity provider and can be transmitted using the token ID in a secure manner, a single sign on is possible. Single Sign On is the process of user authentication using a single set of credentials for many applications at once. To implement it, a single centralized provider is needed to store all user session information and a protocol that can securely share session information with other applications. This mechanism is often found in large companies where employees need access to many applications simultaneously or during a working day. In order to avoid the risk of an employee password leak in one of the applications, a single identity provider is used that manages the sessions and the employee account. The application no longer needs to store information about session, taking a risk of this data being leaked. It is enough to contact the identity provider, which will create a session in case of its absence or transfer information about an existing one.

Flows

OpenID Connect defines three types of authentication flows for different clients: the Authorization Code Flow, the Implicit Flow and the Hybrid Flow.
In this work will only be considered the first one, the Authorization Code Flow, and more specifically its enhancement - Authorization Code Flow with PKCE A, will be discussed. It is essentially identical to the flow of the same name from OAuth 2.0. The only difference is that in addition to the **access token**, the identity provider also issues an **ID Token** to the client in response to the authorization code, which can then be verified. The client can also use the *UserInfo* point to retrieve new information about the user.

Summary

OpenID Connect (OIDC) issues ID tokens, which contain user identity information in a secure, JSON Web Token (JWT) 2.5 format. It also provides mechanisms for single sign-on (SSO), making it easier for users to access multiple applications. By incorporating features like scopes, claims, and standard user info endpoints, OIDC offers a flexible, interoperable solution that simplifies identity management and enhances security for web, mobile, and desktop applications.

2.5   JSON Web Token

JSON Web Token, or JWT for short, is a standard for secure transmission of claims in insecure environments in JSON format. The main features of its architecture are compactness, simplicity, usability and security. The token format, signature and encryption capabilities, and ease of use provide these features. Although much more complex systems are still in use, JWTs have a broad range of applications.

Format

A JSON Web Token looks like this (newlines inserted for readability):

```
eyJhbGciOiJIUUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

In its compact form, JSON Web Tokens consist of three parts separated by dots ("."). These three parts called Header, Payload and Signature are arranged one by one as `"aaaaaa.bbbbbb.cccccc"` where Header, Payload are encoded in `Base64Url` format [7] and the last one is the signature which will be described later.

An example of an unencoded token:

```
{
    "alg": "HS256",
    "typ": "JWT"
},
{
    "sub": "1234567890"
    "name": "John Doe",
    "admin": true
}
```

Header and Payload contain **claims** 2.3. Some of these claims and their meaning are defined as part of the JWT spec. Others are user defined. One of the features of JTW tokens is the standardisation of claims that can be used in an application. Some of the standardised claims are discussed in section 2.3. Another key aspect of JWTs is the possiblity of **signing** them using JSON Web Signatures (JWS, RFC7515[3]), and/or **encrypting** them, using JSON Web Encryption (JWE, RFC7516[4]). Together with JWS and JWE, JWTs provide a powerful, secure solution to many different problems [18].

JSON Web Signatures

There are several types of signing algorithms available. The JWS specification requires a single algorithm to be supported by all conforming implementations:

- HMAC using SHA-256, called **HS256** in the JSON Web Algorithms specification[5].

The JWS specification also defines a series of recommended and supported algorithms such as:

- RSASSA PKCS1 v1.5 using SHA-256, called **RS256**

In this study we will look at using the RS256 algorithm. The difference between it and HMAC (which uses shared secret) is the use of public and private RSA key pairs [9].

In the course of its work the signature is performed using the private key and the further validation is performed using the public key. Thus `base64Url` encoded *Header* and *Payload*

---

[3]https://datatracker.ietf.org/doc/rfc7515
[4]https://datatracker.ietf.org/doc/rfc7516
[5]https://datatracker.ietf.org/doc/rfc7518

as well as private key are inputs for RSA signing function. The signature output is also encoded in base64Url format and concatenated to the *Header* and *Payload*, resulting in the JWT token structure 2.5.

JSON Web Encryption

While JSON Web Signature (JWS) provides a means to validate data, JSON Web Encryption (JWE) provides a means to make data opaque to third parties [12]. Opaque in this case means unreadable. Encrypted tokens cannot be verified by third parties. This allows tokens to be used more freely, given that only trusted resources will be able to retrieve the information. At the same time, this makes it difficult to implement JWT in public clients, since public clients cannot store any secrets or private keys on their side. The use of encryption implies a secure environment and confidential clients.

The shared secret scheme works by having all parties know the **shared secret**. Each party in possession shared secret can both **encrypt** and **decrypt** the information.

While in JWS, the party holding the private key can sign and verify tokens, while the parties holding the public key can only verify those tokens. In JWE the party holding the **private key** is the only party that can **decrypt** the token [18]. In other words, **public key** holders can **encrypt data**, but only the party holding the **private key** can **decrypt** (and encrypt) that data.

Since this project will demonstrate the implementation of public clients, encryption will not be used.

Encryption algorithms recommended by the JWA specification[6]:

- **RSAES-PKCS1** (marked for removal of the recommendation in the future).

- **RSAES-OAEP** with defaults (marked to become required in the future).

- **AES-128** and **AES-256** Key Wraps.

- **Elliptic Curve Diffie-Hellman Ephemeral Static (ECDH-ES) using Concat KDF** (marked to become required in the future).

- **ECDH-ES + AES-128 or AES-256** Key Wrap.

JWTs and OAuth 2.0

JWTs are well suited to the requirements of the OAuth 2.0 protocol 2.3. Signed JWTs are good **access tokens** because they can be encoded with all the necessary data to distinguish access levels to a resource, can have an expiry date, and are signed to avoid validation queries against the authorisation server. Several federated identity providers issue access tokens in JWT format. JWTs can also be used for **refresh tokens**. However, there are fewer reasons to use them for this purpose.

JWTs and OpenID Connect

JSON Web Token is also used in OpenID Connect 2.4. This data format is used to transfer the ID token. Using this format allows authenticated user data to be quickly transferred to clients in a signed and encrypted format. In this way the client can verify that the user session data has not been altered and contains legitimate information.

---

[6]https://datatracker.ietf.org/doc/rfc7518

Chapter 3

Introduction to Identity Management

This chapter will describe the theoretical part related to the identity management topic. The concept of identity management and why it is necessary for the implementation of single sign-on mechanism will be discussed. Since the purpose of this paper is to use Keycloak as an identity provider it will be discussed why it is used in the project, the concepts of how it works, and how to integrate it into the application.

3.1 Definition

Identity management encompasses the management of individual identities and their authentication, authorization, roles, and privileges and permissions within or across system and enterprise boundaries, with the goal of increasing security and productivity while decreasing cost, downtime, and repetitive tasks. Identity management thus constitutes an essential capability for attaining trusted clouds [21].

3.2 Identity Providers

Identity Providers (IdPs) play a key role in identity management by authenticating the identity of users and providing this information to service providers. This process, utilizing standards such as `OpenID Connect`, allows service providers to provide appropriate access to users based on authenticated identities without directly managing user credentials.

3.3 Single Sign-On

IdM systems provide a centralized platform for managing user identities and attributes across multiple systems and applications. This centralization is essential for single sign-on since it allows users to access multiple services with a single set of credentials, simplifying the login process and reducing the cognitive load on users. Centralization ensures that user identity information is consistent across all systems. Any changes, such as updating a user role or changing a password, are propagated across all services, improving security and operational efficiency. In an SSO scenario, the IdP is responsible for validating user credentials and issuing authentication tokens that other applications can trust. This centralized

approach not only improves user experience and productivity by reducing the number of times users need to log in, but also strengthens security in the organization's environment.

3.4 Introduction to Keycloak

Keycloak is an open source Identity and Access Management (IAM) tool licensed under the Apache License 2.0. There is also a downstream project called **RedHat SSO**. It can be used as a centralized identity manager allowing you to store user data, manage user roles, groups, sessions. Keycloak also allows you to categorize users into two different realms. For example, when writing an application for a bank, we can separate users and employees into two different realms to simplify management. There is support for three protocols - OpenID Connect, OAuth 2.0 and SAML 2.0.

3.5 Why to use Keycloak?

There are plenty of alternatives such as Gluu, FreeIPA, WSO2, FusionAuth, and Ory Hydra. Although their feature sets are quite similar, some of them will work well in specific use-cases. This paper focuses on Keycloak because it has the ability to be deployed in cloud environments, has a huge community and a large number of configuration guides and books. Also a lot of features are supported out of the box which makes it easy to configure. The most important thing to note is that it can be deployed with Docker or Kubernetes, which is certainly important in today's container world. Keycloak supports different kinds of encryption methods and has built-in features such as Brute Force Detection.

3.6 Distributions of Keycloak

There are currently three types of Keycloak server distributions available:

- Keycloak on Quarkus.

- Keycloak on Docker.

- Keycloak Operator for Kubernetes and OpenShift.

Keycloak Operator also supports clustering, so it is possible to create a number of Keycloak servers that run as a single identity center, with Kubernetes distributing the load between the servers.

3.7 Keycloak Integrations

How do you integrate keycloak into your application? It is possible to integrate Keycloak into almost any popular programming language such as Java, Python. C#, Scala and frameworks like Quarkus, Angular and so on. Integration into Java is done using the Spring Boot framework library. Until 2022, integration into Java was done using special adapters such as Keycloak Spring Boot Adapter or WildFly Subsystem Adapter. But in 2022 developers announced deprecation of adapters. They were replaced by more flexible libraries such as `keycloak-core`.
The transition from adapters to libraries unified work with all kinds of Java applications

and no longer tied developers hands by developing a separate adapter for each framework. A shift from adapters to libraries replaced the approach to integrating Keycloak into an application. The architecture of libraries is different from Keycloak adapters. It is more oriented to work with standard protocols such as OAuth 2.0 and OIDC.

3.8   Social Login

Keycloak also allows you to use various social identity providers such as Google, Twitter, Facebook, Github and so on. Their use can be configured in the admin panel.

3.9   Keycloak Identity Management

Keycloak offers a web-based user interface where developer can configure access or user settings. There is a possibility create groups, roles, assign access to specific clients, or set the password complexity level.

3.10   Realms

Once you have an administrative account for the Admin Console, you can configure realms. A realm is a space where you manage objects, including users, applications, roles, and groups. A user belongs to and logs into a realm. One Keycloak deployment can define, store, and manage as many realms as there is space for in the database [15].

3.11   Clients

Clients are entities that can request Keycloak to authenticate a user. Most often, clients are applications and services that want to use Keycloak to secure themselves and provide a single sign-on solution. Clients can also be entities that just want to request identity information or an access token so that they can securely invoke other services on the network that are secured by Keycloak [15].

3.12   Keycloak Admin API

Keycloak provides the ability to manage users in a RESTful manner by sending requests to an API. This requires configuring a special admin account that will have access to the requests. For example, you can query all database users by sending GET request to "`/admin/realms/realm/users`". It will return a stream of users, filtered according to query parameters [13].

3.13   Conclusion

The choice of an Identity and Access Management (IAM) solution depends on the use case and project requirements such as security, protocol support, and extensibility. Keycloak provides a large number of features, fine-grained and at the same time easy customization, a large community and ease of deployment. It is well suited for the role of identity manager based on OpenID Connect protocol and SSO implementation.

Chapter 4

Requirements

This section will cover all the requirements for this thesis assignment. This section includes parts that must be fulfilled in full, but there will also be parts that are only partially fulfilled. In this project a prototype application will be developed that will not be used and deployed in the real world. The programs developed under this thesis are of an observational nature and serve as a **showcase** for other developers who wish to learn the protocols and tools used to implement them in applications. In general, there are two parts to all requirements. One part describes functional requirements and the other part sheds light on non-functional requirements.

4.1 Functional Requirements

The application's functionalities and behaviour are described in the functional requirements. It is the application's bare minimum need. The project cannot be completed without those prerequisites.

User Authentication and Login

The user authentication procedure is as follows: the user accesses the base URL of the application in the browser. To authenticate the user, the user will be sent to the authentication page of the authorization server. To authenticate, the user must enter their username and password. If the user is authenticated, the user will be logged in and will be able to view the secure application home page. If authentication fails, the user will receive an error message stating that the credentials are incorrect. Without authentication, the user will not be able to access any of the application's protected resources.

Authorization

To specify that a group of users has the right to do or see certain things in the application, the system uses a customizable authorization restriction structure. Without proper authorization, a user cannot perform actions or access protected application resources. There should be functions that only certain types of users with proper authorization can perform. For example, a system user is authorized to view the application home page. However, he/she will not be able to modify any protected data due to lack of authorization. On the other hand, an administrator can create and edit existing application functions.

Single Sign-On

After the user has authenticated, his session should be stored with the identity provider. Further, if he has an active session in any of the client applications he should be able to authenticate in another application without having to re-enter his credentials. His session information should be available in any of the applications. The session and user data must be identical in both client applications.

Separation of Access Tokens

Each client is configured such that the identity provider issues different access tokens for each client. Thus the access tokens for client application A must be different from the access token for client application B. In case a user has gone through the authenticated process and received an access token for service A, his access token cannot be used to make a request to server B and vice versa.

Single Logout

The user must be able to logout of the system if they want to. The user must first log in before attempting to log out. When a user logs out, they will not be able to access protected application resources until they log in again.

Cloud Deployment

Both web clients, Keycloak as well as two servers should be run in a cloud-native environment and tests should be conducted based on the above requirements. The applications must be easily scalable, comply with cloud application standards such as endpoints for health checks and sessionless operation.

Chapter 5

Design

This chapter describes the design procedures for all applications needed to demonstrate and implement the OpenID Connect protocol, the Keycloak connection as identity manager and the single sign-on mechanism itself.

5.1 System Overview

A prototype of four applications was developed to demonstrate this project. Two of them act as a client web application, the other two respectively play the role of a server for them. The first application works as a user portal, the second one is a note-taking application. **Customers Application** and **Notes Application** for the notes application. The block architecture of the entire prototype is indicated in 5.1.
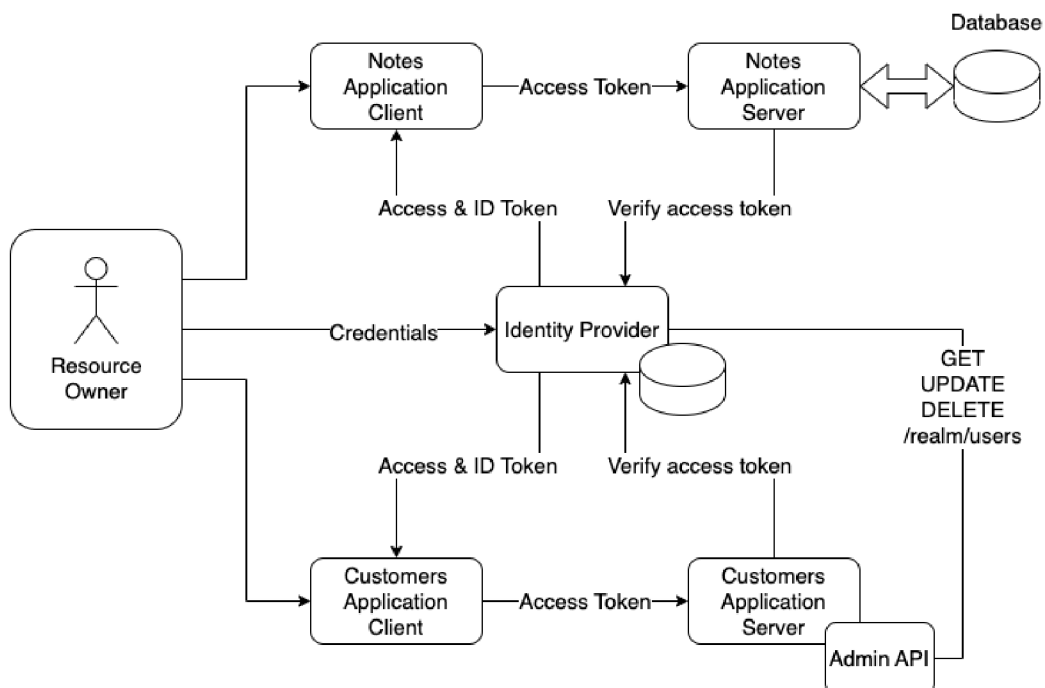

Figure 5.1: System block overview.

In this architecture, the user is a resource owner. He uses client applications to obtain resources from resource servers. In this case, customers application and notes application access separate servers. Both clients and both servers are standalone applications and have their own port. Communication between them takes place via HTTP protocol. This architecture was chosen in order to separate the access tokens. Accordingly, a token received in the customers Application **cannot be used** to access the notes application server and vice versa. Both client applications redirect the user to the identity provider endpoint, where the user can be authenticated using his credentials.

There are two resource servers, **Customers Application Server** and **Notes Application Server**. Both are RESTful Spring Boot applications. Notes application server has a database connection and API that supports CRUD operations. Customers application server uses the Keycloak Admin Client to retrieve data about all users using admin api 3.12. In fact, you can manage users directly using Keycloak as a user data manager. But in the real world, more flexible LDAP-based systems are often used.

5.2   System Use Cases

Since the main point of the work is to configure application security and demonstrate the use of authentication protocol, there are not many use cases in general. There are four types of users: anonymous, user, admin and superadmin. The super admin role is used to configure the use cases and assign administrators to the application. An unauthenticated user, anonymous, has the ability to login or register. The user can also create an account using Google or GitHub.

In Customers Application, an authenticated user can only get the full list of users. Admin has full permissions in the application and can create, delete, modify any users. Customers application use case diagram can be found in appendix G

In Notes Application, an authenticated user can manage their notes and create new notes. Admin has full rights in the application and can retrieve, create, delete, modify any notes. Notes application use case diagram can be found in appendix F

5.3   Functional Architecture

The functional architecture of this project describes how clients access user information on resource servers. There are 6 objects involved in the process. The user is the owner of the resource and communicates with clients through a browser. Identity provider in the system is the Keycloak server. Client application A and B are Notes application and Customer application respectively. Both clients configured with Authorization Code Grant with Proof Key for Code Exchange, its detailed description can be found in section 2.3. Functional architectire diagram can be found in appendix E.

After the user presses the login button in client application A, it starts the authentication process. Next, the client application sends a request for an authorization code to the identity provider. The identity provider redirects the user to its endpoint, where the user authenticates. After successful authentication, the user is redirected back to the client page and the client receives access token A along with the ID token. The user's browser stores the session information. The client can then request data from server A's resource. The resource server in turn validates the access token using the server's identity provider and

decodes it. The server then finds the claim `aud`, which contains information about which client the token was issued to. If the `aud` field contains the value `client-application-a`, the server sends a successful response to the client along with the necessary data.

The user can also go to client-application B and click the login button. The client will then try to find the session information in the browser cookie. If the cookie is found, the client authentication is successful without entering user credentials. The client also requests the access token B from the identity provider of the server. In case the user session is still active, it issues the access token B to the client. The client can query the data resource of server B using the token. The process of token validation is similar. First, the server validates the access token using the server's identity provider and decodes it. If the `aud` field contains the value `client-application-b`, the server sends a successful response to the client along with the data it needs.

5.4   Keycloak Clients and Realm Configuration

By default, the root realm in Keycloak is called the `master` realm. In it, a child client called `demo` will be created. In this realm will exist users from all the applications, their roles and clients. There will be a total of two clients in the realm. One client for customers application and one for notes application.
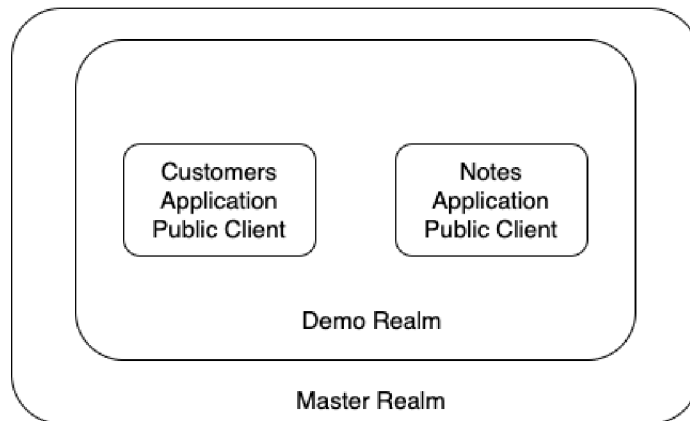

Figure 5.2: Keycloak Configuration Diagram.

5.5   Database Design

There is only one note database object. Primary key is the *id* field of type *Long*. The *content* field is of type *String* and contains note data. The *owner* field contains the username of the user who created the note. The username of the user in Keycloak is immutable, so it can be used for the database.

Chapter 6

Implementation

6.1 Used tools and libraries

This section provides an overview of the main tools and technologies selected for application development. Each tool was selected for its specific features that meet the needs of developing, managing, and deploying a modern scalable web application.

Angular

Angular is a powerful front-end web application platform led by the Angular Team at Google and a community of individuals and corporations. Angular was chosen for its robust data binding capabilities, extensive HTTP communication tools, and its material design interface components which provide a sleek, modern UI. Angular also supports the modular organization of functionality through services and dependency injection, streamlining the development of large-scale applications. This project will use Angular 17.3.6.

Angular CLI

Angular Command Line Interface (CLI) is a tool that simplifies the development of an Angular project. It allows developers to create different application modules and configure them automatically. The decision to use Angular CLI in this project was driven by its efficiency in handling routine tasks, allowing more focus on business logic and feature development. The version of Angular CLI used is 17.3.6.

Angular OIDC Library

`angular-oauth2-oidc` library [19] is required to handle authentication and authorization through OpenID Connect (OIDC) and OAuth 2.0 protocols. It abstracts away the complexity of implementing mechanisms implementing these protocols, but also provides a clear and flexible interface to work with them. This includes managing the details of token acquisition, renewal, and expiration seamlessly. `angular-oauth2-oidc` adheres to the OAuth 2.0 and OIDC specifications, ensuring that implementation is compliant with these standards, which is crucial for interoperability with various identity providers. It supports various OAuth 2.0 flows and features like the Authorization Code Flow with PKCE, which is recommended for web applications due to its enhanced security properties. Using this library allows you to customize application security regardless of the type of identity provider used in the overall architecture. Thus, the built application will be able to interact not only

with Keycloak, but also with other identity providers. In summary, using angular-oauth2-oidc in this project helps streamline authentication processes, ensures adherence to security standards, and enhances the overall security and maintainability of the application.

Spring Boot

Spring Boot is an extension to the Spring framework that simplifies the writing and configuration of Spring applications. It was chosen for its ability to rapidly set up standalone, production-grade Spring based applications with minimal Spring configuration. Spring Boot's embedded server, configuration, and wide range of starters make it an ideal choice for building microservice architectures. This project will use Spring Boot 3.2.5.

Spring Security

Spring Security is a powerful and highly customizable authentication and access-control framework. It is the de-facto standard for securing Spring-based applications [1].

Spring Data JPA

Spring Data Java Persistence API (JPA), part of the larger Spring Data family, makes it easy to easily implement JPA-based repositories. It greatly improves the implementation of data access layers by providing clear and flexible interfaces and beans.

Maven

Apache Maven is a build automation and project management tool used primarily for Java projects. It helps to manage the dependencies, build, configure and deploy Java applications. Its use of Project Object Model (POM) files ensures that project setups are simple and reusable. Apache Maven Apache Maven 3.9.6 was used for this project.

Minikube

Minikube is chosen for its ability to run Kubernetes locally on a personal computer, providing a platform to test and develop Kubernetes-based applications without the overhead of setting up a Kubernetes cluster. It is well suited for this project because it is designed for development and testing, allowing developers to see how applications would behave in a production-like environment on their local machine. The minikube version: v1.32.0 was used, as well as the `hyperkit` driver.

6.2   Identity Provider Setup

To begin, the configuration of the Keycloak server will be discussed. For this project, the keycloak build for the container is ideal. This project will use the latest version of Keycloak 24.0.3.

The Identity Provider must be configured according to the diagram 5.2. That is, a child realm must be created in which clients must be configured for each of our client applications. Roles and users for testing and administration must also be created. The clients configuration will allow the client applications to be connected to them and split

into separate configurations. In this way it will be possible to manage the information contained in the tokens issued for a particular client.

To start the server with the docker it is required to use the command:

```
docker run -p 8080:8080 \
    -e KEYCLOAK_ADMIN=admin \
    -e KEYCLOAK_ADMIN_PASSWORD=admin \
    quay.io/keycloak/keycloak:24.0.3 start-dev
```

By default, Keycloak starts on port 8080. With the `-p 8080:8080` option it is `hostPort:containerPort`. It is also required to set a username and password for the superadmin who has access to the „master" realm. After a successful launch there is an opportunity to go to Keycloak Admin Console at the link `http://keycloak:8080` and login using admin username and password specified earlier.

Create New Realm

Best practice is to create child realms from the master realm. This way it is possible to separate users of different projects. After successful login using admin's credentials using the drop-down menu of realms in the upper left part of the window it is possible to create a new realm. It is required to enter the name of a new realm and to enable it. There is also an option of importing a realm in JSON format, which will be described later. The realm for this project will be called `demo`. The screenshot of the realm creation is in the apendix part of the work, which can be viewed here B.

Create New Client

The following section describes how to create two public clients for client applications in the **Clients** section. When configuring the client in the **Access Settings** section, the *Valid redirect URIs* and *Web origins* parameters must be set to service URL. The first parameter means a valid URI pattern to which the browser can redirect the user after a successful login. This protects the user from being redirected to a malicious URI after a successful login, as the client application can choose the redirect url itself. The Web origins parameter defines the allowed CORS sources.

Next, *Standard flow* and *Implicit flow* should be enabled under **Capability config**. This will allow the client application to use Authorization Code Flow with PKCE and to retrieve session information from browser cookies in case a session already exists.

Thus, `notes-public-client` and `customers-public-client` clients will be created.

The screenshot of the client creation can be viewed here C.

Custom Scopes and Claims

To separate the access tokens for the two microservices, we need to create a custom scope for each of them. The developer should create a new scope named `notes-app` and `customer-app`. Then assign a custom requirement to each scope. This can be implemented using Keycloak mappers. A mapper is a mechanism that allows us to automate the assignment of statement values to our scopes. For example, a mapper can be used to assign a user's date of birth to an access token assertion. To create a custom mapper, go to *Clients scopes -> Client scope details -> Mappers -> Add mapper*. Then we need to

specify the mapper type, name, its value, and choose which token it will be bound to. In our case, the mapper type will be `Audience`. In the **Included Client Audience** field, there is a need to select the name of the client that will be included in the value of the assertion. In case of a custom scope named `notes-app`, it is required to select the name of the client `notes-public-client` created in the previous step. This operation should also be performed for the `customer-app`. In the **Included Client Audience** field specify `customer-public-client`. The screenshot of the mapper creation can be viewed here D.

As a result of these actions, two scopes `notes-app` and `customer-app` were created, each of which stores the name of the corresponding client in the `aud` field.

The next step is to add the created scopes to the client's default scopes so that they are displayed in the payload of access token. To do this, go to *Clients -> Client details -> Clients scopes -> Add client scope* and select the newly created `notes-app` scope. The same operation should be repeated for the `customers-public-client` client. This is how the access token payload looks like after these operations:

```
{
  "exp": 1714926394,
  "iat": 1714926094,
  "jti": "bb39fd03-3ce4-4b31-9f46-536f2fde604a",
  "iss": "http://keycloak:8080/realms/demo",
  "aud": [
     "notes-public-client",
     "account",
  ],
  "sub": "886b8e5f-0220-4fbb-ab97-4263ef23b9d9",
  "typ": "Bearer"
}
```

Create Roles

In Keycloak, it is possible to configure roles on a realm or client scale. Thus, it is possible to separate the roles of different application users for more efficient management. Roles can be assigned to specific groups of users or to individual users. As part of the configuration of this project, an administrator role will be created within the „demo" realm. Thus the administrator role will be displayed in any access token issued to this user. User permissions depending on the role can be configured in the application that receives the access token. In order to create a role, it is necessary to go to *Realm roles -> Create Role* and assign the value `ADMIN` to the Role name field.

Create Users

After creating the necessary roles, the first users should be created. Within the framework of this project three users were created: admin, user, realm-admin. The user admin was assigned the role `ADMIN` created in the previous step. The user with username **user** has no roles and is similar to the one that will be registered in the client application. The user **realm-admin** was assigned the role `realm-management:realm-admin`. This user is required to use the Keycloak Admin API 3.12 in the Customers Application.

6.3 Customers Application Server Setup

This section will describe the most important parts and interesting details of the server implementation for the Customers Application. The server will be a standalone microservice and will communicate with the client via HTTP protocol via REST API. Then it should be configured as a Resource Server in terms of OAuth 2.0. The resource server will be receiving requests from applications with an HTTP Authorization header containing an access token. The resource server needs to be able to verify the access token to determine whether to process the request.

Security Config

The first thing to do is to protect the server from unauthorized access. For this purpose, the class `SecurityConfig` with annotations `@Configuration` and `@EnableWebSecurity` was created. This will make the class configurable and also allow to configure the security behavior using the `SecurityFilterChain` bean. This code snippet contains the configuration of the bean:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http)
        throws Exception {

    http.csrf(t -> t.disable());
    http.authorizeHttpRequests(authorize ->
                    authorize
                            .requestMatchers(ALLOWED_PATTERNS)
                            .permitAll()
                            .anyRequest()
                            .authenticated());

    http.sessionManagement(session -> session
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS));

    http.oauth2ResourceServer(oauth2 -> oauth2.jwt()
        .decoder(jwtDecoder()));
    return http.build();
}
```

Requests matching patterns in *ALLOWED_PATTERNS* are allowed without authentication. Within this application, healthcheck and OpenAPI endpoints are in the list of allowed patterns.

All other requesters are required to provide an access token.

The `sessionManagement()` method configures the session management policy to be stateless. This means that each requester must go through the authentication process independently and the server will not store information about its session.

The `oauth2ResourceServer` method designates the server as an OAuth 2.0 Resource Server and assigns a custom JWT decoder, which will be discussed next.

A CORS filter was also used, the code of which is taken from public source[1].

Access Token Validation

This code section shows the configuration of the JWT token decoder.

```java
public NimbusJwtDecoder jwtDecoder() {
    NimbusJwtDecoder jwtDecoder =
            JwtDecoders.fromOidcIssuerLocation(issuer);

    jwtDecoder.setJwtValidator(new DelegatingOAuth2TokenValidator<>(
            JwtValidators.createDefault(),
            new AudienceValidator(audience)
    ));
    return jwtDecoder;
}
```

This decoder is specifically set up for validating JSON Web Tokens in a Spring Security OAuth 2.0 resource server environment.

Instantiating a Jwt Decoder

The decoder is initialized using the factory method `fromOidcIssuerLocation()`. Thus the JWT decoder will automatically configure itself using the OpenID Connect discovery document published at a URL derived from the `issuer`. The issuer variable contains the URL of the Keycloak realm as *http://keycloak:8080/realms/demo*. This way the decoder can find an endpoint [14] containing public keys enabled by the realm, encoded as a JSON Web Key (JWK), and the Keycloak server signature algorithm used. In this case **RS256** algorithm is used, which is the most secure practice.

Setting the JWT Validator

The method `setJwtValidator()` allows to assign a number of validators for the received token. Default validators typically include checks for the token expiration, the token not being used before its validity period starts and the issuer.
A new class called `AudienceValidator` was also created. This validator ensures that the JWT is intended for the current resource server by checking the audience claim created in section 6.2 of the token against the expected value. In the case of customers application, the value of the audience field should be `customer-public-client`.

Jwt Authentication Converter

The main purpose of the `JwtAuthenticationConverter` bean is to configure Spring Security to work with JWT tokens for authentication and authority mapping in applications that use Keycloak as an identity provider. By extracting roles from the `realm_access` statement, it effectively maps Keycloak roles to Spring Security authorities, allowing fine-grained access control based on the user roles defined in Keycloak. Thus the *ADMIN* role

---

[1]https://stackoverflow.com/questions/36809528/spring-boot-cors-filter-cors-preflight-channel-did-not-succeed

created in section 6.2 will be converted to a `GrantedAuthority` class, which will be used to protect the application.

RestController Class

In general the application has two REST controllers, one for the administrator and one for the normal user.

Service Layer Interface

Both controllers use the `CustomerService` interface, which plays the role of a contract for the service layer of the application. This abstraction allows for flexibility in how the services are implemented and makes it easier to modify or replace the service logic without affecting the controller that depends on it. Interface-based design supports the use of different service implementations if needed, such as switching from a database to a web service for retrieving customer data without changing the controller logic.

Service Implementation

The `CustomerService` is implemented by the `CustomerServiceImpl` class providing the concrete functionalities outlined by the interface. The `CustomerServiceImpl` is tagged using the `@Service` annotation making it a Spring-managed service bean. By using the `@Service` annotation, Spring will automatically detect this class during component scanning and will manage its lifecycle, including creating and injecting it wherever the `CustomerService` interface is required.

Role-based Access

The entire admin controller is annotated with `@PreAuthorize("hasAuthority('ADMIN')")`. Thus, only a user with an access token that specifies the administrator role can access any method of this controller.

The regular user controller also uses the `@PreAuthorize` annotation, but to verify that the user is only requesting their own data. The implementation of the user data update method is shown in the following code snippet:

```java
@PreAuthorize("(#jwt.getClaim('sub') == #id.toString())")
@RequestMapping(value= "{id}/update/", method = RequestMethod.PUT)
public ResponseEntity<CustomerDTO> updateCustomer(
        @PathVariable UUID id,
        @AuthenticationPrincipal Jwt jwt,
        CustomerDTO customer) {

    boolean isUpdated = customerService.updateCustomer(id, customer);
    if (isUpdated) {
        return new ResponseEntity<>(HttpStatus.OK);
    }
    return new ResponseEntity<>(HttpStatus.NOT_FOUND);
}
```

The rule specified in the `@PreAuthorize` anotation checks that the requested user id is equal to the user id in the user's access token. To get the user `id` contained in the claim `sub` the method `getClaim()` is used.

Keycloak Admin Client

For user management, Keycloak can be integrated into the application as a Spring-managed bean. The following code snippets describe this process:

```
@Bean
public Keycloak getKeycloak() {
    return KeycloakBuilder.builder()
                .serverUrl(serverUrl)
                .realm(realm)
                .clientId(clientId)
                .grantType(OAuth2Constants.PASSWORD)
                .username(name)
                .password(password)
                .build();
}
```

Using Keycloak Admin Client the developer can retrieve and modify user information. The following code snippet shows the implementation of the method to get all users in JSON format:

```
@Override
public List<CustomerDTO> findAll() {
    List<UserRepresentation> reps = keycloak.realm(realm).users().list();
    List<CustomerDTO> customers = CustomerMapper.mapRepToCustomers(reps);
    return customers;
}
```

The Admin Client is used to interface with Keycloak programmatically for administrative purposes such as creating, modifying, and deleting user accounts. The admin client does not interfere with or replace the standard authentication and authorization processes that are handled via OpenID Connect. Instead, it complements these processes by providing a means to manage user data and roles, which are crucial for maintaining the security and integrity of user information.

6.4 Notes Application Server Setup

In the Customers Application Server Setup section of the thesis it was described in detail how the resource server is configured in terms of security, work with tokens and the architecture of the server itself. The Notes Application Server is configured in a similar way, except for some details that will be described below. Since the main purpose of the work is the OpenID Connect protocol, the work with databases is rather an auxiliary component. Therefore, this part of the application will be described briefly.

Security Configuration

The security configuration of this application completely repeats the configuration of the previous one with the exception of the audience claim check. In this application the `AudienceValidator` class checks for the presence of the `notes-public-client` value in the claim field.

Database Setup

The notes application server does not use the Keycloak Admin Client because it is not designed to work with users. Since its main purpose is to create notes, a database needs to be configured. Java H2 database was chosen for this purpose.

Repository Layer

NoteRepository interface was created to work with the database. This interface is an extension of Spring Data JPA's JpaRepository, which simplifies the data access layer by automating CRUD operations (Create, Read, Update, Delete).

To work with the methods of the interface, the `NoteServiceImpl` class uses constructor injection to receive an instance of `NoteRepository`. By separating concerns into different layers (repository and service), the application follows a clean architecture approach, which decouples the data access logic from business logic. This separation makes the system easier to maintain and evolve.

Docker Image

Both servers have the ability to package the application as Docker image, which is then required to run in the cloud. This was implemented by using the Maven utility and configuring the `pom.xml` file.

To package the application, all that is required is to enter the command:

```
mvn spring-boot:build-image
```

Image will be created and added to the local repository with the tag `latest`.

6.5 Customers Application Client

This part of the paper will describe a web application for customer management. The application is written in Type Script language using Angular framework. The client configuration for working with OAuth 2.0 and OpenID Connect protocols, role-based access, working with ID and access tokens and communication with the server will be described in detail.

OIDC Setup

This part of the paper will describe a web application for customer management. The application is written in TypeScript language using Angular framework. The client configuration for working with OAuth 2.0 and OpenID Connect protocols, role-based access, working with ID and access tokens and communication with the server will be described in

detail. This client uses the library `angular-oauth2-oidc` to work with the protocol and tokens.

AuthConfig Setup

To begin with, the configuration should be created using the `AuthConfig` class. This class is imported from `angular-oauth2-oidc` library which simplifies client configuration. The code below reflects the configuration of this class:

```
export const authCodeFlowConfig: AuthConfig = {
  issuer: 'http://keycloak:8080/realms/demo',
  useIdTokenHintForSilentRefresh: true,
  redirectUri: window.location.origin,
  clientId: 'customers-public-client',
  responseType: 'code',
  scope: 'openid profile email',
  showDebugInformation: true,
}
```

Key points of this configuration:

- `issuer` - contains the URL of the Keycloak server that the client can access and request configuration using the OpenID Connect discovery document published at a URL derived from the issuer.

- `useIdTokenHintForSilentRefresh` - allows the client to extend the session on its own without having to interact with the user.

- `clientId` - The client ID registered in Keycloak.

- `responseType` - Specifies the OIDC response type to code for the authorization code flow. In case of a public client, the **angular-oauth2-oidc** library will use authorization code flow with PKCE automatically.

- `scope` - Requests OpenID Connect openid, profile, and email scopes.

Using this configuration the application will communicate with the identity provider server using authorization code flow with PKCE.

OAuthService Setup

The authentication service needs to be initialized to handle the tokens directly, as well as manage the authentication process. The method for initialization is illustrated in the following code snippet:

```
function initOAuth(oauthService: OAuthService): Promise<void> {
  return new Promise((resolve) => {
    oauthService.configure(authCodeFlowConfig);
    oauthService.setupAutomaticSilentRefresh()
    oauthService.tokenValidationHandler = new JwksValidationHandler();
    oauthService.loadDiscoveryDocumentAndTryLogin()
```

```
      .then(() => resolve());
  });
}
```

Thus the service is initialized and also receives the configuration created earlier. The `setupAutomaticSilentRefresh()` method sets the automatic refresh of the access token, if the user session is active. `JwksValidationHandler()` class allows validating tokens from identity provider. The final step is the `loadDiscoveryDocumentAndTryLogin()` method, which loads the discovery document from the identity provider. The method will handle the redirections to Keycloak for login, handle the authentication response, and perform the code exchange for tokens using PKCE.

To initialize the `initOAuth()` method, it is required to specify it in the list of providers in the app.module.ts file.

Requests with Access Token

In order for the client to use access tokens to access the server, the server URLs should be added to the `allowedUrls` field in the root configuration. This code snippet further demonstrates this setting:

```
imports: [
  ...
  OAuthModule.forRoot({
    resourceServer: {
      allowedUrls: ['http://customers-server:8082/'],
      sendAccessToken: true,
    },
  }),
  ...
]
```

Any HTTP requests sent to URLs that start with any of the strings in this array will have OAuth2 access tokens automatically attached to them. In your case, all requests to `http://customers-server:8082/` will include the access token. This is crucial for APIs that require authentication, ensuring that only requests to specified URLs carry the access token, thereby avoiding unnecessary or unsafe token exposure. The token is typically added in the HTTP Authorization header as a Bearer token. This setting simplifies the process of making authenticated API calls by handling the token injection process automatically, reducing boilerplate code and potential for errors.

Components Generation

In order to generate a component in Angular, the command below was used: ng generate
component example-component

The following components were generated during the creation of the application:

- `HomeComponent` - the home page of the application.

- `ProfileComponent` - user's profile page, where data from his ID token and access
  token will be reflected.

- `HeaderComponent` - component for header management in the application. It will have
  a login button in case the user has not been authenticated yet or a logout button in
  case the authentication was successful.

- `FooterComponent` - footer part of the page.

- `CustomersViewComponent` - a component that can be accessed only by users who are
  authenticated but do not have admin privileges.

- `AdminViewComponent` - component with page for user administration. Only users
  with *ADMIN* role will be able to access it.

Components break the application into modules that are easy to modify and separate
different logic.

Decode ID Token

Using the `getIdToken()` method from the `angular-oauth2-oidc` library, the developer can
access the token in the application code itself. Using the method `getIdentityClaims()` it
is possible to access the token and use them in any client component in which an instance
of `OAuthService` class is imported.

Claims are of type `Record<string, any>`, which means we can access any claim in a
similar fashion:

```
this.givenName = claims['given_name'];
```

To decode the token and get it in JSON format this application also uses also the
`JwtHelperService` class from the `angular-jwt` library.

This is done as follows:

```
const helper = new JwtHelperService();
this.idTokenDecoded = helper.decodeToken(this.oauthService.getIdToken())
```

Routing Configuration

In order for the user to be able to navigate through the various components created in the previous section, a routing configuration must be created.

The following code snippet demonstrates how this is done in the `app-routing.module.ts` file:

```typescript
export const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent }
  { path: 'profile', component: ProfileComponent,
    canActivate: [AuthGuard].
  },
  { path: 'customers-view', component: CustomersViewComponent,
    canActivate: [AuthGuard] }
  },
  { path: 'admin-view', component: AdminViewComponent,
    canActivate: [AuthGuard],
    data: { role : 'ADMIN'}
  },
  { path: '**', redirectTo: 'home' }
];
```

Routes

Paths like 'home', 'profile', 'customer-view', and 'admin-view' are set up with corresponding components. The `canActivate` property uses `AuthGuard` to protect the routes, ensuring that only authenticated users with the appropriate roles can access certain parts of the application like profile, customers view, and admins view. We should pay attention to setting the path to AdminViewComponent, more precisely to the `data` parameter. Using this parameter we can get this field in the `AuthGuard` class and find out what role is required for this page.

AuthGuard

This section specifies the implementation of the `AuthGuard` class that implements the `CanActivate` interface from **Angular's Router** package, providing a way to decide if a route can be activated based on certain conditions.

```
export class AuthGuard implements CanActivate {
 ...
   canActivate(route: ActivatedRouteSnapshot,
               state: RouterStateSnapshot): boolean {

     if (this.oauthService.hasValidIdToken() &&
         this.oauthService.hasValidAccessToken()) {

       const claims : any = this.oauthService.getIdentityClaims();
       const roles : any = claims.role ? claims.role : '';
       if (route.data?.['role']) {
         if (roles.indexOf('ADMIN') === -1) {
               this.router.navigate(['/home']);
               return false;
         }
       }
       return true;
     }
     this.router.navigate(['/home']);
     return false;
   }
...
}
```

It then retrieves identity claims from the token using the `hasValidIdToken()` method, which typically include user information and roles. The code extracts roles from the claims. If no roles are found, it defaults to an empty string. It checks if the user has the admin role by searching the roles string. If a route has a specific role requirement (e.g., *ADMIN* for certain routes), and the user does not have this role, they are redirected to the */home* route, and access to the target route is denied. If the user is not authenticated (no valid ID token and access token), or if they lack the required role, they are redirected to the */home* page.

Service Class

In order to implement methods that interact with the server-side API, a service class called `CustomerService` was created. The class uses **Angular's HttpClient** to send HTTP requests and handle asynchronous responses via observables.
It uses `environment.apiBaseUrl` to determine the base URL of the API, which ensures that the application can adapt to different environments (development, production, etc.) without code changes. The implementation of the `getCustomers()` method is specified in the following code snippet:

```
public getCustomers(): Observable<Customer[]> {
  return this.http.get<Customer[]>(`${this.apiServerUrl}/customers`);
}
```

The method sends the request to the server, and receives the response in the form of users in JSON format. Users are converted into `Customer` object and can be used in the application.

Implementation of `deleteCustomer()` method code snippet:

```
public deleteCustomer(id : string): Observable<void> {
  return this.http.delete<void>(`${this.apiUrl}/admin/customers/${id}`);
}
```

The access token configured in the 6.5 section is also automatically added to the requests made to the server.

View Components

Since the service described in the last section is a layer for the implementation of the communication logic, the interaction with the service at the component level was also implemented. To interact with a service in any of the components, a constructor injection is used. Then the component can implement its methods based on the service. For example, this is how a method from the `CustomersViewComponent` class looks like to get the list of users from the service:

```
public getCustomers(): void {
  this.customerService.getCustomers().subscribe(
    (response: Customer[]) => {
      this.customers = response;
    },
    (error: HttpErrorResponse) => {
      alert(error.message)
    }
  )
}
```

This way the method can be assigned to a button in the component's HTML templating. To display user objects in HTML, a structural directive `NgFor` is used that renders a template for each item in a collection. The directive is placed on an element, which becomes the parent of the cloned templates.

6.6   Notes Application Client

The Notes client application is configured identically to the user management application. The client works with entity notes and provides information about them to the user. Although they manage different data entities, the pattern for fetching, displaying, updating, and deleting data via API calls is consistent across both clients. While the core application configuration might be similar (e.g., authentication, routing), each client has unique environment variables and API endpoints.

This approach allows the system to scale easily, either by adding new features to existing clients or by using the same architectural blueprint to build new clients for different data entities in the future. Any updates to shared components or services can be propagated easily across both clients, enhancing maintainability. This design principle is inherent in the real world of enterprise applications, as often applications follow the same design pattern but perform different functions.

Docker Image

The client application for customer and note management can be packaged as a Docker image. This is essential in the case of deploying applications in a cloud environment. For this purpose a Dockerfile has been created in which the Docker image configuration is located. In order to run an Angular application in a container it is required to use an image framework that contains node.js as well as an nginx server. The Dockerfile configuration is shown below:

```
FROM node:18-alpine as build
WORKDIR /app
COPY package*.json ./
RUN npm install
RUN npm install -g @angular/cli
COPY . .
RUN npm run build
FROM nginx:latest
RUN rm -rf /usr/share/nginx/html/*
COPY --from=build app/dist/customers-client/* /usr/share/nginx/html
COPY nginx/nginx.conf /etc/nginx/conf.d/default.conf
CMD ["nginx", "-g", "daemon off;"]
EXPOSE 80
```

At the beginning, the `package.json` file is copied, which contains the Angular project's dependencies. After that, all dependencies are installed and the project build is started. Then all the contents of the **dist** folder (it contains the build result) are copied to the nginx directory */usr/share/nginx/html*. There is also a configuration file for the nginx server, which contains a very basic configuration. Then the nginx server is started.

Chapter 7

Testing

This project will not be used for the real world and serves only as a prototype in one possible way to implement OAuth 2.0 and OpenID Connect in a cloud environment. Since this project serves as a concept for implementing protocols and not microservices in general, the testing part does not contain unit or integration tests. All tests were performed on applications running on Minukube. This section investigates whether the requirements of the project have been achieved.

7.1 User Authentication and Login

One of the main functional requirements was user authentication using Authorization Code Grant with Proof Key for Code Exchange flow using the OpenID Connect protocol. Once a user gets to the root page of any of the clients, they can start on the login button. He will then be redirected to the Keycloak server URL where he can authenticate or create a new account. A new account can also be created using Google or Github. After the user has successfully entered his data, he will be presented with a window with the scopes that the application requires from Keycloak. If the user agrees to the requirements, they will be redirected back to the client page where they can access the secure pages.
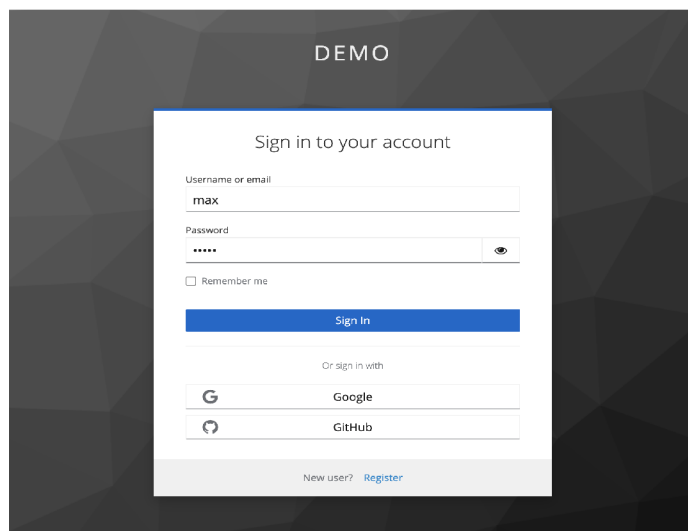

Figure 7.1: Screenshot of Keycloak login page.

7.2 User Profile

One of the pages of both client applications is the user profile. The user profile contains the user data taken from the access token and the token ID. Also at the bottom of the profile are the tokens themselves in plain and decoded states.

Figure 7.2: Screenshot of user profile page.

7.3 Single Sign-On

After the user authenticates to the client application, they can go to the URL of the notes application and click the login button. Since Keycloak already has this user's session in cookies, the user will be automatically authenticated without going to the Keycloak page. The session bound to the user will be transferred using an ID Token and also stored in cookies. The authentication and authorization procedure will be separated: the ID Token will pass the session information to the client application, and the access token will give the right to request the server. In case the user authenticates for the first time, he will be redirected to an agreement to provide scopes to this client. Screenshot of a part of a user's profile page with his ID Token can be found in appendix H.

If a user logs out of one of the applications, the session is deleted and all tokens bound to it become invalid after 5 minutes. To check this case, you should authenticate in both clients and then press the logout button in one of them. After that, going to the admin panel of Keycloak in the **Sessions** section you can make sure that the session bound to the user has been deleted.

7.4 Authorization

Authorization is based on the user's access tokens, which store information about the user's role. If a user has the *ADMIN* role in the advertisement mashup, he/she will be able to access the admin page of the application. After the user goes to the admin panel of the

application, he can delete any user in the case of customers application. A user without the admin role cannot delete users. Authorization takes place at the client application level - that is, a user without the admin role will not be able to access the admin panel, as well as the server application - a request with an access token without the *ADMIN* role will result in a negative response from the server.

Figure 7.3: Screenshot of Customers Application admin page.

7.5 Separation of Access Tokens

In order to prevent the token from being passed to third parties by any other application connected to the same identity provider, it was decided to separate access tokens for each application. Thus the token received in customers application cannot be used to access the notes application server. Tokens have different audience so they can be used only for their own services. In order to check this, it is enough to get the token in customers application and then apply it in a request for the notes service. The result will be HTTP 401 Unauthorized response as on the screenshot in appendix I.1.

7.6 Cloud Deployment

It is also possible to build a Docker image based on each of the created applications. Using these images, as well as the publicly available Keycloak server image, it is possible to run all applications using the docker compose utility or create a cluster using Minikube. All applications were run in a Minikube cluster at the same time and all functionality listed in the sections above was tested.

Chapter 8

Lessons Learned

While working on this thesis implementing a cloud-based single sign-on (SSO) system using OAuth 2.0 and OpenID Connect with Keycloak, a number of valuable lessons were learned that not only deepened the understanding of identity management systems, but also provided a foundation for best practices in the field.

8.1 Technical challenges and solutions

One of the main challenges of this work is my lack of experience in building system architecture. One of the first and key points in developing this kind of prototypes is to plan the system architecture. Since my experience in this area is limited, I often wrote applications and then completely changed the architecture, so I had to partially rewrite existing applications as well as create new ones. Thanks to this problem, I realized the importance of careful planning of the environment before full integration. The solution to this problem was to study a large number of sources related to this topic, as well as the advice of a more experienced engineers.

The Angular framework also turned out to be quite a difficult technical problem. Initially it was chosen as a framework well suited for writing a client to a Java service. In the course of writing the work it turned out that this framework is quite difficult to master and requires more careful study. Quite simple things, such as creating forms require a lot of effort and often incorrect configuration of just one component can affect the rest. The `angular-oauth2-oidc` library, on the contrary, pleased with its simple customization, as well as a large number of all sorts of sources on customization in the public domain.

Also, one of the problems inherent in all systems with a large number of services that need to be written in a cloud environment is the CI/CD pipline and a single configuration manager. This problem was not addressed within the scope of this work. Its solution is the use of DevOps platforms such as **GitLab** or **Ansible**, which facilitate the deployment, configuration and monitoring of a large number of microservices.

8.2 Best Practices Identified

Token separation

A best practice was the use of minimum access principles. By limiting access to tokens based on the exact requirements of different parts of the application, security was greatly improved and the potential impact of compromised tokens was reduced.

Number of scopes

Developers should also minimize the number of scopes issued to the client by default from identity provider, since most of this information is not required by the client.

Using PKCE

Using PKCE allows to avoid CSRF and interception attacks, which also increases application security.

Enforcing exact paths in identity provider configurations

After a client has been registered with identity providers the Valid redirect URIs and Web origins settings must be set. Setting these parameters in the production release of the application is mandatory and protects the user from potential redirection of the client to a malicious site.

8.3 Identity provider selection

The choice of Keycloak as the identity provider was crucial due to its broad support for OAuth 2.0 and OpenID Connect, as well as its adaptability to cloud environments. This choice confirmed the project's initial hypothesis that Keycloak could simplify the implementation of SSO between services, although it required adjustments to address the specific challenges within the work.

8.4 Personal and professional growth

This project has greatly enhanced my technical skills, especially in the areas of security and cloud native architectures. It also improved my problem solving skills as I dealt with complex integration and architectural issues and learned to adapt theoretical knowledge to practical problems.

As the world of technology is rapidly evolving, new security protocols and standards are emerging, so application developers must not only become more knowledgeable and experienced in application development, but also in the security and proper configuration of products such as Keycloak.

Chapter 9

Conclusion

The objective of this thesis was to implement and evaluate a cloud-native Single Sign-On (SSO) solution using OAuth 2.0 and OpenID Connect protocols, with Keycloak serving as the central Identity and Access Management system. Functional tasks of the project included implementation of these protocols into modern client and service applications capable of running in the cloud, setting up Keycloak server, user authentication and authorisation capability and establishing a secure and scalable SSO architecture.

The implementation of the demonstration system includes two client and two service applications as well as a Keycloak server. This architecture allowed to test user authentication and authorisation on the client side as well as authorisation on the service side. The use of two different clients allowed the single sign-on scheme to be configured and tested, as well as the separation of access to the service applications. Also during the solution the Keycloak server was studied and configured, which allowed to centralise the identity and access management, which is necessary in the implementation of the single sign-on scheme.

The result of the implementation was the ability for a user to authenticate once to use two client applications without revealing his credentials to either of them. After successful user authentication on the Keycloak server side, clients receive session information in the form of an ID token, as well as an access token for communicating with services. Both client applications receive different tokens from the Keycloak server to access their microservices, which minimises the chances of unauthorised access in case the tokens of one of the clients are compromised. The entire system was deployed on a local Kubernetes cluster using Minikube, and its functionality was tested both manually and using tools like Postman.

The process of writing all applications as well as the configuration of the Keycloak server was carefully documented. Configuration files were created to deploy all applications in the cluster.

The requirements for this thesis were fully met. The project deepened my understanding of the security protocols underlying modern authentication systems. The experience gained from this project is valuable not only for understanding the complex details of identity and access management, but also for applying these concepts to real-world applications. Further extensions of the project may include more complex examples of applying the technologies used, such as cross domain SSO in a cloud environment or connecting LDAP to Keycloak as a centralised user data storage and management system.

Bibliography

[1] BROADCOM. *Spring Security* online. Available at:
https://spring.io/projects/spring-security.

[2] CANTOR, S.; HIRSCH, F.; KEMP, J.; MALER, E. and PHILPOTT, R. *Security Assertion Markup Language (SAML) V2.0 Technical Overview* online. 15. march 2005. Available at:
https://docs.oasis-open.org/security/saml/v2.0/saml-bindings-2.0-os.pdf.

[3] GAUTIER, E. *Understanding OAuth 2 Access Token Claims* online. 26. february 2023. Available at:
https://www.cerberauth.com/understanding-oauth2-access-token-claims.

[4] GUEVARA, H. *How SAML Authentication Works* online. 2021. Available at:
https://auth0.com/blog/how-saml-authentication-works/.

[5] HAMMER LAHAV, E. *The OAuth 1.0 Protocol* RFC 5849. RFC Editor, april 2010. Available at: https://doi.org/10.17487/RFC5849.

[6] HARDT, D. *The OAuth 2.0 Authorization Framework* RFC 6749. RFC Editor, october 2012. Available at: https://doi.org/10.17487/RFC6749.

[7] INC., A. *How SAML Authentication Works* online. Auth0 Inc. Available at:
https://jwt.io/introduction.

[8] INC., A. *Authorization Code Flow with Proof Key for Code Exchange (PKCE)* online. Auth0 Inc., 2024. Available at: https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-pkce.

[9] JONES, M. B. *JSON Web Algorithms (JWA)* RFC 7518. RFC Editor, may 2015. Available at: https://doi.org/10.17487/RFC7518.

[10] JONES, M. B.; BRADLEY, J. and SAKIMURA, N. *JSON Web Token (JWT)* RFC 7519. RFC Editor, may 2015. Available at: https://doi.org/10.17487/RFC7519.

[11] JONES, M. B. and HARDT, D. *The OAuth 2.0 Authorization Framework: Bearer Token Usage* RFC 6750. RFC Editor, october 2012. Available at:
https://doi.org/10.17487/RFC6750.

[12] JONES, M. B. and HILDEBRAND, J. *JSON Web Encryption (JWE)* RFC 7516. RFC Editor, may 2015. Available at: https://doi.org/10.17487/RFC7516.

[13] KEYCLOAK. *Keycloak Admin REST API* online. Keycloak Authors, 2024. Available at: https://www.keycloak.org/docs-api/24.0.3/rest-api/.

[14] KEYCLOAK. *Securing Applications and Services Guide* online. Keycloak Authors, 2024. Available at:
https://www.keycloak.org/docs/latest/securing_apps/#endpoints.

[15] KEYCLOAK. *Server Administration Guide* online. Keycloak Authors, 2024. Available at: https://www.keycloak.org/docs/latest/server_admin/.

[16] RAGOUZIS, N.; HUGHES, J.; PHILPOTT, R.; MALER, E.; MADSEN, P. et al. *Security Assertion Markup Language (SAML) V2.0 Technical Overview* online. 25. march 2008. Available at: https://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html.

[17] SAKIMURA, N.; BRADLEY, J.; JONES, M. B.; MEDEIROS, B. de and MORTIMORE, C. *OpenID Connect Core 1.0* online. The OpenID Foundation, 2014. Available at: https://openid.net/specs/openid-connect-core-1_0.html.

[18] SEBASTIÁN E. PEYROTT, A. I. *The JWT Handbook* online. Auth0 Inc., 2016-2018. Available at: https://auth0.com/resources/ebooks/jwt-handbook.

[19] STEYER, M. *Angular-oauth2-oidc Library* online. Manfred Steyer. Available at: https://manfredsteyer.github.io/angular-oauth2-oidc/docs/.

[20] VAPEN, A. *Web Authentication Using Third-Parties in Untrusted Environments*. 1st ed. Linkopings Universitet, 2016. ISBN 9789176857533. Available at: https://ebookcentral.proquest.com/lib/vutbrno/detail.action?docID=30401342.

[21] YELURI, R. and CASTRO LEON, E. *Building the Infrastructure for Cloud Security*. 1st ed. Apress L. P., march 2014. ISBN 9781430261469. Available at: http://ebookcentral.proquest.com/lib/vutbrno/detail.action?docID=6422525.

Appendix A

Authorization Code Grant with PKCE
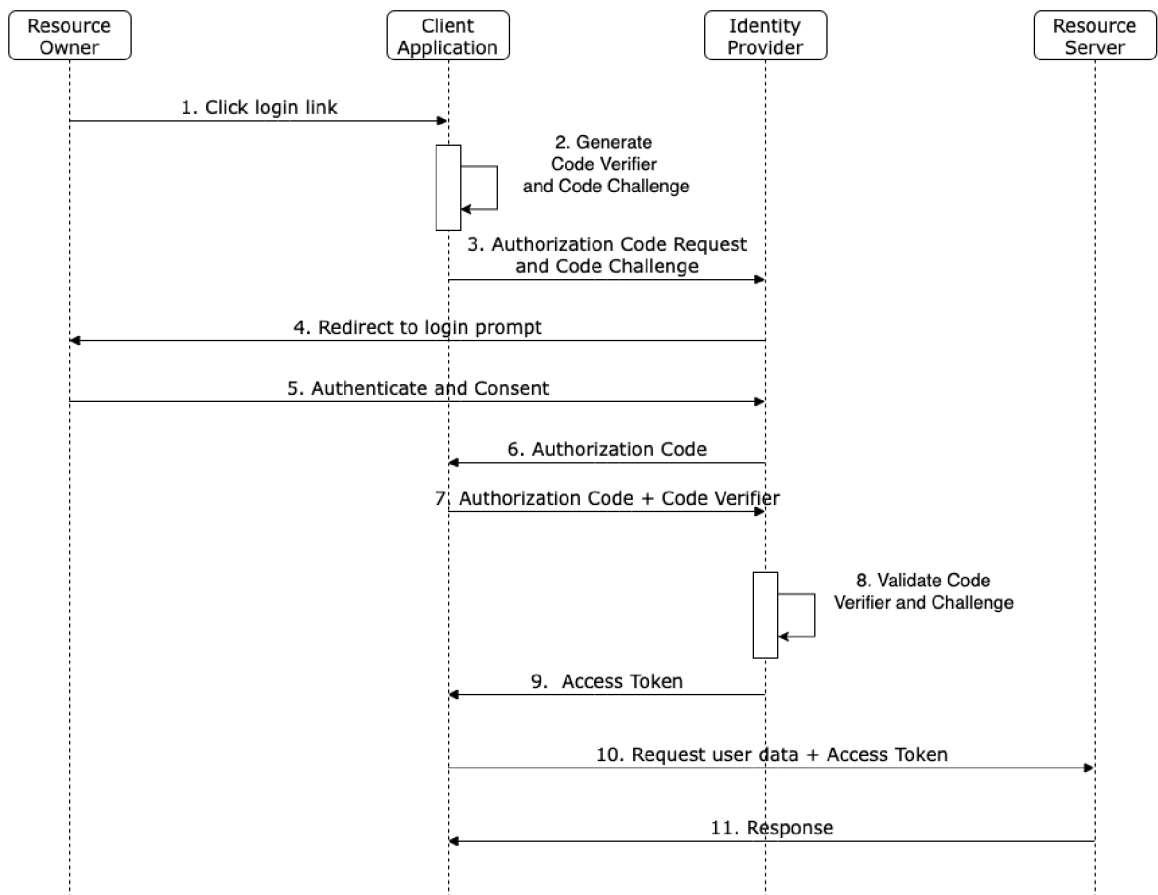
Figure A.1: Authorization Code Grant with PKCE diagram.

Appendix B

Keycloak realm creation
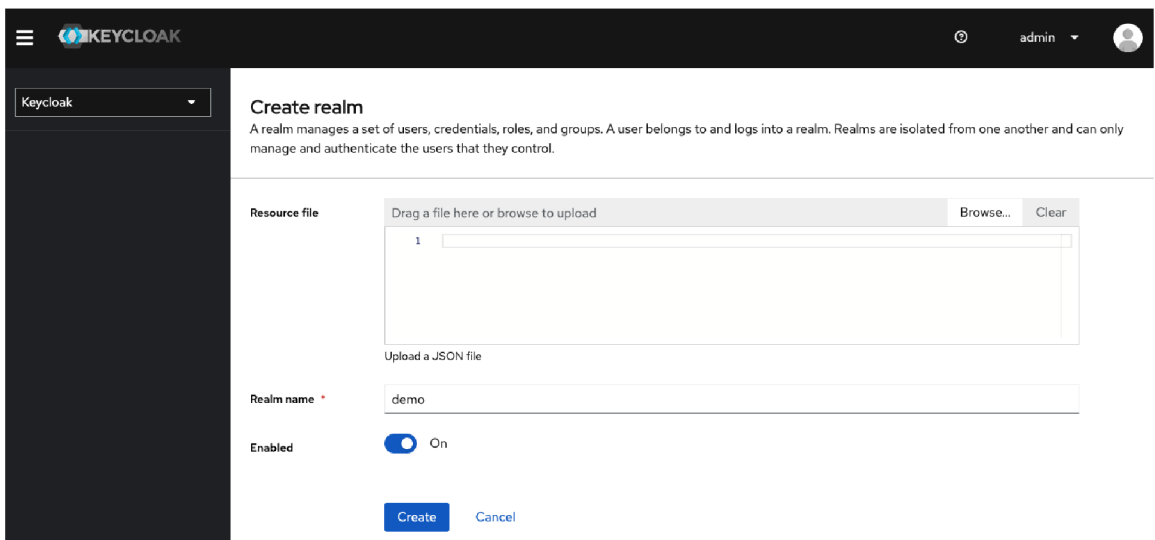
Figure B.1: Screenshot of Keycloak realm creation.

Appendix C

Keycloak client creation
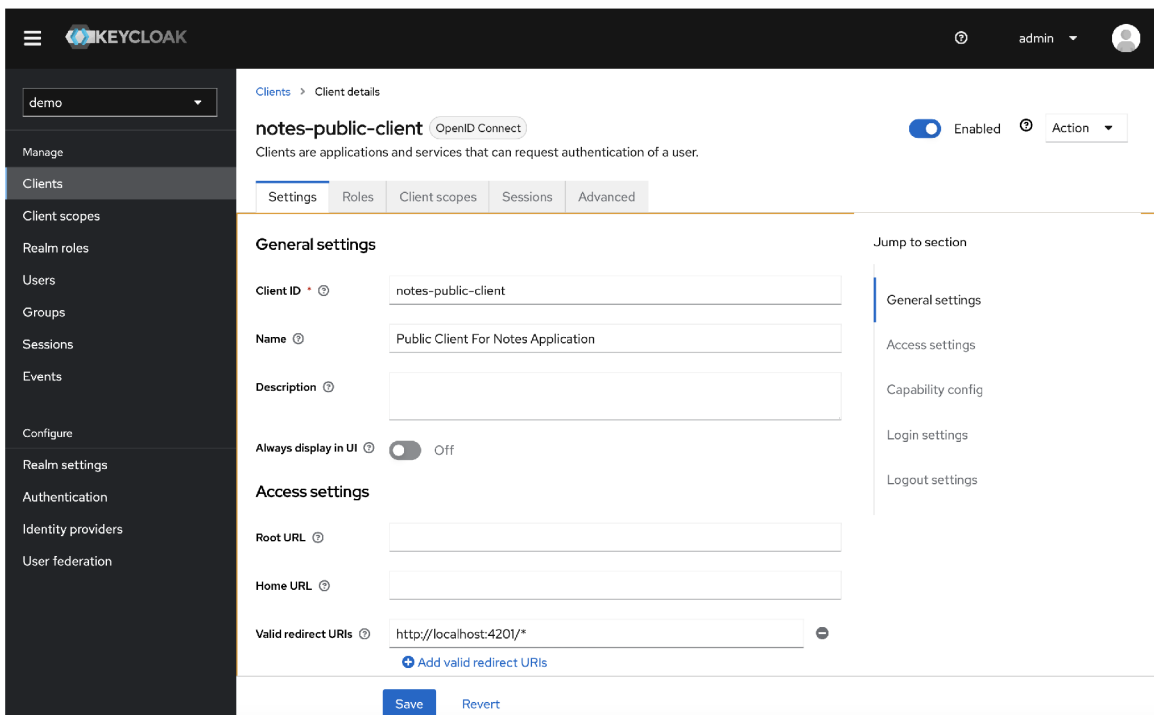
Figure C.1: Screenshot of Keycloak client creation.

Appendix D

Keycloak mapper creation
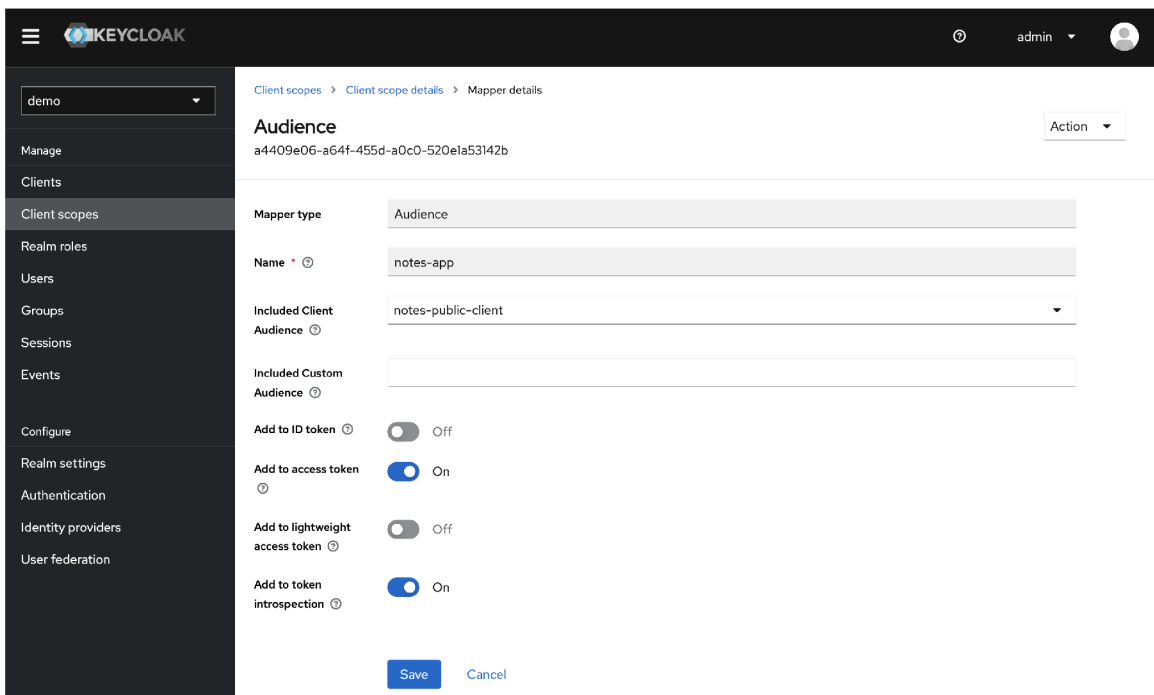
Figure D.1: Screenshot of Keycloak mapper creation.

Appendix E

Functional Architecture
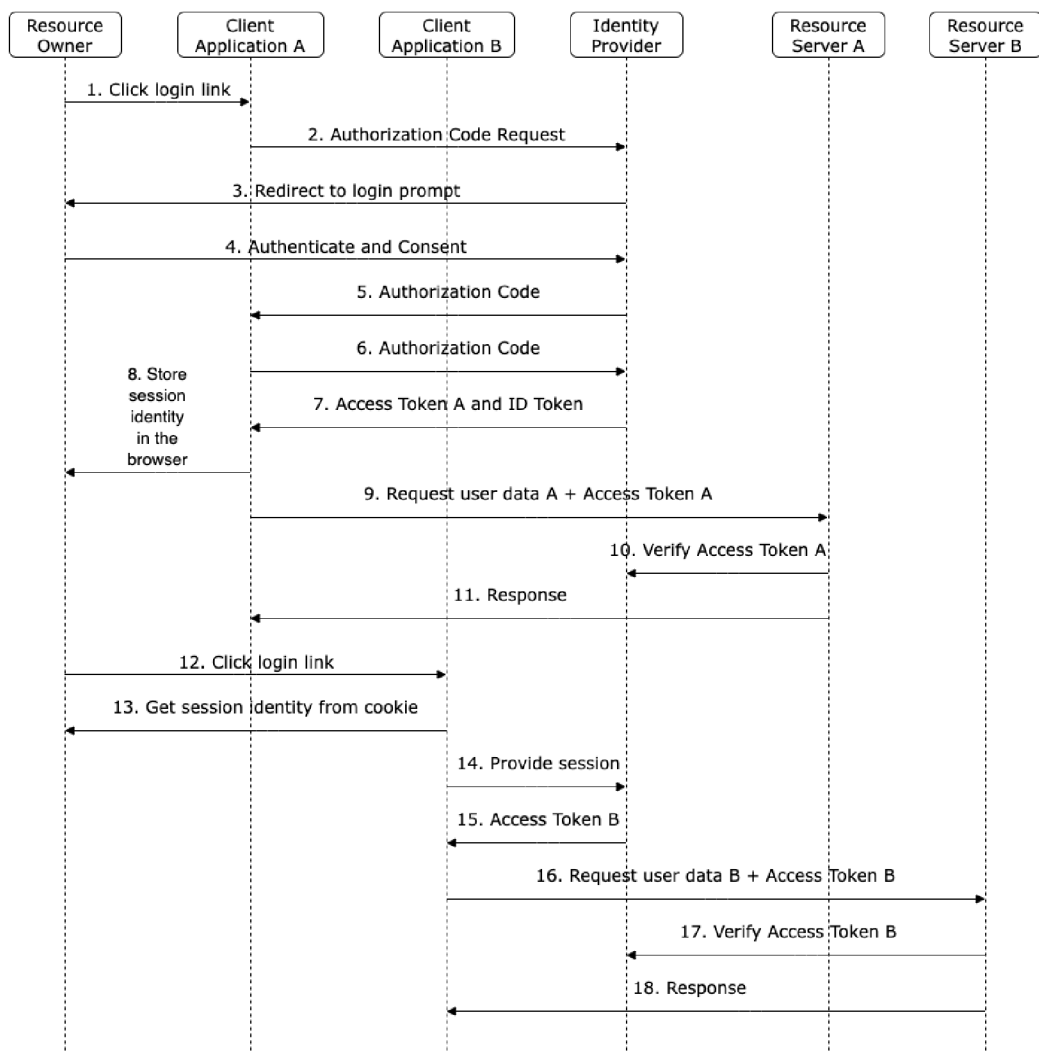
Figure E.1: Functional Architecture Diagram.

Appendix F

Notes Application Use Case Diagram

Figure F.1: Notes Application Use Case Diagram.

Appendix G

Customers Application Use Case Diagram

Figure G.1: Customers Application Use Case Diagram.

Appendix H

User's Profile Page with ID token

OIDC Token:

eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6I
CJmYmRNc2ZOTmxpU285My1lcEQ4Wm44RmFHcD
lPYUVXZC10dEQ2OWZpSUJVIn0.eyJleHAiOjE3MTU
yMjM3NDQsImlhdCI6MTcxNTIyMzQ0NCwiYXV0aF9
0aW1lIjoxNzE1MjIyNTMyLCJqdGkiOiI0NjhlZTc1NS02
YjhjLTQyOTktOGRlZi0yMjM2ZWQ5MGMxNTkiLCJpc
3MiOiJodHRwOi8vbG9jYWxob3N0OjgwODAvcmVh
bG1zL2RlbW8iLCJhdWQiOiJmcm9udGVuZC1wdWJs
aWMtY2xpZW50Iiwic3ViIjoiMGZjY2IyYjAtZWIwOS00
NDFlLThmYzYtMDY1MjM4NGViZGRiIiwidHlwIjoiSUQi
LCJhenAiOiJmcm9udGVuZC1wdWJsaWMtY2xpZW5
0Iiwibm9uY2UiOiJTMUoxTnpaV2VVSTBTekJzTlhWT
2FVVk5iazFxZVlmU2Q2U2xaaFExWnJjiWE
E0TlhwTU0wOXEiLCJzZXNzaW9uX3N0YXRlIjoiNzR
mM2VjN2ItYWQ2Zi00ZTg5LTkwNjQtOTc3Mjg1MGF
mMjE3IiwiYXRfaGFzaCI6ImFNanlEcUQ2UWR3bnhG
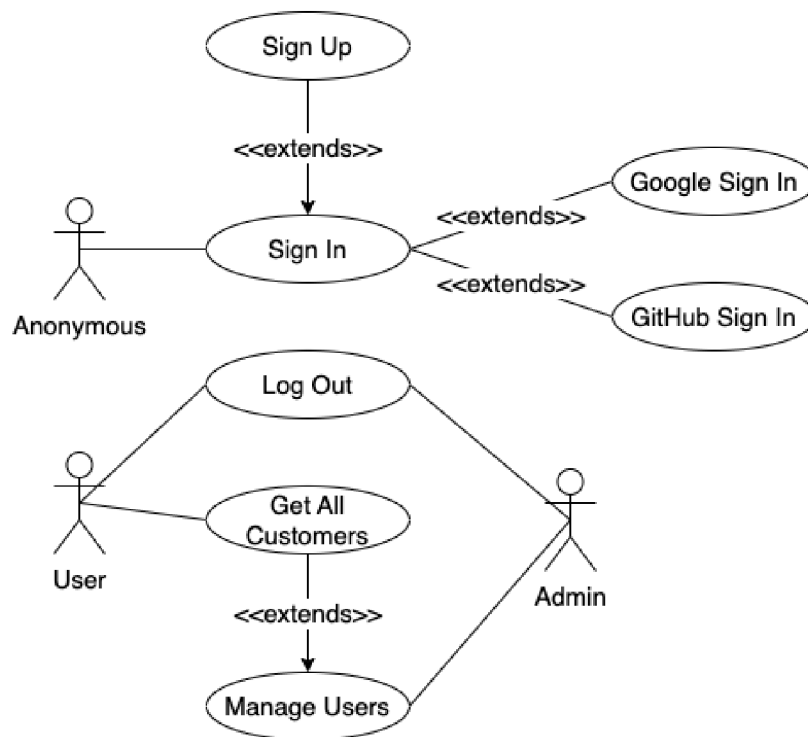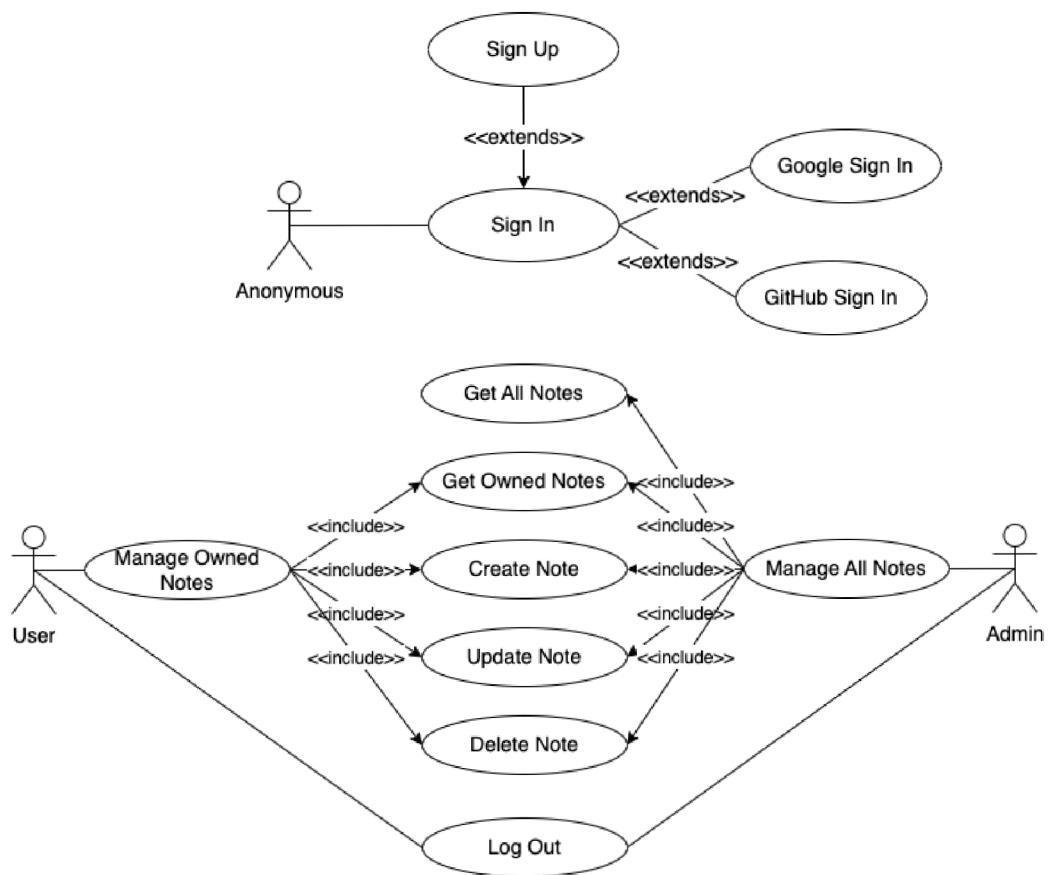bG5PRW45VkEiLCJhY3IiOiIxIiwic2lkIjoiNzRmM2VjN
2ItYWQ2Zi00ZTg5LTkwNjQtOTc3Mjg1MGFmMjE3Iiw
iZW1haWxfdmVyaWZpZWQiOnRydWUsInJvbGUiOls
ib2ZmbGluZV9hY2Nlc3MiLCJBRE1JTiIsInVtYV9hdX
Rob3JpemF0aW9uIiwiZGVmYXVsdC1yb2xlcy1kZW1
vII0sInJlYWxtX2FjY2VzcyI6eyJyb2xlcyI6WyJvZmZsa
W5lX2FjY2VzcyIsIkFETUlOIiwidW1hX2F1dGhvcml6Y
XRpb24iLCJkZWZhdWx0LXJvbGVzLWRlbW8iXX0sI
m5hbWUiOiJNYWtzeW0gS292YWwiLCJwcmVmZXJ
yZWRfdXNlcm5hbWUiOiJtYXgiLCJnaXZlbl9uYW1lIjo
iTWFrc3ltIiwiZmFtaWx5X25hbWUiOiJLb3ZhbClsImV
tYWlsIjoibWF4QGdtYWlsLmNvbSJ9.nmRWlSX2VAL
hbTQDDS_lowIhVgvhWqZ6B0xlvJma-
PZ04Hu1XXZYUcAu6QMa0EUvbqcDZOnKQR5rpJtA
Yl8o74p0ibsuA2xJ9lr5DlizuUIvxlw_AWie368kM0Bd
3WQr2cihfLYS0CQWwCRaJMwyeIphdjXRggzZqWvj
x- UBirDWLu0Ys8kCEofNPLEcq-

Decoded OIDC Token:

```
{
  "exp": 1715223744,
  "iat": 1715223444,
  "auth_time": 1715222532,
  "jti": "468ee755-6b8c-4299-8def-2236ed90c159
  "iss": "http://10.108.144.30:8080/realms/der
  "aud": "customers-public-client",
  "sub": "0fccb2b0-eb09-441e-8fc6-0652384ebddl
  "typ": "ID",
  "azp": "customers-public-client",
  "nonce": "S1J1NzZWeUI0SzBsNXVOaUVNbk1qZVlfS
  "session_state": "74f3ec7b-ad6f-4e89-9064-97
  "at_hash": "aMjyDqD6QdwnxFlnOEn9VA",
  "acr": "1",
  "sid": "74f3ec7b-ad6f-4e89-9064-9772850af21
  "email_verified": true,
  "role": [
    "offline_access",
    "ADMIN",
    "uma_authorization",
    "default-roles-demo"
  ],
  "realm_access": {
    "roles": [
      "offline_access",
      "ADMIN",
      "uma_authorization",
      "default-roles-demo"
    ]
  },
  "name": "Maksym Koval",
  "preferred_username": "max",
  "given_name": "Maksym",
  "family_name": "Koval",
  "email": "max@gmail.com"
}
```

Figure H.1: Screenshot of a part of a user's profile page with his ID token.
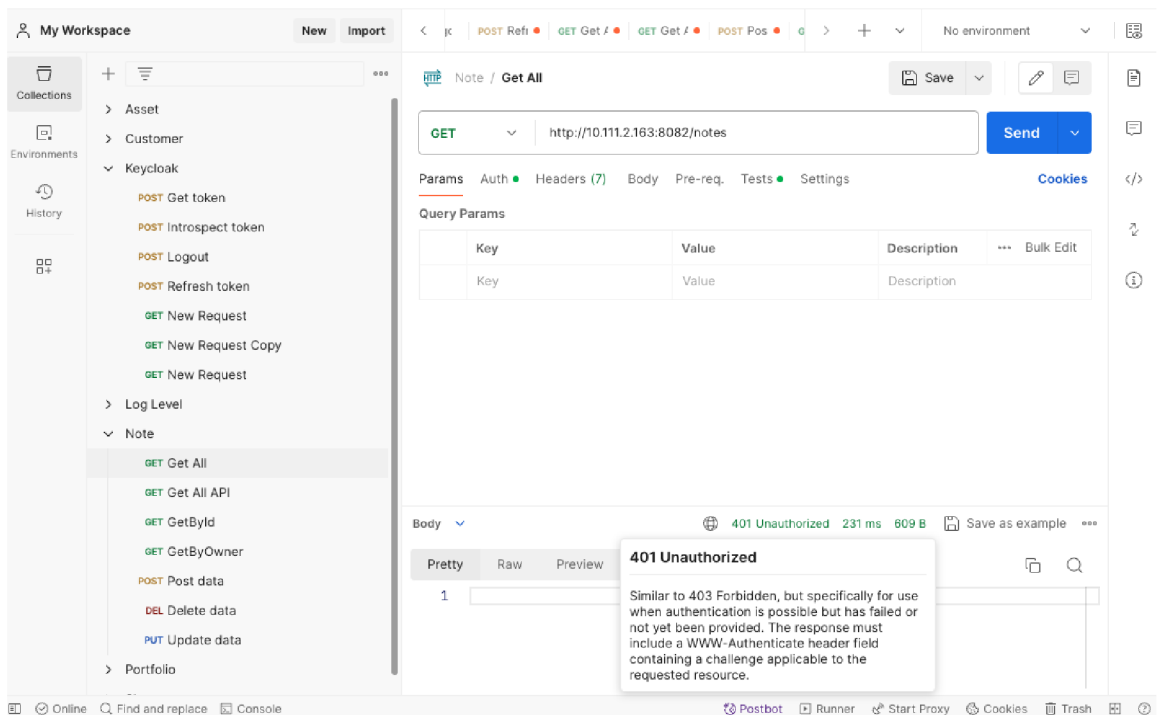
56

Appendix I

Postman Application Screenshot

Figure I.1: Screenshot from Postman application with 401 Unauthorized response.