Hochschule für Angewandte Wissenschaften Hamburg
*Hamburg University of Applied Sciences*

# Bachelor Thesis

**Abdur Rahim**

Conception of Authentication and Authorization of Users and Microservices in an Angular, Spring Boot Application Using Keycloak and OAuth2.0

*Fakultät Technik und Informatik*
*Department Informations- und Elektrotechnik*

*Faculty of Engineering and Computer Science*
*Department of Information and Electrical Engineering*

Abdur Rahim

# Conception of Authentication and Authorization of Users and Microservices in an Angular, Spring Boot Application Using Keycloak and OAuth2.0

Bachelor Thesis based on the examination and study regulations
for the Bachelor of Engineering degree programme
Information Engineering

at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg

Supervising examiner: Prof. Dr. Pawel Buczek
Second examiner: Prof. Dr. Robert Heß

Day of delivery: 05 July 2021

**Abdur Rahim**

**Thema der Arbeit**

Konzeption der Authentifizierung und Autorisierung von Benutzern und Microservices in einer Angular, Spring Boot Anwendung mit Keycloak und OAuth2.0

**Stichworte**

Authentifizierung, Autorisierung, Microservice, Angular, Spring Boot, Keycloak, OAuth2.0

**Kurzzusammenfassung**

Ziel dieser Bachelorarbeit ist es, das Konzept des Authentifizierungs- und Autorisierungsprozesses einer Microservice-basierten Prototyp-Anwendung zu entwickeln. Das Konzept des Projekts wird dann in einer praktischen Anwendung implementiert. Im Rahmen dieser Arbeit werden die Anforderungen des Projekts diskutiert und das Konzept darauf aufgebaut. Darauf aufbauend wird die Implementierung des Gesamtprojekts abgeschlossen und beschrieben. Die Arbeit erfordert ein gründliches Verständnis von Softwareentwicklung und -architektur.

**Abdur Rahim**

**Title of Thesis**

Conception of Authentication and Authorization of Users and Microservices in an Angular, Spring Boot Application Using Keycloak and OAuth2.0

**Keywords**

Authentication, Authorization, Microservices, Angular, Spring Boot, Keycloak, OAuth2.0

**Abstract**

This bachelor's thesis aims to discover the concept of a microservice-based prototype application's authentication and authorization process. The project's concept will then be implemented in a real-world application. Throughout this thesis, the project's needs will be discussed, and the concept will be built upon them. The implementation of the full project will be completed and described based on this. The work necessitates a thorough understanding of software development and architecture.

# Table of Contents

# 1 Introduction

The microservice architecture pattern has grown in popularity as a software development architecture. It has many advantages, including autonomous scalability, flexibility in technology selection, decentralization of feature management, and the ability to deploy microservices individually. As a result, Hermes Germany's Integration Platform utilized microservice architecture in the creation of their applications. However, together with the numerous advantages of migrating monolithic systems to microservices design, enterprises must also handle new security concerns. It is required to keep in mind that the security of microservices necessitates the adoption of recent security trends by specialized security teams. Since microservices are distributed, the management of the microservice application becomes more difficult. Implementing authentication and authorization processes in microservice applications to manage security and access control is one of the most challenging tasks.

## 1.1 Objectives

This thesis aims to find a proper conception to protect the operation and configuration of the Integration Platform of Hermes Germany’s applications from external influences by means of appropriate authentication and authorization. Hence a prototype application using microservice architecture pattern behind an API Gateway will be developed where the authentication and authorization will be implemented using OAuth 2.0 and Keycloak. The user interface will be developed using Angular's latest version.

# 2 Theory

In this section the theory of the technology stacks that are needed for this work will be explained briefly. The purpose of this section is to have a better realization on Software Architecture, Microservices, Spring Boot, Angular, Authentication, Authorization, Keycloak, and OAuth2.0.

## 2.1 Software Architecture

While developing software, the first thing that comes to mind is the architecture of the software. Software architecture is the structure of a system. It is the blueprint of the software. To develop a software architecture, there are few things that have to be known, which are the components of the software, the relationship between the components, how the components will communicate with each other, details about the data structure, data flow, control flow, etc. This leads to the software architectural patterns. The software architectural pattern defines how large or how small the software components should be. (Len Bass, 2012)

### 2.1.1 Monolithic

A monolithic application is one massive container in which the whole application or the system is composed in one piece. The goal of the monolithic application design is to drive several allied tasks together. The monolithic applications are typically very compound applications that enclose multiple closely constrained functions. The components of the application are related to each other and dependent on each other. It is called tightly coupled architecture, where every component and its related components must work correctly to run the program. Monolithic was one of the most used architecture, and it is still one heavily used architecture. Because this architecture is not complex, and it is straightforward to deploy. However, over few years, there have been challenges with this kind of architecture. (Richardson, 2019)

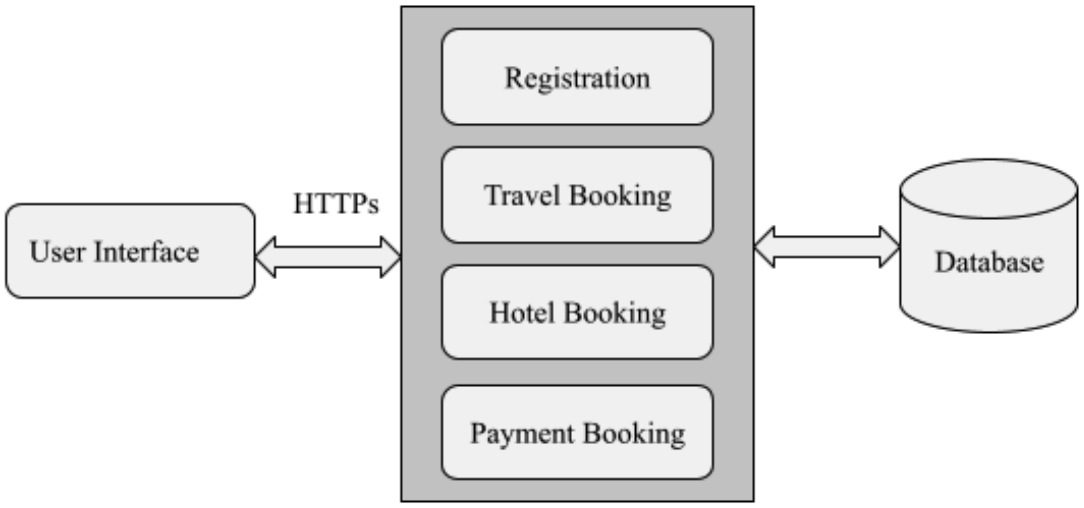


Figure 2.1: Monolithic architectural pattern example

Considering the above example of an online hotel management system, all these services would be deployed inside a big app server where all the components are heavily coupled. That causes a lot of rigidness. The first challenge in monolithic architecture is scalability. For instance, in the above example, if an enhancement is needed in the hotel booking database, the software developer will make changes in the hotel booking service while the travel booing service, payment service, registration service will not be affected by the changes. However, it is not possible in monolithic architecture. To accomplish the enhancement, the entire application must bring down. Because in monolithic architecture, everything resides in a single deployment artifact. The single failure point is another stress. If a small failure occurs in the application, the entire application goes down after fixing the collapse, an entirely new version of the system to be created and deployed. Over the period, the application grows bigger and more complex. As a result, the application developers find it hard to develop and maintain such an application. In conclusion, such systems are difficult to scale and unreliable. As a result, adhering to agile architecture and standards becomes a considerable challenge. (Richardson, 2019)

### 2.1.2 Microservices

Microservice architecture is a unique approach to software development that focuses on creating single-function modules with very well interfaces and functions. Microservices allow complex applications to be implemented using an architecture made up of several tiny, decoupled services and processes. Microservices communicate to each other through interfaces as well as provide individual functionalities

to the application. The implementation specifications are shielded from the outside through interacting via interfaces. On the principle of microservices, very complex software programs can be built and implemented in a modular manner. Individual microservices are self-contained and can be deployed on multiple machines in distributed systems or on cloud platforms. The services can be started, stopped, modified, and scaled independently of one another. Microservices and containers are used to create so-called software systems that entirely leverage the capabilities and benefits of cloud environments.

A microservice is designed by the fact that it performs a single or limited of tasks. The task must remain manageable, and the services could be replaced without colossal effort. Small development teams manage the microservices, allowing them to make improvements and updates quickly and flexibly. In the below example architecture, the payment service is only responsible for performing the payments. Any modification needed in the payment service can be easily done without impacting the other services. Microservices can be polyglot as well. For instance, in the below example, the responsible software development team for the travel booking service may use Java and Spring Boot to develop the system. However, the software development team responsible for payment service may use different technology stacks like Python, Node, etc. They can also use other databases as well.

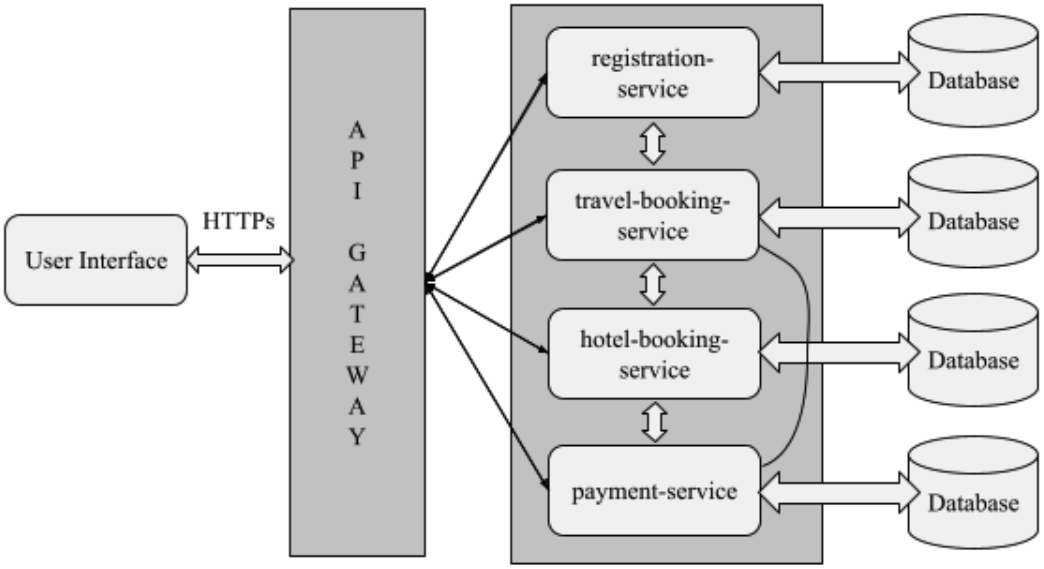


Figure 2.2: Microservice architectural pattern example

The microservices are hosted on separate machines, virtual servers, or operating systems. This distributed technique avoids microservices interfering with each other or potentially overloading host systems. There are some challenges in microservice architecture like managing microservices, monitoring, fault tolerance, cyclic dependencies, DevOps culture, etc. However, as businesses strive to become more

agile and shift toward DevOps and test automation, the practice has gained popularity in recent years. (Singh, 2018), (Walls J. , 2017)

### 2.1.3 Monolith vs. Microservices

From the above theoretical discussion, both the software architecture pattern, monolithic and microservices, have their advantages and disadvantages. In the following table, the main differences between monolithic and microservices architecture have been pointed out. (Kharenko, 2015)

| Categories | Microservices | Monoliths |
|---|---|---|
| **Release** | Because of the technological difficulties that microservices face, they will be slower at first. Later, though, it will be quicker. | As the program increases in size, it becomes faster at first, then slower later. |
| **Refactoring** | Since adjustments are implemented inside the microservice, it is simpler and safer. | It is difficult to do because changes can have an effect on many areas. |
| **Scaling** | When required, each service could be scaled separately. | The term "scaling" refers to the deployment of the entire monolithic application. |
| **Deployment** | It is possible to deploy it in tiny chunks, with just a single service to be deployed at a time. | The entire application must be deployed at all times. |
| **Comprehendibility** | Since the codebase is purely modular and services follow the single responsibility concept, it's simple to understand. | Because of the high level of difficulty, it is difficult to comprehend. |

| | | |
|---|---|---|
| **Implementation** | Every service is coded in the programming language that best meets the requirements. | Usually, only one programming language is used. |
| **Agility** | Modifications may be made to every service separately. | A latest version of the implemented system must be built and deployed to make improvements to the system. |
| **Developement Team** | A dedicated team is in charge of each microservice. | A single wide application is worked on by one or maybe more teams. |

### 2.1.4 Microservices Communication

Communication between microservices can occur at many levels.

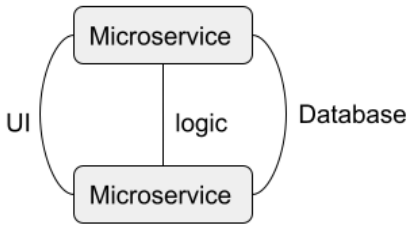


Figure 2.1.4: Microservice communication at different levels

Various techniques for microservice communication can be implemented at the user interface level. Microservices should, in general, have their own user interface. In the simplest scenario, the UI is built as a single-page application (SPA). Other microservices can then be accessed with a simple hyperlink. Another option is to use a single Single Page Application for all microservices, with the different microservices kept independent. Similarly, the user interface might be created using simple HTML.

Representational State Transfer (REST), Service Oriented Architecture (SOAP), and messaging can all be used to communicate at the logic level. Messages are transmitted between separate microservices using messaging.

The microservices are still distinct at the database level, but they can share a database and access database information. However, because modifications to the data sets can cause issues in other microservices, this kind of interaction is not encouraged. (Wolff, 2016)

## 2.2 Spring Boot

The Spring Framework is a Java platform that is free and open source. It is the most widely used Java application development platform. Millions of software developers use spring Framework to build high-performing, easily testable, and reusable code. Spring Boot can be thought of as the next chapter in the Spring Framework. Spring Boot provides the flexibility to build self-contained, production-ready Spring-based software that can be just run. Usually, the development of Spring software begins with the manual editing of many XML (Extensible Markup Language) files for configuration, but with Spring Boot, this is not mandatory. Manual configuration is no longer needed with spring boot. (Walls C. , 2016) (Walls C. , 2016) (Walls C. , 2016)

### 2.2.1 Spring Data

Spring Data is a powerful high-level data source platform aimed at unifying and simplifying access to various persistence support types, including relational and NoSQL databases systems. Spring Data's goal is to provide a stable and straightforward development model for data connectivity while preserving the unique features of the underlying database system. This is a broader project that includes several database-specific subprojects. The main features of spring data are-

- Concepts for custom object mapping and a robust repository.
- Basic properties are provided by the execution domain base classes.
- Derivation of dynamic queries from resource method names.
- Transparent modification history is supported.
- Spring MVC controllers are integrated in a more advanced way etc.

Spring data has several modules. One of the most popular spring data modules, Spring Data JPA (Java Persistence API), makes it simple to set up JPA-based repositories.

**Spring Data JPA:** It is a Spring Data extension module. It enables the formation of entities and repositories using the Java Persistence API (JPA) as the default Hibernate implementation. This provides Object Relational Mapping (ORM), dynamic requests, and the already existing Pagination functionality. (Walls C. , 2016)

### 2.2.2 Spring Cloud

Spring Cloud is indeed a framework based on Spring Boot, a platform that was established to make it simple to develop applications. Spring Cloud connects Spring Boot to various third-party libraries and frameworks, including Netflix components, Consul, and Zipkin. Every one of those libraries/frameworks implements specific functions in the Spring Boot application, such as load balancing and service discovery. Spring Cloud is focused on offering a pleasant out-of-the-box experience for everyday use cases, as well as an extensible mechanism to accommodate other scenarios. The following are the main characteristics of the Spring cloud.

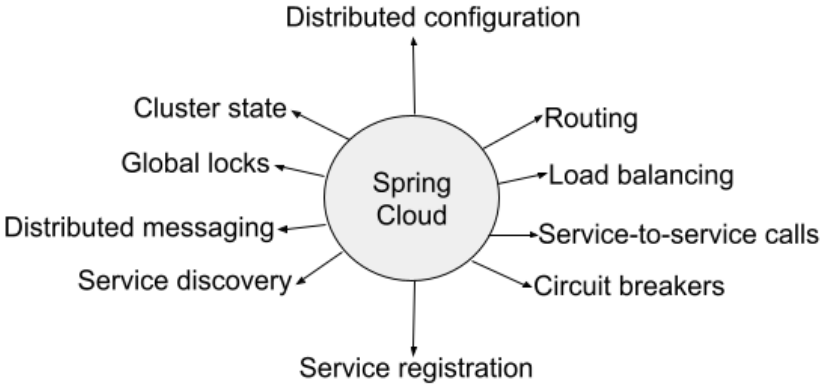


Figure 2.2.2: Features of Spring Cloud

In this thesis project, few of spring cloud features will be implemented.

### 2.2.3 Spring Security

Spring is a well-known and commonly used framework in the Java environment. Spring Security is just one of those frameworks. Spring security uses basic servlet filters to offer authentication and authorization to an application. Because they are open to anybody who uses the internet, web apps are prone to security and attacks. Because web apps are open to anybody who uses the internet, they are vulnerable to security threats and attacks. Some REST URLs (Uniform Resource Locator) may have access restrictions for individual users to keep the web resource secured. In this situation, Spring security can be used to secure the URLs.

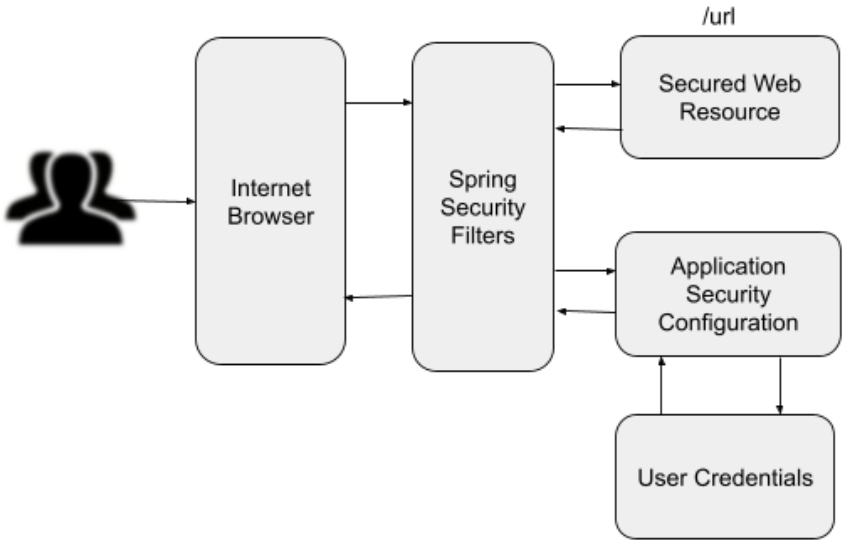


Figure 2.2.3: Spring security overview

Spring security overview has been shown in the above figure. When a user tries to access a protected resource through a web browser and receives an error message. Spring security comes into play in this situation. The spring security filters will intercept the queries, pre-process them, and then determine whether they have permission to access the protected web page. It will examine the security configurations of the application as well as the user's credentials. The user can access the online resource if they are authenticated and has the necessary authorization. All of this happens in the background. The basic notions that spring security is based on are authentication, authorization, password storage, and servlet filters. In this thesis project, spring-security OAuth2.0 (Open Authentication) will be used for the user authentication and authorization. (Spilcă, 2020)

### 2.2.4 OAuth2.0

The term OAuth stands for Open Authorization, and the version is 2.0. OAuth2.0 is an authorization framework that allows applications to access user data and conduct tasks on their behalf. OAuth2.0 is now an industry-standard authorization system that defines specialized authorization flows for the many software applications that software developers develop. For instance, there is an authorization flow suitable for Web apps and another that is suitable for mobile applications or smart devices. As a result, there is no single authorization method; instead, different authorization methods exist for various sorts of apps that developers design. A delegated authorization framework is what OAuth is also known as. Users can use OAuth to allow a third-party application to access their data or execute particular activities without revealing their username and password. And the good news is that they can only have

restricted access to a user's data or undertake a limited set of operations on the user's behalf with the help of OAuth authorized applications. In earlier days, before using the OAuth framework, the log in to the social network worked as follows:

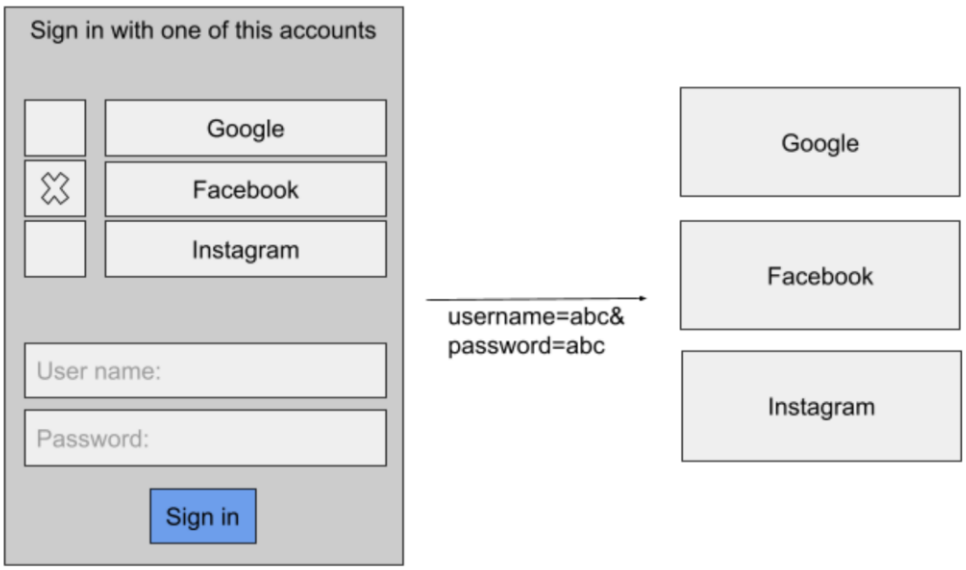


Figure 2.2.4.a: Social network log in system in earlier days

The user was given the option of using one of their social media accounts to log into an application. The user will then put their username and password for the chosen social network. The mobile application or website then uses the username and password to submit an HTTP (Hypertext Transfer Protocol) request to a social network that the user has chosen. The username and password would be stored in the tenant's database to access user data whenever they needed it. Because the user provided their username and password to the third-party program, this method was highly insecure. And now that the third-party application has the user's credentials, it can access any of the user's data at any moment. The chances of a user's login and password being stolen are incredibly high in this scenario. OAuth framework was designed to solve this problem.

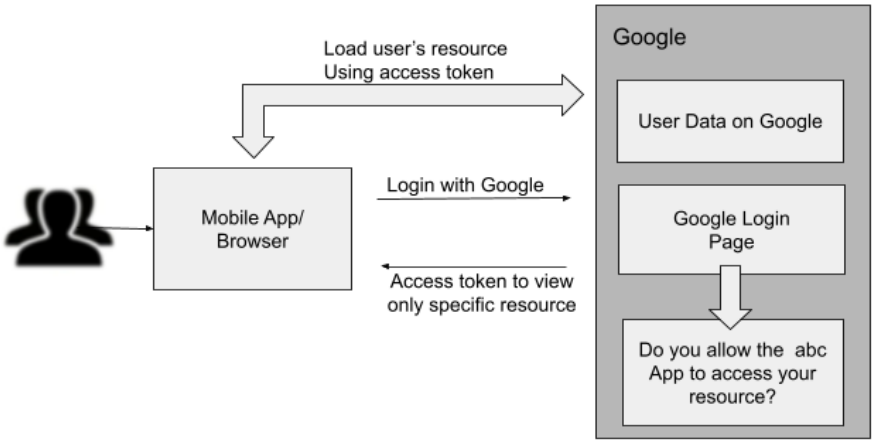


Figure 2.2.4.b: Google login using OAuth framework

Using the OAuth framework, the user will be sent to a Google login page instead of submitting their username and password to a third-party mobile application or website. As a result, the user will log in using their Google credentials via the Google login page, where the user will enter their username and password into a Google-designed and-created login page. The third-party mobile application does not have access to the user's Google password. If the login is successful, the authorization server, in this case, Google, will ask the user if they want this Third-Party application to see their data. If the user clicks the Allow button and authorizes permission, Google will send a unique access token to the third-party mobile app, allowing it to access user data. Because the user does not disclose their username and password to a third-party application, this method is highly secure. Because the third-party application is unaware of the user's actual login and password, it cannot save it in its database or have unrestricted access to any personal information. Instead, it can only access a limited collection of user data for a limited period. (Spilcă, 2020)

**Actors in OAuth:** OAuth contains four actors and they are resource owner, client, resource server and authorisation server.

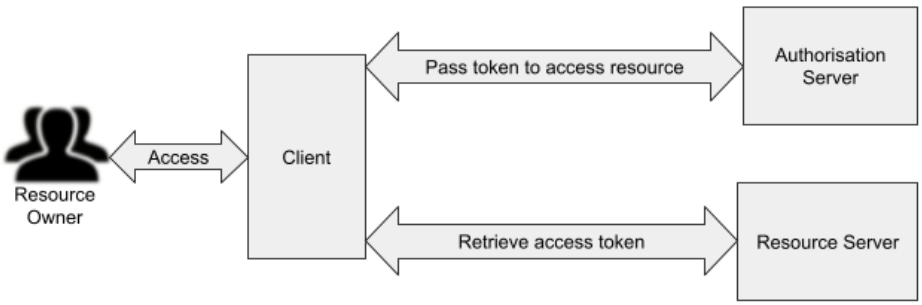


Figure 2.2.4.c: Actors in OAuth framework

**Resource Owner:** Resource Owner is the initial actor in the OAuth framework. The information or data being accessed is the resource, and the owner is the person who owns that information. If a person holds the information that is being accessed, then the resource owner is a user. As a result, a resource owner is a user who owns data and may grant access to it.

**Client:** The second actor of the OAuth framework is the client. A client is an application that accesses information on behalf of a user. It can be a Web application, a mobile application, or simply a Postman HTTP client that sends an HTTP request to retrieve the user's data. As a result, a resource owner is a user who accesses their data on the resource server via a client application.

**Resource Server:** A resource server is a server that is hosting user information. It might be a Google server, a Facebook server, or the developer's server where user data is stored. A resource server is not always hardware because the user's information is usually accessible via an API. When a developer creates RESTful microservices, each microservice in the application is treated as a resource server.

**Authorisation Server:** The fourth actor in the OAuth framework flow is an authorization server. After successfully verifying the resource owner and getting authorization, the authorization server issues access tokens to the client application. Google, Facebook, Twitter, and Microsoft, for example, all have their authorization servers. As a result, developers can add features that allow users to log in to their apps using their Facebook, Google, or Twitter accounts. Developers, on the other hand, can set up their authorization servers. Keyclaok Authorization Server will be used as an authorization server in this thesis project. (Senthilvel, 2017)

**OAuth Tokens:** The Access Token and the Refresh Token are the two types of tokens used in OAuth 2 authentication.

**OAuth Access Token:** The client application uses the access token to make requests to secure resources. When a user allows an application, the authorization server issues an access token to the client. An appeal is performed on behalf of the user because this token is related to the user, which is used as authorization criteria for accessing secured resources on a server.

**OAuth Refresh Token:** Usually, the refresh token is sent along with the access token. When the previous access token expires, the refresh token would be used to obtain a new one. The client supplies the refresh token rather than the usual grant type and receives a new access token. The use of refresh tokens provides for a short expiration period for resource server access and a long expiration period for authorization server access.

**OAuth2.0 Client Types:** The authorization server must identify which client would be registered while registering it. Two categories of clients are defined by the OAuth2.0 framework: Confidential client and public client.

**Confidential Client:** A confidential client is a program that keeps a client's password secret from the global community. The authorization server provides this client password to the client application. To prevent fraud, the authorization server uses this credential to identify the client.

**Public Client:** A public client is software that is incapable of maintaining the confidentiality of client credentials. For example, a mobile phone app or a desktop application that includes the client's credentials. An application like this might be cracked, revealing the password. A JavaScript program functioning in the user's browser would be the same.

**OAuth2.0 Grant Types:** The grant type determines how an app obtains an access token. Because many kinds of apps exist, and some apps are more secure than others, several OAuth Grant types were created, and each grant type is used to obtain an access token in a unique way.

**Authorization Code:** This grant type is commonly utilized in both server-side and mobile applications.

**Client Credentials:** This grant type is commonly used within server-side applications with no user interface and is not accessible to the end-user directly.

**PKCE Enhanced Authorization Code**: he PKCE Enhanced Authorization Code grant type is an addition to the authorization code grant type. This grant type is intended for usage in applications that are unable to keep client credentials private. JavaScript applications, for example, or mobile applications are examples of this.

**Authorization Code:** Typical web applications could use Authorization Code Flow, which trades an Authorization Code for a token because they are server-side programs with no publicly accessible source code. (Justin Richer, 2017)

## 2.3 Angular

Angular is one of the most popular web application frameworks built on Typescript for developing SPA (Single Page Application). It claims a set of developer tools to assist developers in developing, building, testing, and updating a large and complex application. The main feature of Angular is that it allows developers to create apps that run on almost any platform, including mobile, web, and desktop. It is a

component-based application framework for building scalable applications. Angular comes with a set of very well libraries that offers a broad range of functionalities, such as routing, form handling, client-server communication, and much more. (Angular applications: The essentials, 2018)

### 2.3.1 TypeScript

For Angular software development, TypeScript is the primary language. It's a code editor with type safety and tooling support at design time.

TypeScript cannot be run natively in browsers. The TypeScript compiler must be used to transpile TypeScript into JavaScript, which necessitates some settings, such as tsconfig.json—TypeScript compiler settings, typings—declaration files for TypeScript. The TypeScript compiler settings for this thesis project looks like-

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/out-tsc",
    "sourceMap": true,
    "declaration": false,
    "downlevelIteration": true,
    "experimentalDecorators": true,
    "moduleResolution": "node",
    "importHelpers": true,
    "target": "es2015",
    "module": "es2020",
    "lib": [
      "es2018",
      "dom"
    ]
  }
}
```

Figure 2.3.1: TypeScript Compiler settings, tsconfig.json

### 2.3.2 Angular CLI

The Angular CLI is a command-line interface (CLI) for automating the application development process easily and quickly. It allows developers to-

- create an Angular application from scratch.

- run a development server that supports live reload.
- execute the unit tests for the application.
- insert new features to the already created Angular application.
- build the software in preparation for launch to production.

Node.js and npm (Node Package Manager) must be installed on the developer's PC before Angular CLI can be used.

```
npm install -g @angular/cli
```

The above-mentioned command is used to install Angular CLI. To create a new Angular project using the command-line interface, the command is-

```
ng new application-name
```

### 2.3.3 Modules

Angular applications are called modular applications. A module in Angular application is a framework for combing related components, services, directives, and pipes so that they can be coupled with other modules to form an application. Every module may have its own logic and metadata. There are five types of logic and metadata-

- bootstrap - it tells Angular which component the app should be bootstrapped from.
- declarations - it specifies which components are accessible for use in a given module.
- imports - it allows exported modules to be used in other modules in the current module.
- exports - it allows different modules to use logic from the current module and
- providers - it embeds services that other modules in the current module needs. Directives, services, and pipes are examples of these.

In a typical Angular application, six different types of modules can be seen. However, among them, AppModule and RoutingModule are very common modules in Angular applications.

**AppModule:** In an Angular application, this is the only mandatory module. AppModule is the bootstrapped core module that allows the program to run. Here is a simple AppModule example-

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { ExampleComponent } from './component/example/example.component';

@NgModule({
  declarations: [
    AppComponent,
    ExampleComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Figure 2.3.3: AppModule example

**RoutingModule:** It is an important part of Angular application where the routes of applications need to be configured. If an Angular application is generated by Angular CLI, it automatically creates routing module, otherwise routing module need to be created manually if needed. Here is a simple Routing-Module example-

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = []

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Figure 2.3.3: AppRoutingModule example

All the routes of the application need to be placed in the Routes array.

### 2.3.4 Components

Components are an important element of angular. Every angular component has an HTML template that specifies how the component will appear on the page. A Typescript class that determines the functionality and as well as a CSS selector which specifies how the component is utilized in a template, as shown in the diagram below.

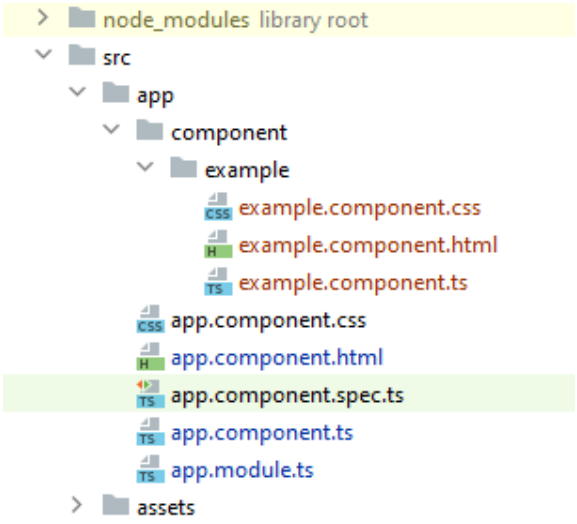


Figure 2.3.4: Angular component structure

The Typescript class example.components.ts, the HTML template example.component.html, and the CSS selector example.component.css are shown in Figure 2.3.4. The AppComponent is the root component of any angular application. All the other components must be nested into the root component.

### 2.3.5 Directives and Data Bindings, Interpolation, Event Binding, Property Binding

**Directives:** Directives are the most significant part of an Angular application, and the component, the most commonly used Angular unit, is basically a directive. Component, attribute directive, and structural directive are the three different forms of Angular directives. Components are nothing more than directives with templates attached to them. Attribute directives change the behavior and look of the document object model (DOM). The conditional style is applied to the elements. Structural directives are designed to construct and destroy DOM elements effectively.

**Data Binding:** Communication can be equated to data binding. The component's TypeScript code communicates with the component's template using data binding. This interaction might be one-way or two-way. There are four types of data binding..

**Interpolation:** String interpolation is one-way data binding in which content is placed between a pair of curly brackets. It frequently refers to a component property by its name. The value of the corresponding component attribute is replaced by Angular. The syntax example of string interpolation is given below-

```
<span>{{ title }} app is running!</span>
```

Figure 2.3.5.a: String interpolation syntax.

The title attribute will be replaced with its string value as shown in Figure 2.3.5.a.

**Event Binding:** In an angular application, event binding involves listening for and based on the user activities such as mouse clicks, keypress, movements of mouse and touches, it responses. Syntax example of event binding-

```
<button (click)="onSubmit()">Submit</button>
```

Figure 2.3.5.b: Event binding syntax

The destination event is named click, and the template expression is onSubmit (), as illustrated in Figure 2.3.5.b.

**Property Binding:** Property binding is a one-way communication system. It allows for value changes at any time, however at the component level. Bind to a property of an element by enclosing it in square brackets, [], which designates the property as a goal property. The syntax example property binding is given below-

```
<img [src]="imageUrl">
```

Figure 2.3.5.c: Property binding syntax.

As shown in Figure 2.3.5.c, the [src] property is the name of the element property <img>.

### 2.3.6 Routing

Routing helps users to navigate from one page to the next while doing actions in an Angular application. A browser URL can be regarded as a request for a client-generated display. The Router can be tied to a link on a website, and it will route to the corresponding information view when a user clicks it. (Wilken, 2018)

## 2.4 Keycloak

Keycloak is a well-known identity and access management server that supports OAuth2.0 and OpenID connect authorization procedures. It is free and open-source. It can be downloaded and executed on any server as a standalone application. In OAuth flows, there are four different participants.

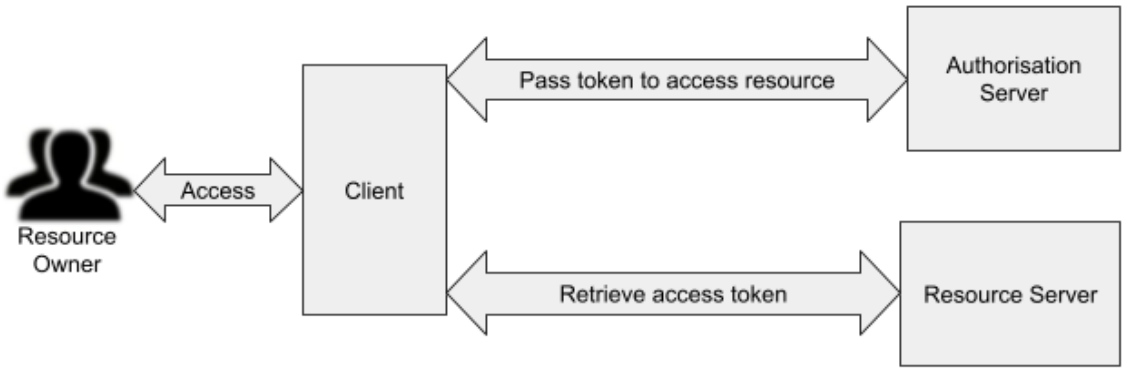


Figure 2.4.a: Participants in OAuth flows.

A resource owner who is also a user. The client application, which might be either a website or a mobile app. The resource server could be a spring boot application or a small microservice running behind an API Gateway. Finally, an authorization server is used to authenticate the resource owner or user using their username and password and provide an access token to the client application. This is where Keycloak came into play. It can be used as an authorization server with the Spring boot application. The applications are not required to deal with login forms, user authentication, or user storage.

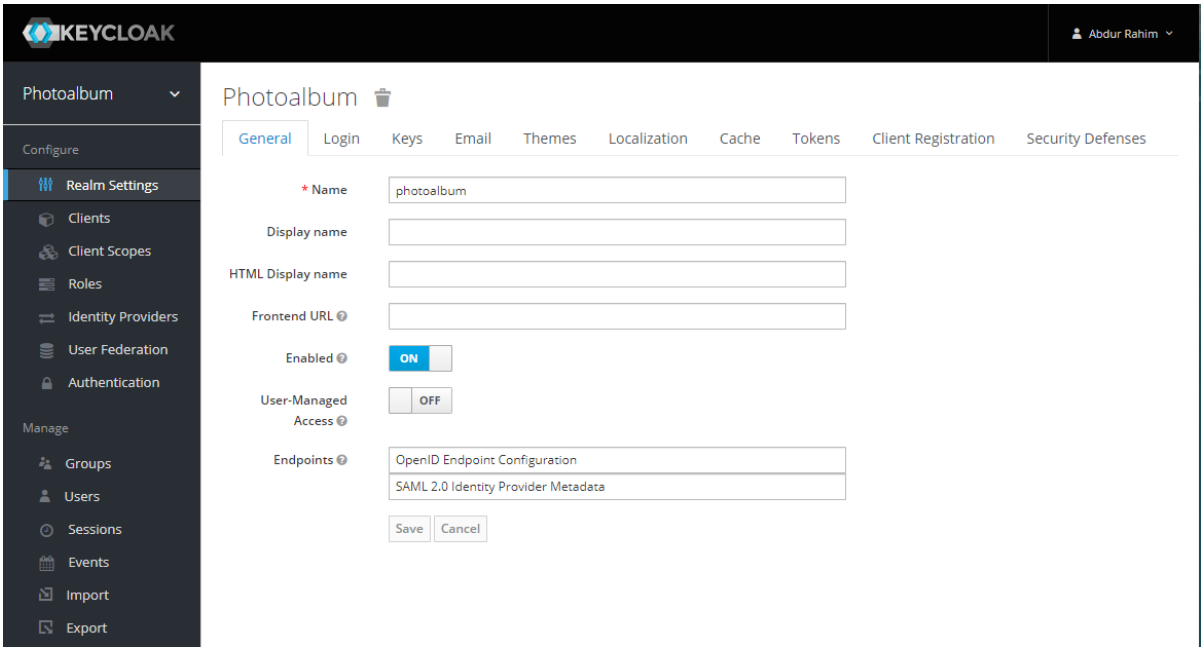


Figure 2.4.b: Keycloak administrator dashboard.

Keycloak has an excellent web interface for managing OAuth client applications, users, their credentials, and user rules. It has a powerful administration dashboard with a lot of capabilities that let manage a lot of things from a single online interface.

### 2.4.1 Working Methodology of Keycloak

The working methodology of Keycloak has been shown in the below figure-

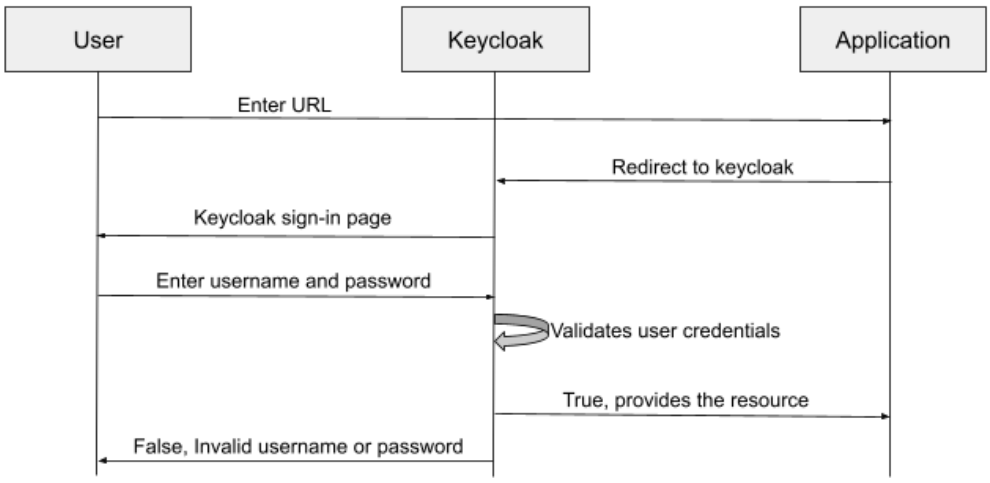


Figure 2.4.1: Keycloak working principles.

The application's base URL (Uniform Resource Locator) needs to be stored in the Keycloak administration console as a redirect URL. When a user enters the base URL in a browser to access the application's resources, the user is redirected to Keycloak and to the Keycloak login page to authenticate the user. The user must put his username and password on the Keycloak login page, after which the user will be redirected back to the program with a code that is only valid for a short period. The application uses this code and the client ID and client secret to communicate with Keycloak, and Keycloak responds with an access token, an ID token, and a refresh token. Only one of these tokens is required for the application to determine which rights the user has, and the user will be permitted or refused access to a particular protected URL based on the claims.

### 2.4.2 Authentication

Authentication is the procedure of authenticating a user by requiring them to submit a valid username and password before being permitted access. Both the server and the client rely on it to validate a user. Suppose a user tries to access data from the server, and the server first must know who is trying to access the data before providing it. In that case, the server utilizes an authentication process. The client uses it to verify that the server is the one that it claims to be.

### 2.4.3 Authorization

The procedure of providing user permission to do something on the application, databases, server, etc., is known as authorization. It is a method of determining whether or not a user has the privilege to utilize a resource. After someone's identification has been successfully verified, authorization takes place. It is about allowing the partial or complete right to the resources such as databases, servers, and other important information in order to complete the task in an application.

### 2.4.4 Single-Sign On (SSO)

Single sign-on and sign-out are supported by Keycloak, which means that once a user has logged in to one application, they will not need to log in to another. Furthermore, if a user logs out of one program, they will be logged out of the others as well. When using SSO, users have to enter their passwords once, but they are always logged in.

### 2.4.5 Social Login

Keycloak also supports social login, which means that users can log in to the app using their Facebook, Google, Twitter, or GitHub accounts, among others. Keycloak can also use existing OpenID connect or

SAML (Security Assertion Markup Language) identity providers to authenticate users. To make it function, there is not much effort to be done.

### 2.4.6 User Federation

If an organization already has a user database, Keycloak can be integrated with it. Keycloak supports existing LDAP (Lightweight Directory Access Protocol) and Active Directory servers and an own provider can be implemented, which may be used to link Keycloak with an existing MySQL database example.

# 3 Requirements

This chapter will go through all of the requirements for this thesis assignment. There will be parts of the requirements that must be completed to a particular level, but there will also be parts where full completion is not required. Because a prototype application will be developed for this thesis project, it will not be deployed for real-world use. Only the application's core functionality will be incorporated in the existing application. As a result, just the project's core functionality must be developed. There are two sections to the overall requirements. The requirements section has been split into two sub sections which are functional requirements and non-functional requirements.

## 3.1 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. (Jaeckel, 2018)

### 3.1.1 User Data

One of the main features of the application is that the user can login to the system. To login to the system the user credentials must be stored in a database beforehand. In the database the minimum data fields that the user will be needed while login is username or user email and user password. For the security purpose the user password will be stored as bcrypted form. For example, if the user password is "rahim", in the database it will store as-

"$2a$10$.Vd8scGxtLHkwHPbrVcn6usVgc2XWFrFZjNfzIJyDIIE6lwYU2Gqu" which is bcrypted form of "rahim".

### 3.1.2 User Authentication and Login

The primary goal of the project is to verify the user's identity by logging into the system. The user authentication procedure is as follows: the user accesses the application's base URL in the browser. To authenticate the user, the user will be sent to the authorization server authentication page. To be authenticated, the user must enter his or her username and password. If the user is authenticated, the user will

be logged in to the system and will be able to view the application's protected homepage. If the authentication fails, the user will receive an error message stating that the credentials are incorrect. Without authentication, the user will be unable to access any of the application's protected resources.

### 3.1.3 User Can Log Out

A user should be able to log out of the system if they choose to. The user must first log in before attempting to log out. When a user logs out, he/she will not be able to access any protected resources of the application again until he logs back in. Once the user presses the log out button, he/she will be forwarded to the authorization server authentication page, where he or she can log back in.

### 3.1.4 Authorization

To indicate that a group of users is authorized to do or see particular things in the application, the system employs a configurable authorization constraint structure. Without proper authorization a user cannot do or access the protected resources of the application. There should be functions that can be only perform by certain kinds of user with granted permission. For example, a user of the system has the authority to view the homepage of the application. However, he/she cannot modify any protected data because of lack of authorization. On the other hand, an admin can create and edit existing feature of the application. Though he/she do not have authorities to delete any data from the database. However, the super admin has all the authorities of system including deleting permission.

### 3.1.5 Data Storage

Another core requirement of the system is data storage. A service's data should be able to be saved by the system. The application should provide the opportunity to create new objects of the entities of the project. The objects of the entities should be store in a table in the database. There should be possibilities to modify the existing data when required. If a data is no more required, the data could be deleted from the database according the user authorizations. Each service of the application should have separate database as well.

## 3.2 Non-Functional Requirements

The creation and management of a software system, as well as its fundamental usability and resource utilization, are heavily influenced by non-functional requirements. They even have an impact on an

application's performance and structure. Non-functional requirements become increasingly relevant as a software system grows larger and more complicated.

### 3.2.1 Performance

In non-functional criteria, performance of the system is the most important requirement, and it influences practically all of the others. This is usually the crucial factor when selecting an application for a given activity. As a result, when a new activity is initiated and all the data associated with it needs to be loaded, a quick response is expected. A suitable database must be connected, requiring the least amount of information transfers and a quick querying time.

### 3.2.2 Reliability

For a system to be welcomed by a user or maybe another system, it must be reliable. As a result, if the software's reliability is low, the system will provide no benefit and will be unusable. As a result, the system's proper functioning must always be guaranteed. If a mistake occurs, a safe transition must be ensured. It would be an inexcusable error if a malfunction resulted in the disclosure of information to third parties.

### 3.2.3 Extensibility

In the software development systems, the current systems are being extended or modified more frequently. This may be essential if the present industry standards change or the system environment requires new functions. As a result, a system's complexity might swiftly rise, affecting its overview. Nonetheless, a developer should be able to adjust the system's dependability, security, and usability as simply as possible. Interfaces should also be developed so that other functionalities can be included to the system.

### 3.2.4 Safety

Safety must also be a core condition for the system's operation. In the system to be developed, two areas of security must be mentioned directly. On the one hand, in terms of data processing, and on the other, in terms of user data. Both elements of security are critical and must remain hidden from outsiders. It would be dangerous if someone other than the owner of the database could access and change the stored data. To prevent this, it is vital to adhere to the most up-to-date security standards in order to secure data from unwanted access.

### 3.2.5 Maintainability

The other non-functional condition that all software must meet is maintainability. This feature shows the average time, as well as the degree of flexibility with which a system can be recovered following a failure.

# 4 Design

This chapter will go over the many procedures involved in designing the various elements of the application.

## 4.1 System Overview

PhotoAlbum is a prototype application which has been developed for the thesis project. The authentication and authorization of users is the main emphasis of this program. As a result, the application's other features did not receive nearly as much attention. The block architecture of the entire system looks as follows-

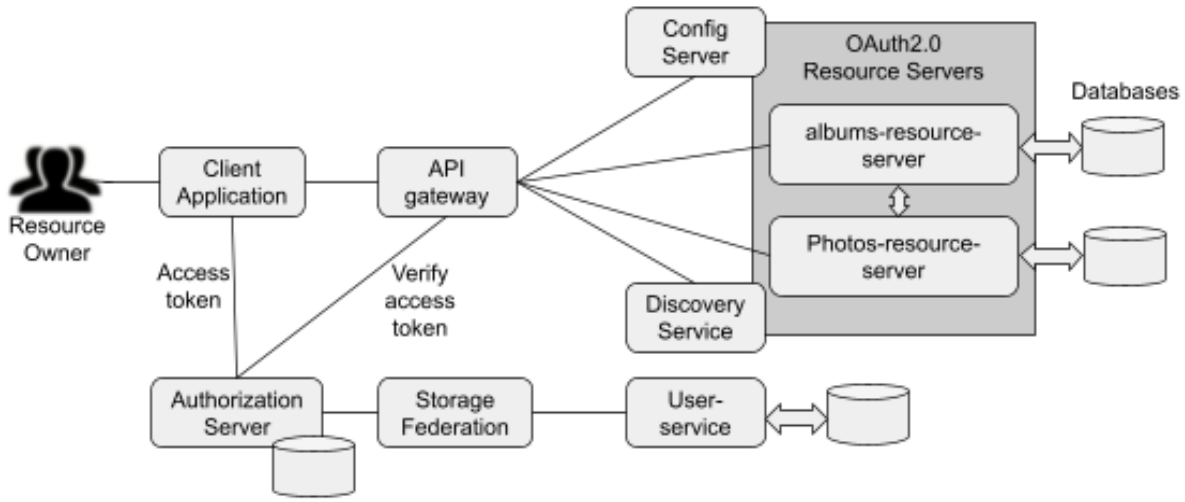


Figure 4.1: System block overview

Architecture has been considered in this application in which a resource owner, who is actually a user, uses a client application to obtain resources from the system's resource servers. The user is redirected to the Authorization Server to be authenticated and obtain an access token by the client application. The client application will then interact with the resource servers using this access token to perform an operation on behalf of the user. There are two resource servers, album-resource-server, and photos-resource-server in this application. In a real-world application, however, there could be hundreds of such resource servers. Each resource server is a small, self-contained Spring boot application with its own port number. In modern applications, these small standalone spring boot applications are known

as microservices. Additional services can be launched to help the application continue to run and respond to incoming requests if one microservice becomes overloaded with traffic or goes down. As a result, a new architectural style emerges. When a server-side application has a lot of resource servers or microservices, they're put behind an API gateway, which works as the only point of entry for all of them. The client application will not have to cope with various port numbers of different resource servers. In this application, the API gateway runs on port 8090, and the client application communicates with the resource servers over the API gateway's single port, which 8090. The API Gateway accepts the incoming HTTP requests and routes them to the appropriate microservices. The API gateway has been set up to work with the discovery service and to deliver requests evenly across numerous instances of different microservices using load balancing. The resource servers have been set up so that each one runs on a unique port number, allowing many instances of the same resource server to run on different port numbers. Now, Because every resource server is set up as a Eureka client, when a resource server starts up, it will register its location with the Eureka Discovery Service, giving Eureka access to the IP address and port number of the individual resource server instance. The location of each of these resource services will be retrieved from the Eureka discovery service by the spring cloud API Gateway. In the spring cloud API gateway, routes have been configured to use a special load balancing prefix along with the resource server name, and each resource server has been registered with the discovery service. This will cause the spring cloud API Gateway to use its built-in load balancing client to distribute incoming HTTP requests evenly among the resource server's running instances. The authorization server, the user-service, and the user-storage provider do not go behind the API gateway. The eureka discovery service helps the microservices to discover or find each other. The user-service and the user-storage-provider are implemented to authenticate the user by an authorization server whose user credentials are stored in an external database. In the real-time application, every system has its own remote database to store user credentials. To authenticate these users, user-storage-provider comes into play. In this application, user-service is a standalone application. It has its own database, which stored the user credentials. However, the authorization server can authenticate these users using user-storage-provider. Another block of the application config-server. The spring boot config-server is implemented to centralize the resource server's common configuration properties. Centralized configuration is a design in which all service configurations are controlled in a single location rather than dispersed across multiple services. On initialization, each service gets its properties from the centralized repository. In this application, from the GitHub repository.

## 4.2 Functional Architecture

The application's functional architecture describes how the resource owner can get protected resources from the resource servers. There are six participants in this application, and they are resource owner who is the user, client application, API gateway, authorization server, resource server A, in this case, albums-resource-server, and resource server B, which is in this case, photos-resource server. The process begins with a user requesting a resource through a client application via an API gateway. Say a user requests for a resource from the albums-resource-server, the API gate receives this request and route it to the respective resource server. Because the albums-resource-server is configured as a resource server, it uses an identity provider to verify the provided access token. The user will receive a 401-unauthorized status code if the access token is invalid or not provided.

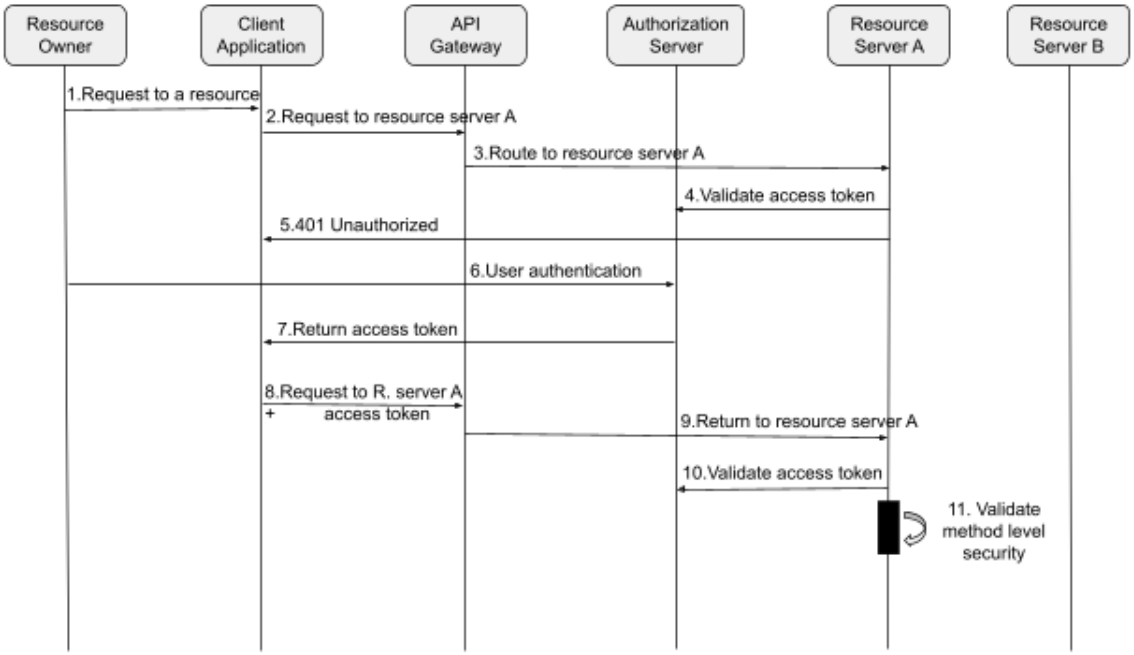


Figure 4.2: System functional architecture

To obtain an access token from the identity provider server, the client application must follow one of the authorization code flows, in this case, the PKCE (Proof Key for Code Exchange) authorization grant flow, and the user must authenticate during this process. The client application receives an access token after the user authentication is completed according to the authorization code flow. When the client application has received an access token, it will use the API gateway to send another request to the albums-resource-server. However, this time it will include an access token in the authorization header. The albums-resource-server will validate the access token with the identity provider server, and the

spring cloud gateway will route this request to them. Suppose the identity provider successfully validates the access token. In that case, the albums-resource-server will process the request and perform more granular authorization by running method-level security expressions and determining whether the user to whom the access token was issued has the authority to perform the requested operation. If all goes well, the request will be processed, the operation will be completed, and the albums-resource-server will respond.

## 4.3 System Use Cases

Because the prototype application's primary goal is to ensure the application's security, there aren't many use cases. There are three types of users: guests, admins, and super admins. Those who are just authenticated but have no unique role in the application are called guests. The super admin gets the superadmin role, while the admin gets the admin role. Each user group interacts to specific scenarios. Before gaining access to any resources, all users must first authenticate.

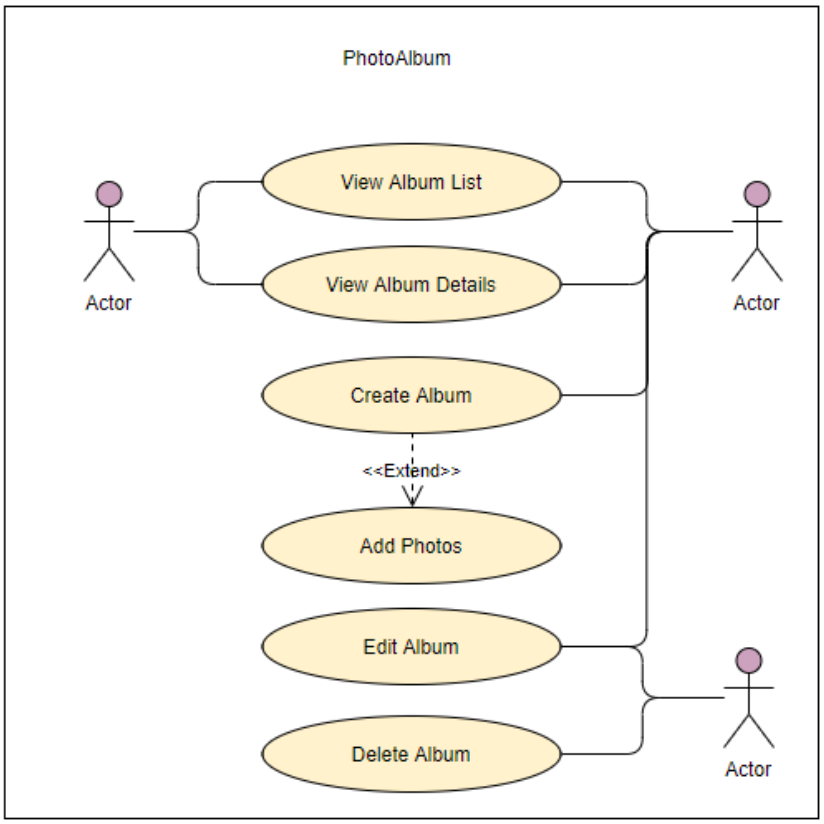


Figure: Use case diagram

When a user is a guest, they can access the album list after logging into the system. Guests can also view the details of the induvial album. When a user has the admin role, he has access to the album list and individual album details. In addition, the admin has the authorization to create new albums. The administrator can add photos to a new album after it has been created. The administrator can also add images to an album that already exists, they can edit the existing album as well. The admin, on the other hand, does not have the authority to delete an album. The only user group in the system that has authorization to all use cases is super admin. The super admin has access to the album list and details, as well as the authorizes to create new albums and modify existing ones. Furthermore, only the super admin has the authorization to delete an unnecessary album from the system's database.

## 4.4 Database Design

The most of microservices in a microservice software architecture pattern require data to be stored in a database. In this prototype application the albums-resource-server stores the records of the album and the photos-resource-server stores the information of photo in the database. There is also a standalone spring boot application which is user-service, which stores user data into a database. In microservice architecture pattern one of the key requirements is the microservices must always be loosely connected to each other so that they could be independently created, deployed, and scaled.

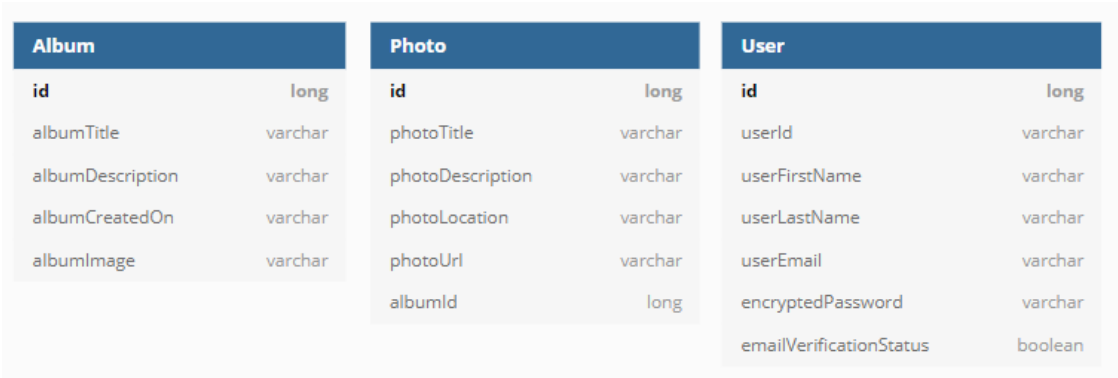


Figure: Database design

The persistent data of each microservice has been kept private to that microservice and is only reachable via its Application Programming Interface (API). The transactions of a microservice are limited to its database. The database for a service is essentially part of the service's functionality. Other microservices services are unable to access it directly. For the simplicity of the prototype application the MySQL database has been implemented.

# 5 Implementation

This chapter has presented the system's implementation, which is based on the preceding chapter. Here the most important code snippets of the project as well as the needed configurations will be described.

## 5.1 Authorization Server Setup

There are four parties in the OAuth2.0 authorization process: the resource owner, client application, resource server, and finally, an authorization server that allows the resource owner or user to log in with their user name and password and issue an access token to the client application. This is when Keycloak comes into play. There is a multitude of alternative authorization and identity access providers to choose from. Among them, however, Keyclaok is one of the most popular authorizations and identity access providers. Keycloak can be configured in a variety of ways. Since this is a prototype application, Keycloak was downloaded as a standalone application. Keycloak version-12.0.4 was downloaded for this project. To start the standalone application, opened a command-line window and navigated to the Keycloak bin folder, then run the ./standalone.sh file as follows-

```
RahimAbd@N0820074 MINGW64 ~/Desktop/Akash/Software/keycloak-12.0.4/keycloak-12.0.4/bin
$ ./standalone.sh
```

Figure 5.1: Command to start Keycloak server

By default, the Keycloak server runs on port 8080. Once the Keycloak server is up and running, it can be accessed by going to http://localhost:8080/auth via an internet browser. The first step is to create a Keyclaok administrator account, which will be used to create additional realms, users, and roles, among other things. The administrator has now entered the Master realm, which has been pre-defined. The topmost ranked is the master realm. This realm's admin users have the ability to observe and administer any other realm. (Server Installation and Configuration Guide, 2019)

### 5.1.1 Create New Realm

After login to Keycloak, to handle an application and users, it is best to construct a new realm. By selecting the drop-down icon near to the Master realm in the top left corner, a new realm entitled photoalbum is created as follows-

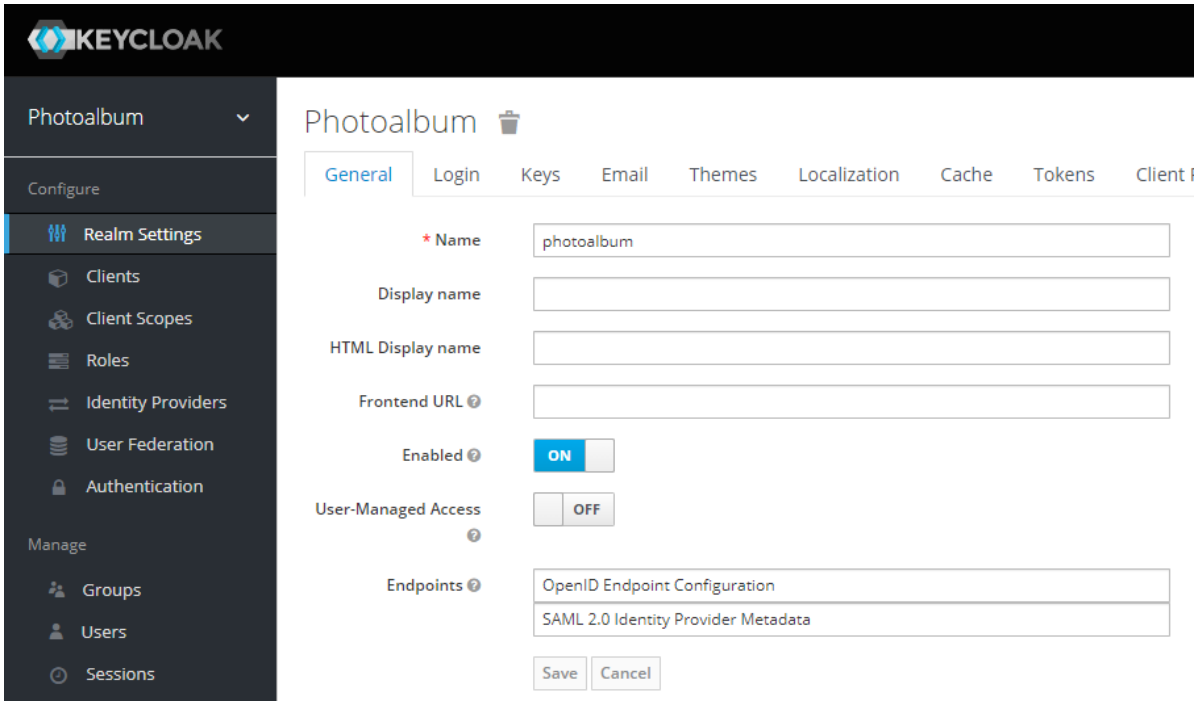


Figure 5.1.1: Screenshot of Keycloak dashboard on creating new realm

It is required to input the realm name while creating a new realm, and then the realm is enabled. The rest of the realm creation form's fields are optional.

### 5.1.2 Create New Client

Each application that communicates with Keycloak is referred to as a client. It is mandatory to create a client after creating the new realm. By clicking on the Clients menu from the Keycloak dashboard sidebar, a new client id named photo-album-client was created.

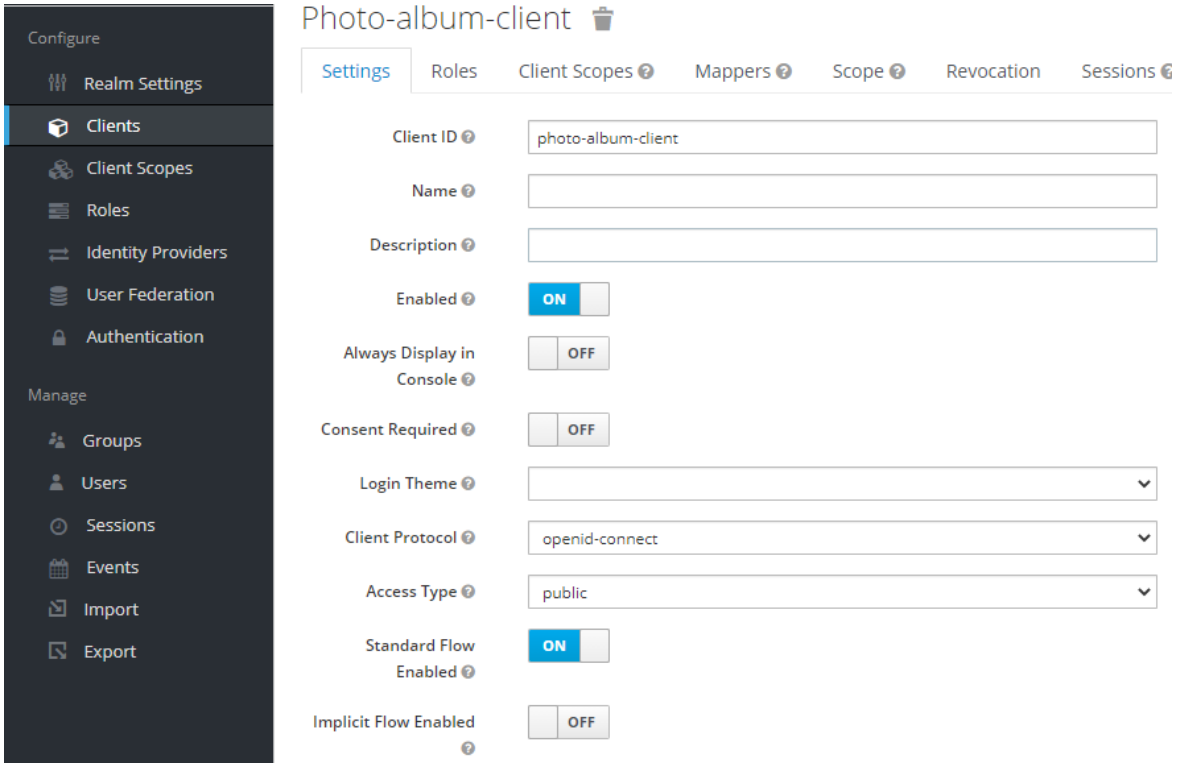


Figure 5.1.2.a: Keycloak dashboard on creating new client id

While creating a new client id, it has been ensured that the enabled tab is on, the client protocol is openid-connect, the access type is public and the standard flow is enabled.


Figure 5.1.2.b: Keycloak dashboard redirect URL while creating new client id

The valid redirect URL is a required field, even though the Root URL can be left blank. Because the front end was built with Angular, the valid redirect URL can be either http://localhost:4200/* , which is the default Angular URL, or *, which indicates any redirected URL is valid. The web origin field has been filled with the same value as the valid redirect URL.

### 5.1.3 Create Roles

Roles are used to offering the right authorization to a resource. When using the Spring Boot application to handle requests, the roles must be specified. It is not mandatory to create a role in Keycloak. However, because user authorization is a key feature of this prototype project, three roles were designed to meet the requirements: user, admin, and superadmin.

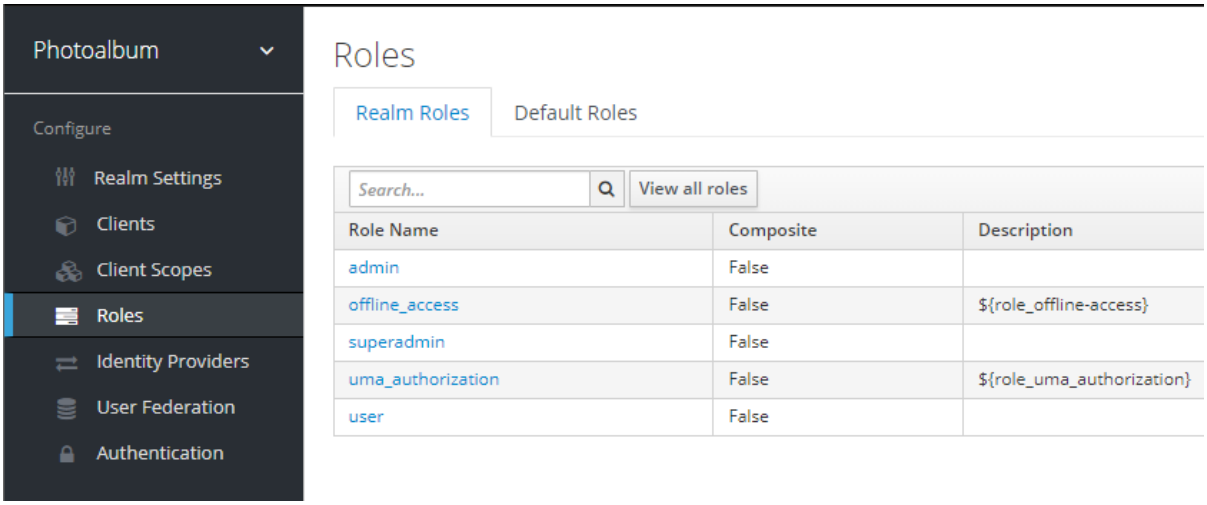


Figure 5.1.3: Create roles in Keycloak

The required roles were created by clicking the Roles menu from the Keycloak dashboard sidebar and then Add Role.

### 5.1.4 Create User

The final stage in this project's authorization server configuration was to create users and assign the appropriate roles to them. Users are typically stored in different databases in real-world applications. However, three users named Akash Sarker, Super Admin, and Test User were created in Keycloak for the sake of simplicity.

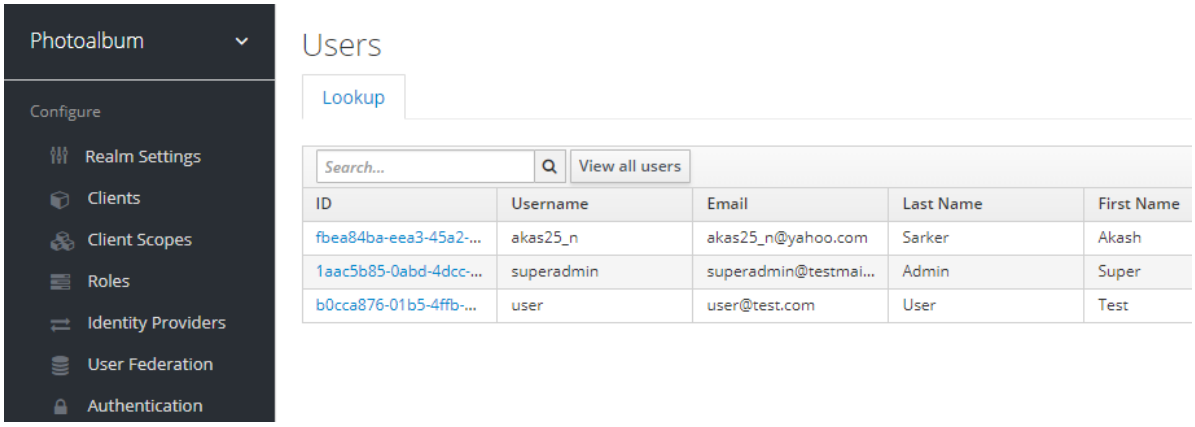


Figure 5.1.4: Create user in Keycloak

These users were created by selecting Add User from the Users menu bar on the Keycloak sidebar. Finally, the admin role was assigned to Akash Sarker, the superadmin role to Super Admin, and the user role to Test User. This concludes the keycloak configuration of the project. (Justin Richer, 2017)

## 5.2 Resource Servers Setup

In OAuth term, a resource server is a Spring boot application with which a client application can save, update, or retrieve information. And, in most cases, the application exposes some website endpoints to which the client application can send HTTP requests. This request is accepted, and a response is returned by the resource server. In this project, two Spring boot microservices were created to configure them as resource server. They are albums-resource-server and photos-resource-server.

### 5.2.1 Add OAuth2 Dependency

It was necessary to add a very required dependency to the newly created Spring Boot projects to act as a Resource Server and connect with the Authorization server to validate the access token. This dependency is- spring-boot-starter-oauth2-resource-server. The pom.xml files for both freshly constructed microservices were opened, and the necessary dependency was added to them.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

Figure 5.2.1: Spring boot starter oauth2 resource server dependency

There is no necessary to insert any more Spring Security dependencies to this pom.xml file because the preceding dependency already supports Spring Security libraries.

The spring-boot-starter-web was included as the only extra dependency to enable the resource servers to run as a RESTful Web Service application.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Figure 5.2.1.a: Spring boot starter web dependency

### 5.2.2 Create RestController Class

Rest Controller classes were created in both microservices to make the resource servers perform as a RESTful Web Service platform. A couple of web service endpoints in these Rest Controller classes provide plain text as a reply. AlbumsController was created in the albums-resource-server.

```
@RestController
@RequestMapping("/api/albums")
public class AlbumsController {

    private AlbumService albumService;

    public AlbumsController (AlbumService albumService) {
        this.albumService = albumService;
  }
```

Figure 5.2.2.a: Code snippets of AlbumsController

The Spring @RestController was used in the above code snippet, which designates the class as a controller, with each method returning a domain object rather than a view. The @RestController annotation consists of @Controller and @ResponseBody spring annotations.

@RequestMapping annotation is one of the commonly used spring annotations in web applications. HTTP (Hypertext Transfer Protocol) requests are linked to spring MVC (model-view-controller), and REST controller handles methods by this annotation. The @RequestMapping annotation is used in the class and also on the method level in a controller. Here the class level annotation @RequestMapping(“/api/albums”) means that this the root endpoint of this controller class. All the method level endpoints extend the root endpoint. List of all the method level endpoint including the root URL in the AlbumController RestController along with HTTP method type and a short description of the endpoints are mentioned below in tabular form.

| HTTP method | Endpoints | Description |
|---|---|---|
| GET | /api/albums/ | Retrieve the list of albums |
| GET | /api/albums/photos/albumId | Retrieve all the photos for the albumId |
| POST | /api/albums/ | Create new albums |
| GET | /api/albums/id | Find a single album by albumId |
| PUT | /api/albums/id | Update an album by albumId |
| DELETE | /api/albums/id | Delete an album by albumId |

Figure: 5.2.2.b: Endpoints list in the AlbumController

The PhotosController is also a RestController and the /api/photos is the root endpoint of the PhotoController. The PhotoController also have some method level endpoints.

```
@RestController
@RequestMapping("/api/photos")
public class PhotoController {

    private PhotoService photoService;

    public PhotoController (PhotoService photoService) {
        this.photoService = photoService;
    }
```

Figure 5.2.2.c: Code snippets of PhotoController

List of all the method level endpoints including the root URL in the PhotoController RestController along with HTTP method type and a short description of the endpoints are mentioned below in tabular form.

| HTTP method | Endpoints | Description |
|---|---|---|
| GET | /api/photos/ | Retrieves all photos |
| GET | /api/photos/albumId | Retrieves all the photos of an album by albumId |
| POST | /api/photos/albumId | Insert photos into the album by albumId |

Figure: 5.2.2.d: Endpoints list in the PhotoController

The spring-boot-starter-oauth2-resource-server dependency was added to both resource servers. As a result, all web service endpoints of the resource servers are secured. The client application must provide a verified access token that the Authorization Server created to interact with Resource Servers.

### 5.2.3 Validate Access Token

The following property was added to the application.properties file of the resource server to setup the Resource Server to verify the access token via Keycloak server.

```
spring.security.oauth2.resourceserver.jwt.issuer-uri=
http://localhost:8080/auth/realms/photoalbum
```

Figure 5.2.3: Issuer-uri configuration in the application.properties file

The issuer-uri parameter in this case refers to the “photoalbum” Realm on the Keycloak server which was created. If the issuer-uri value is not correct, the protected resources of the resource server cannot be accessed.

## 5.3 Data Persistence

As illustrated in the preceding section that in the microservice architecture pattern all the microservices usually have their own databases to avoid strong coupling among the microservices. For simplicity, in this prototype project, the MySQL database has been used. There are two resource servers in this project and a separate spring boot standalone project which is user-service. These three spring boot projects have three entities that were annotated @Entity. These entities created three different tables in the databases. To integrate the entities of the microservices of this project with the MySQL database, the

spring-boot-starter-data-jpa and the mysql-connector-java dependencies were injected in the respective microservices..

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

Figure 5.3: Spring data JPA dependency injection

For instance, the Album entity of the album resource server creates album_TBL table in the database.

```
@Entity
@Table(name = "albums_TBL")
public class Album {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;
    private String albumTitle;
    private String albumDescription;
    public String albumCreatedOn;
    private String albumImage;

    public Album() {
    }
```

Figure 5.3.b: Album entity of the album resource server

In the application.properties file of the respective microservices was configured as follows-

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/{database-name}
spring.datasource.username=user
spring.datasource.password=password
spring.jpa.hibernate.ddl-auto = update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=
org.hibernate.dialect.MySQL8Dialect
```

Figure: 5.3.c: Database configuration in the application.properties file

The database name in the spring.datasource.url property will be changed according to the microservices. For instance, for albums-resource-server, the name of the database is album_db; for photos-resource-server, it is photo_db. Hibernate leverages a JPA(Java Persistence API) based property provided by

Spring for (DDL) Data Definition Language. spring.jpa.hibernate.ddl-auto. The typical values for this property are create, create-drop, update, validate, and none. In this project, the value of this property was set to update. This means the hibernate compares the object model formed from the linkages to such existing schema and modifies the schema depending on the differences. Even though the program no longer requires the tables or columns, it still never deletes them.

## 5.4 API Gateway

When a server-side application contains many resource servers or microservices, they are placed behind an API gateway, which serves as the only point of access for all of them. A new spring boot project names api-gateway was created. One essential dependency, named Spring cloud API Gateway, was added to the pom.xml file to let the developed application work as a Spring cloud API Gateway.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Figure 5.4: Spring cloud API gateway dependency injection

The spring boot application uses this dependency to redirect incoming HTTP requests to destination resource servers or microservices. (Minkowski, 2020)

### 5.4.1 Configuring API Gateway Routes

The spring cloud API gateway sends incoming HTTP requests to the destination resource server or microservices running behind the spring cloud API Gateway by configuring the routes. The API gateway in this project runs on the port number 8090. The spring cloud API gateway normally have multiple routes. In this project, the application.properties file was configured as follows to configure routes.

```
spring.application.name=API-GATEWAY

spring.cloud.gateway.routes[0].id = PHOTOS-R-SERVER
spring.cloud.gateway.routes[0].uri = lb://PHOTOS-R-SERVER
spring.cloud.gateway.routes[0].predicates[0] = Path=/api/photos/**
spring.cloud.gateway.routes[0].filters[0] = RemoveRequestHeader=Cookie

spring.cloud.gateway.routes[1].id = ALBUMS-R-SERVER
spring.cloud.gateway.routes[1].uri = lb://ALBUMS-R-SERVER
spring.cloud.gateway.routes[1].predicates[0] = Path=/api/albums/**
spring.cloud.gateway.routes[1].filters[0] = RemoveRequestHeader=Cookie
```

Figure 5.4.1: Routes configuration in the application.properties file

In this project, there are two resource servers, albums and photos resource server. Therefore two routes were configured in the API gateway application.properties file. The spring.cloud.gateway.routes is the starting configuration property for spring cloud API gateway routes. Because the spring cloud API gateway can have several routes, an index of the route was provided in square brackets, in this scenario zero and one. As a result, it is similar to an array of objects, with zero as the first index. The first configuration property is route id, the value of this property is actually the name of the resource server or microservices. In this case they are, PHOTOS-R-SERVER and ALBUMS-R-SERVER. Routes-uri is the next route configuration property. The uri under which the resource server is registered with a discovery service in this circumstance. The value for photos-resource-server is lb:/PHOTOS-R-SERVER and for albums-resource-server is lb:/ALBUMS-R-SERVER. lb stands for load balanced. Predicates are the next route configuration property. Because there can be more than one predicate, the index of the predicates was provided in square brackets. These are Java 8 predicates used to match HTTP requests using headers or methods before routing them to a target uri. It's similar to using a conditional if statement. If the condition is met, the request is forwarded to the destination uri. The value of this property for the albums-resource-server is Path=/api/albums/** and for the photos-resource-server is Path=/api/photos/**. When an HTTP request is routed through API Gateway, a filter is applied. Cookies were filtered from the request header in this case.

## 5.5 Eureka Discovery Service

The technique of one microservice automatically discovering the IP address (Internet Protocol address) of another microservice to connect with it is known as service discovery. If the network location or running the port number of service updates for whatever reason, updated values will be automatically updated in the service registry. When two resource servers in this project desire to connect to each other, the Eureka server provides the destination network location and as well as the port number. To implement the eureka discovery service a spring boot project named discovery-service was created the below mentioned spring dependency was injected in the pom.xml file.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Figure 5.5.a: Eureka server dependency injection

In the application.properties file of the created project was configured as follows-

```
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Figure 5.5.b: Eureka server dependency injection

As illustrated in the above Figure, the port number of the project was changed to 8761 to avoid conflicts. The eureka.client.register-with-eureka was set to false so that eureka discovery service cannot register itself as a discovery client, and eureka.client.fetch-registry was set to false so that it does not look for any other peer registries. Finally, the @EnableEurekaServer annotation was added to the application's root class to make it work as a eureka server. (Chapman, 2014)

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(DiscoveryServiceApplication.class, args);
    }
}
```

Figure 5.5.b: Enabling eureka server

## 5.6 Configure Eureka Client

In this thesis project the two resource servers, albums-resource-server, photos-resource-server, API gateway and spring-cloud-config-server were configured as eureka client. To make them eureka client the first thing was done is that was injected the spring-cloud-starter-netflix-eureka-client dependency in their pom.xml file.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Figure 5.6: Netflix eureka client dependency injection

The spring-cloud-starter-netflix-eureka-client dependency register these project as eureka client to the discovery service. All the eureka client application was given a unique name by configuring the property spring.application.name. For instance, the application name of the album resource server was configured as follows-

```
spring.application.name=ALBUMS-R-SERVER
```

Figure 5.6.a: Configure application name

The specified name for this attribute is registered in the Eureka server by the Eureka client service. The service's IP address will be associated with its name. Other services will use this information to get the network address through the use of the service registry. There are also some other properties what configured to work the services as eureka client. (Configuring Eureka Server, 2018)

```
server.port=0
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
eureka.instance.hostname=localhost
eureka.instance.instance-id=${spring.application.name}:${in-
stanceId:${random.value}}
```

Figure 5.6.b: Application.properties file configuration

Here the server.port was set to zero so that the service can run on random port which is essential for load balancing. When a service runs its multiple instances, the eureka server need eureka.instance.hostname and eureka.instance.instance-id properties configuration to provide the network location correctly. The eureka.client.register-with-eureka and the eureka.client.fetch-registry was set to true so the service can register itself with the eureka server and while starting up, the eureka server finds the peer registry. The address of the Eureka server is indicated by the eureka.client.service-url.defaultZone property. To reach the Eureka application server, the client microservices utilize the following URL-

http://localhost:8761/eureka

Finally, the @EnableDiscovery annotation was added to the application's root class to enable it as a eureka client.

```
@SpringBootApplication
@EnableDiscoveryClient
public class AlbumsResourceServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(
            AlbumsResourceServerApplication.class, args);
    }

    @Bean
    @LoadBalanced
    public RestTemplate getRestTemplate(){
        return new RestTemplate();
    }
}
```

Figure 5.6.c: Enable discovery client and load balancer

@LoadBalanced annotation was added to make sure that when needed it will balance the load. Now the eureka server and all the eureka client project was started. Now navigating to the following URL

http://localhost:8761

the eureka server dashboard was accessible and all the running eureka clients list can be seen.

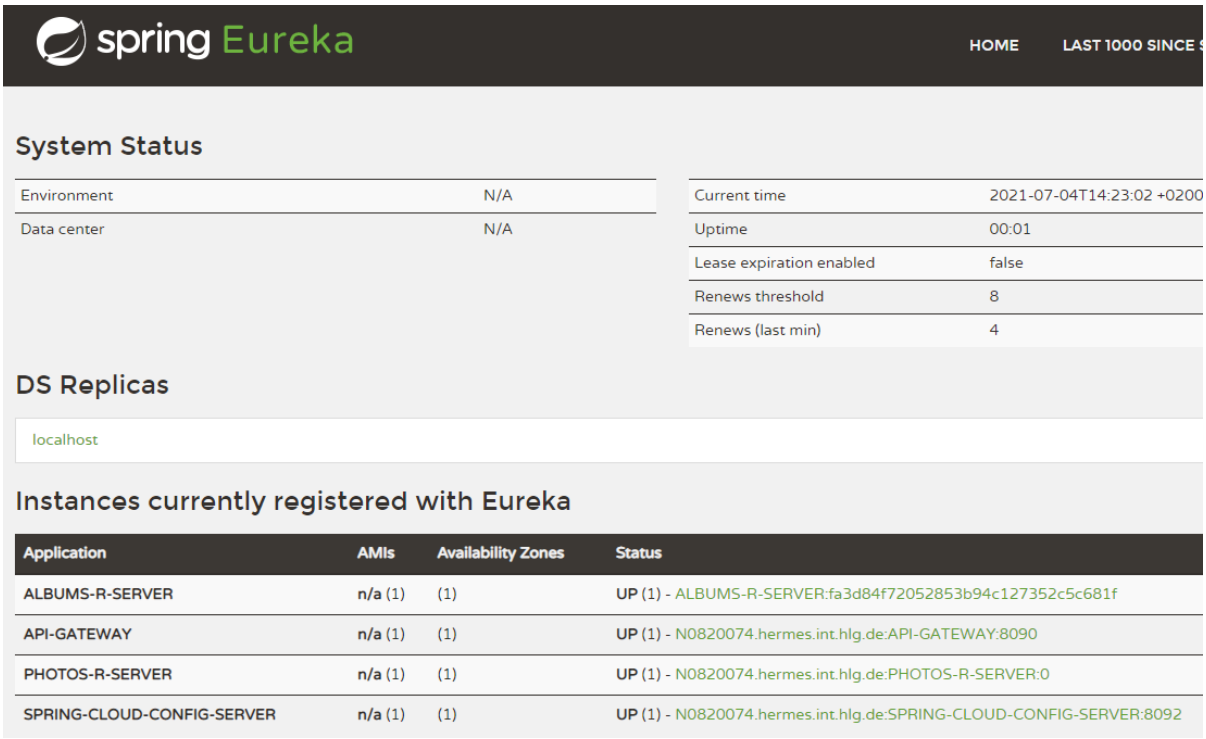


Figure 5.6.d: Screenshot of Eureka server dashboard

## 5.7 Spring Cloud Config Server

A spring cloud config server was implemented to centralize the common configuration properties of the microservices in this project. In the real-world application, config server is very helpful when in a project there are number of microservices who have common configuration properties. Any change in the configuration of these microservices can be done from the single point which spring cloud config server. A spring boot application named spring-cloud-config-server was created. To configure it a spring cloud config server the following dependency injection was mandatory-

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Figure 5.7.a: Spring cloud config server dependency injection

After injecting the config-server dependency, the spring cloud config server was specified by adding the @EnableConfigServer annotation on the spring-cloud-config-server project's root class as follows-

```
@SpringBootApplication
@EnableEurekaClient
@EnableConfigServer
public class SpringCloudConfigServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(
            SpringCloudConfigServerApplication.class, args);
    }
}
```

Figure 5.7.b: Figure 5.6.c: Enable spring cloud config server

The configuration data is kept in a private git repository by the spring cloud config server in this case. As a result, on Github, a spring-cloud-config-server git repository was created. The spring cloud config server was set up to connect to a remote git repository and retrieve application properties, which is subsequently delivered to the resource servers.

```
spring.cloud.config.server.git.uri=\
https://github.com/akas25n/spring-cloud-config-server
spring.cloud.config.server.git.username=
spring.cloud.config.server.git.password=
spring.cloud.config.server.git.uri.clone-on-start=true
```

Figure 5.7.c: Configuration of the application.properties file of the config server

The very first configuration property is the git uri and the value of the git uri is link of the Github repository which was created to store the resource server's common configuration properties. The property is clone-on-start which was set to true to tell the spring cloud config server to clone the Github repository on the project start up. The Github username and password also need to be configured to connect the config server to the remote repository. In the git repository a file named application.properties was created and the following properties configuration were stored into the file since they are common on all the microservices in this project.

```
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
eureka.instance.hostname=localhost
eureka.instance.instance-id=
${spring.application.name}:${instanceId:${random.value}}
```

Figure 5.7.d: Configuration properties on git repository

To make the resource servers work as client of the spring cloud config server, few settings were done. First on the resource servers pom.xml file the following spring dependency were injected-

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

Figure 5.7.e: Injecting the spring cloud config dependency

After that an additional properties file named bootstrap.properties were created in resource server's resources directory. The bootstrap.application file will be loaded before the application.properties file. The make the resource servers be able to connect with spring cloud config server, in the newly created property file two properties were configured as follows-

```
spring.cloud.config.enabled=true
spring.cloud.config.uri=http://localhost:8092
```

Figure 5.7.f: Configuration properties in bootstrap.properties file

Here the config server enabled property were set to true and second uri property were configured with uri of the spring cloud config server. Since the spring-cloud-config-server projects runs on the port number 8092, the uri were set as http://localhost:8092. Finally, the following property were configured in the every microservices of the project which are client of the spring cloud config server. (Piomin, 2020)

```
spring.config.import=optional:configserver:"http://localhost:8092"
```

Figure 5.7.f: Property configuration in resource server's application.properties file

## 5.8 User Storage Provider

In a real-world application, the authorization server must authenticate users whose credentials are stored in an external database; to implement this, the user storage service provider API was used in this prototype project. The User storage provider API is a specific interface that connects external user storage to Keycloak's internal user object model for logging in and managing users. A spring boot project named user-storage-provider was created to allow the Keycloak server to authenticate users by reading user details from a remote database. The user storage provider API was implemented in this project. To

make the project work as user storage API, the following dependencies were injected to the pom.xml file. (Securing Applications and Services Guide, 2019)

```
<dependency>
    <groupId>org.keycloak</groupId>
    <artifactId>keycloak-server-spi</artifactId>
    <version>13.0.0</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.keycloak</groupId>
    <artifactId>keycloak-core</artifactId>
    <version>13.0.0</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.keycloak</groupId>
    <artifactId>keycloak-services</artifactId>
    <version>13.0.0</version>
    <scope>provided</scope>
</dependency>
```

Figure 5.8.a: Dependency injection

To make the user storage provider API works few very important Java class were created. One of them is MyUserStorageProvider.

```
public class MyUserStorageProvider implements
    org.keycloak.storage.UserStorageProvider,
    UserLookupProvider, CredentialInputValidator {

    private KeycloakSession keycloakSession;
    private ComponentModel componentModel;
    private UserApiService userApiService;

    public MyUserStorageProvider(KeycloakSession keycloakSession, Compo-
nentModel componentModel, UserApiService userApiService) {
        this.keycloakSession = keycloakSession;
        this.componentModel = componentModel;
        this.userApiService = userApiService;
    }
```

Figure 5.8.b: Code snippet for the user storage provider class

The MyUserStorageProvider class implements UserStorageProvider, UserLookupProvider and CredentialInputValidator classes. Few important methods were implemented in this class. Among them the getUserByUsername() were implemented to fetch the user by username. The isValid() method were implemented to get verify password response status.

The another very important java class in the user storage provider API were created is MyUserStorageProviderFactory.

```
public class MyUserStorageProviderFactory implements
    UserStorageProviderFactory<MyUserStorageProvider> {

    private static final String PROVIDER_NAME = "user-storage-provider";

    @Override public MyUserStorageProvider create(
        KeycloakSession keycloakSession,
        ComponentModel componentModel) {
        return new MyUserStorageProvider(
            keycloakSession, componentModel,
            buildClient("http://localhost:8091"));
    }
```

Figure 5.8.c: Code snippet for the user storage provider factory class

The storage provider factory class implements the java UserStorageProviderFactory class. The PROVIDER_NAME, in this case "user-storage-provider" is name which were used while deploying the user storage provider API projects to the Keycloak server. After implementing all the other classes in user-storage-provider project, it was packaged deployed on the Keycloak server. So complete this steps, a file names org.keycloak.storage.UserStorageProviderfactory was created and placed into the META-INF.services directory. Inside of this file, the complete name of the MyUserStorageProviderFactory class. Finally, the project was packaged and was deployed to the Keycloak server.

## 5.9 User Service

A separate spring boot project named user-service was developed to demonstrate how to make the user storage provider API accessible to authenticated users by creating HTTP queries to an external micro-service. It is a very simple spring boot project which is connected to a remote database to store the user credentials. To make the project RESTful and for the data persistence the following dependencies were injected-

```
<dependency>
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

Figure 5.9.a: Dependency injection

To store the user credentials in the database, User entity were created with the following instances-

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private long id;
private String userId;
private String userFirstName;
private String userLastName;
private String userEmail;
private String encryptedPassword;
private boolean emailVerificationStatus;
```

Figure 5.9.b: User entity in the user-service project

Here the id is the primary key of the table. The user password was stored in the database as encrypted field to ensure the security. A RestController named UserController was implemented. The UserController has two endpoints.

| HTTP method | Endpoints | Description |
|---|---|---|
| GET | /users/{username}/ | Retrieves user by username |
| POST | /users/{username}/password-verify | Verify the inserted password with the stored password |

Figure: 5.9.c: Endpoints list in the UserController

## 5.10 Client Application

Client application is one of the four parties of OAuth2 authorization process. A mobile app, a web application, or even a desktop program can be used as the client application. A user or a resource owner uses a client application to gain access to data that he owns. In this project, the client application was

developed using Angular. The project was created by using Angular CLI (command-line-interface) tools by running the ng new client-application command. After that as per need of the project few other components, services, models were created. The entire projects structure now looks like –

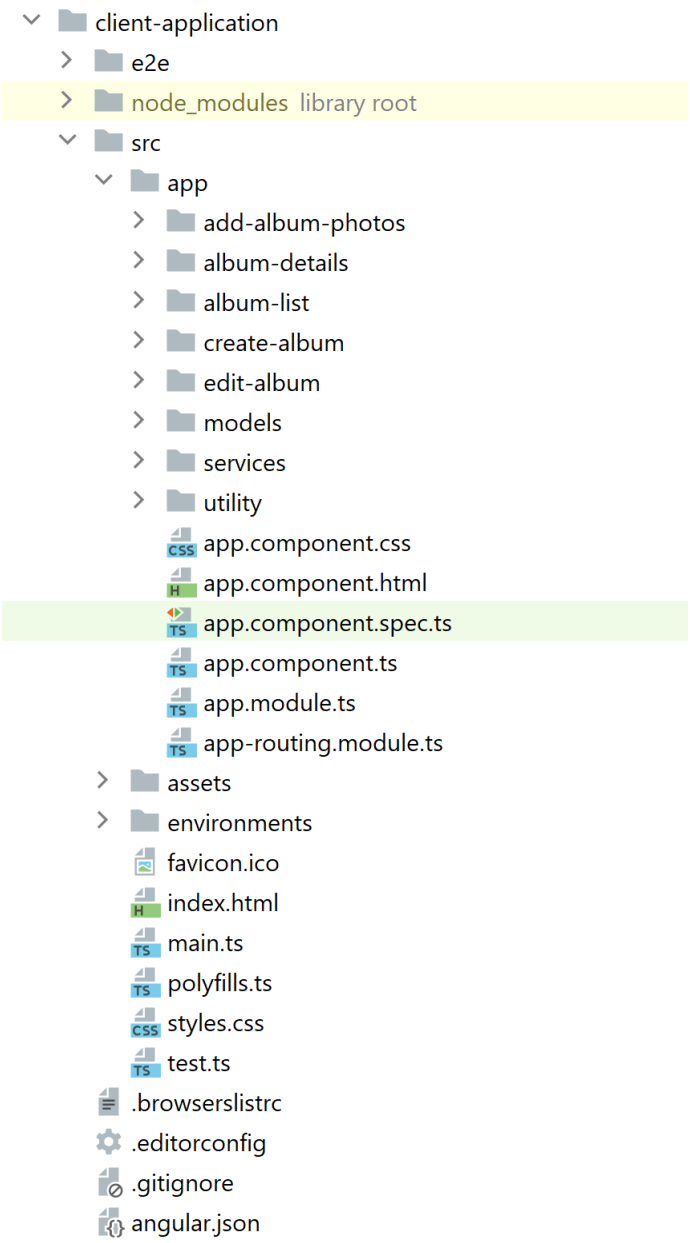


Figure 5.10.a: Client application project structure.

Here the app is root project directory which contains the app.components.ts file which is typescript class of the app component. A component class must be annotated with @Component decorator as follows-

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

Figure 5.10.b: Code snippet of app.component.ts

In the app.component.ts few very important functions were implemented. Among them the getUserName(), onLogout() and getRole() functions are shown below-

```
getUserName(): void{
  this.keycloakService.loadUserProfile().then(profile =>{
    this.userName = profile.firstName + ' ' + profile.lastName;
  });
}

onLogout(): void{
  this.keycloakService.logout();
  this.goHome();
}
getRole(): void{
  if (this.keycloakService.isUserInRole('admin', 'photoalbum')){
    this.userRole = "Admin";
    this.isAdmin = true;
  }
  else if (this.keycloakService.isUserInRole('superadmin', 'photoal-
bum')){
    this.userRole = "Super Admin";
  }
  else{
    this.userRole = "Guest";
    this.isGuest = true;
  }
}
```

Figure 5.10.c: Code snippets of the few functions in app.component.ts

Here he getUserName() function were developed to fetch the current logged in user name from access token. The onLogout() function is responsible to end the current authentication session the logged in user press on the logout button. Finally, getRole() function fetches the current logged in suer role from the access token. The other components in the project are album-list, album-details, create-album, edit-album and add-album-photos component. The album-list component is responsible get the album list from the album-service and then display the album list.

```
public getAlbums(): void{
  this.albumService.getAllAlbums().subscribe(
    (data: Album[]) =>{
      this.albums = data;
```

```
    },
    (error: HttpErrorResponse) =>{
      alert(error.message);
    }
  )
}
```

Figure 5.10.d: Code snippets of getAlbums() function

Here the getAlbums() function receives an Observable from the album service and subscribe the response afterwards to make visible the album object in the html file. The album-details component was created to show the details of induvial album by album id.

```
this.id = this.route.snapshot.params['id'];
this.albumService.getAlbumWithPhotos(this.id).subscribe(data =>{
  this.albumResponse = data;
}
```

Figure 5.10.c: Code snippets of album-details component

The create-album component was developed to create a new album.

```
saveAlbum(){
  this.albumService.createAlbum(this.album).subscribe(data=>
    {console.log(data);
  },
    error=> console.log(error));
}
```

Figure 5.10.e: Code snippets of create-album component

saveAlbum() function saves the newly created album.

The client-application project has two services which are album-service and photo-service. To make a class works as a service the @Injectable decorator must be annotated. The entire album-service are shown below-

```
@Injectable({
  providedIn: 'root'
})
export class AlbumService {
  private baseUrl = "http://localhost:8090/api/albums"
```

```
  constructor( private httpClient: HttpClient) { }

  public getAllAlbums(): Observable<Album[]>{
    return this.httpClient.get<Album[]>(`${this.baseUrl}`);
  }

  public createAlbum(album: Album): Observable<any>{
    return this.httpClient.post<Album>(`${this.baseUrl}`, album);
  }

  public editAlbum(id: number, album: Album): Observable<any>{
    return this.httpClient.put(`${this.baseUrl}/${id}`, album);
  }

  public deleteALbum(id: number): Observable<any>{
    return this.httpClient.delete(`${this.baseUrl}/${id}`);
  }

  public getSingleAlbum(id: number): Observable<any>{
    return this.httpClient.get<Album>(`${this.baseUrl}/${id}`);
  }

  public getAlbumWithPhotos(id: number): Observable<any>{
    return this.httpClient.get<AlbumResponse>(`${this.baseUrl}/${id}`
+"/photos");
  }
```

Figure 5.10.f: Code snippet of album-service class

The @Injectable decorator supply the metadata which Angular needs to inject this as a dependency into just a component. The baseUrl is the root URL of the albums-resource-server. All the functions in the service class return an Observable which afterwards the components subscribe to it to visualize the object dada. The root AppModule imports the AppRoutingModule, which is a highest-level module specialized to routing. In this project, the following routes were configured-

```
const routes: Routes = [
  {path:'home', component: AlbumListComponent, canActivate: [AuthGuard]},
  {path:'', redirectTo: 'home', pathMatch: 'full'},
  {path:'create-album', component: CreateAlbumComponent},
  {path:'album/details/:id', component: AlbumDetailsComponent},
  {path:'album/edit/:id', component: EditAlbumComponent},
  {path:'album/add-photos/:id', component: AddAlbumPhotosComponent}
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
```

Figure 5.10.g: Routes configuration

The path and component are two characteristics of a typical Angular Route: path: a text that corresponds to the URL within address bar of the internet browser. component: When browsing to this route, the

router will construct this component. @NgModule decorator initializes the router here. (Wohlgethan, 2018)

Finally, to initialize the authorization server with the client-application the following code snippets were implemented-

```
export function initializeKeycloak(keycloak: KeycloakService): () =>
Promise<boolean> {
  return () =>
    keycloak.init({
      config: {
        url: 'http://localhost:8080/auth',
        realm: 'photoalbum',
        clientId: 'photo-album-client',
      },
      initOptions: {
        checkLoginIframe : true,
        checkLoginIframeInterval : 25
      },
      loadUserProfileAtStartUp: true
    });
}
```

Figure 5.10.h: Code snippet to initialize Keycloak

The project homepage looks like-

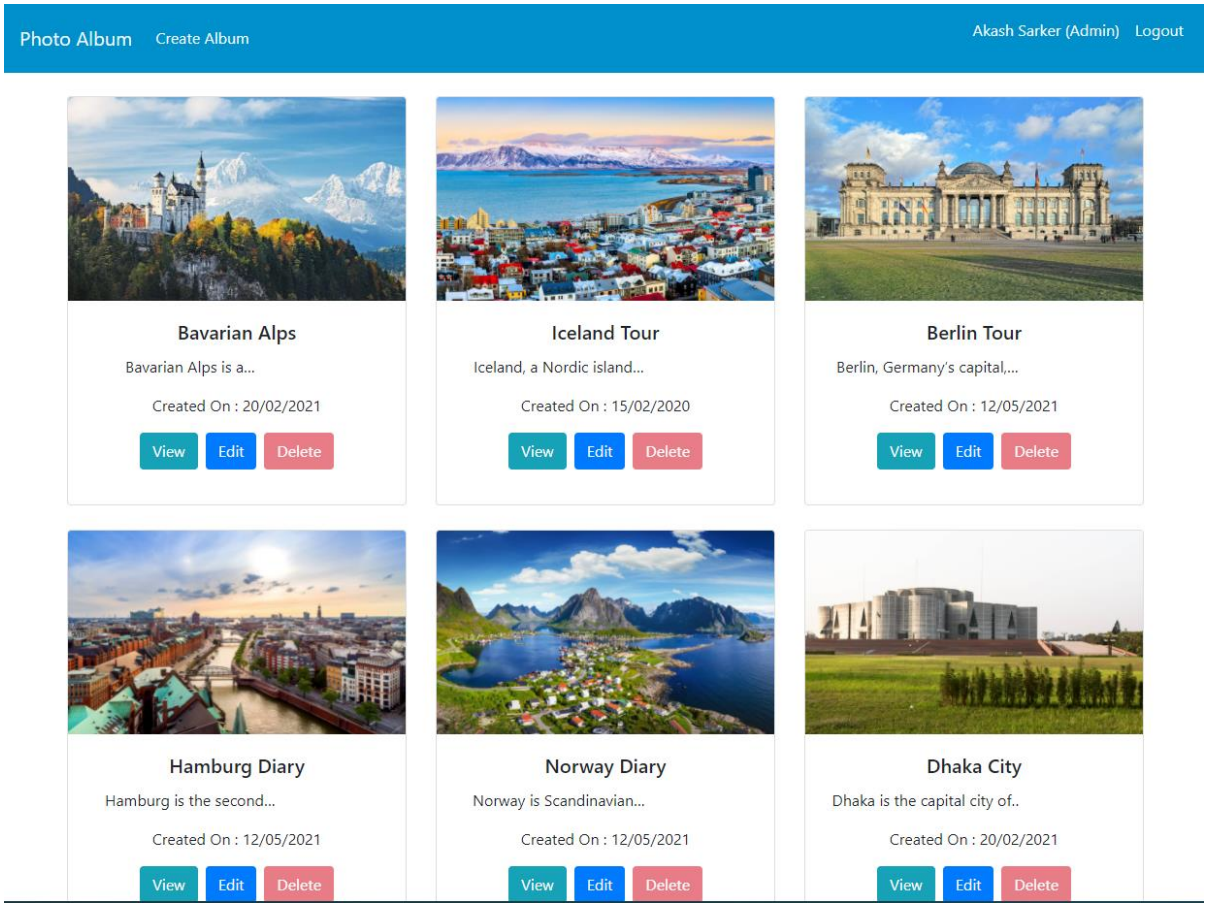


Figure 5.10.i: Screenshot of the project homepage

# 6 System Test

The project which was developed in this thesis a prototype project. This project will not be deployed for real-time use. Only the concept of this project will be implemented to existing microservices. Therefore, no unit or integration tests were implemented in this project. This chapter investigates whether the project's requirements could be met.

## 6.1 Functional Requirements

The functional requirements explain the application's functionality and its behaviour. It is the basic minimal need for the application.

### 6.1.1 User Authentication and Login

User authentication is one of the core functionalities of the developed system. Before getting access to the application every user must authenticate by the authorization server. When the user tries to visit the application base URL, the user is redirected to the Keycloak login page immediately to get authenticated first. In login page, the user must put the correct login credentials, otherwise the user gets error messages.

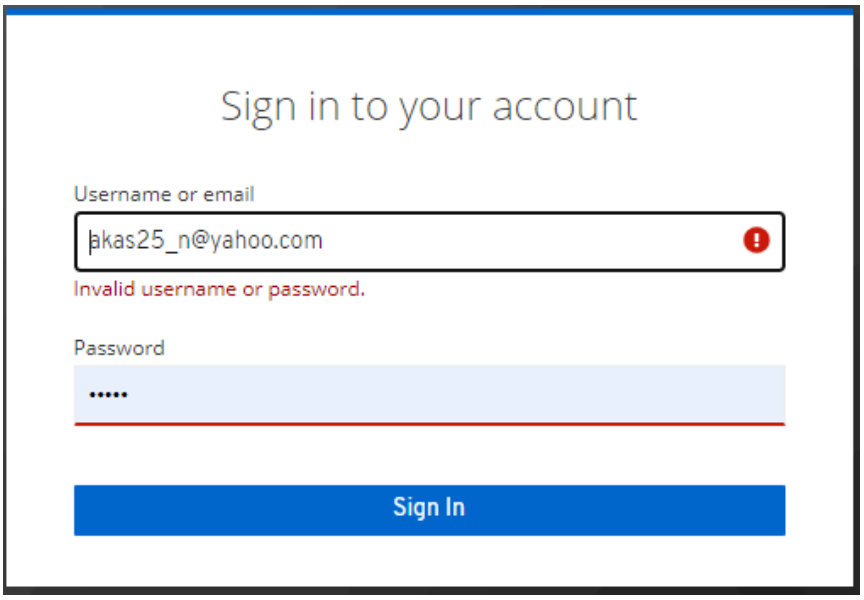


Figure 6.1.1.a: Screenshot of Keycloak login page

When user put right user credentials, the authorization server checks the user credentials behind the scene according to the realm configuration. If everything goes well, the authorization server authenticates the user and issues an access token. With this access token, the user be able to avail the systems resources. For the demonstration purpose, an access token and its payload are shown below-

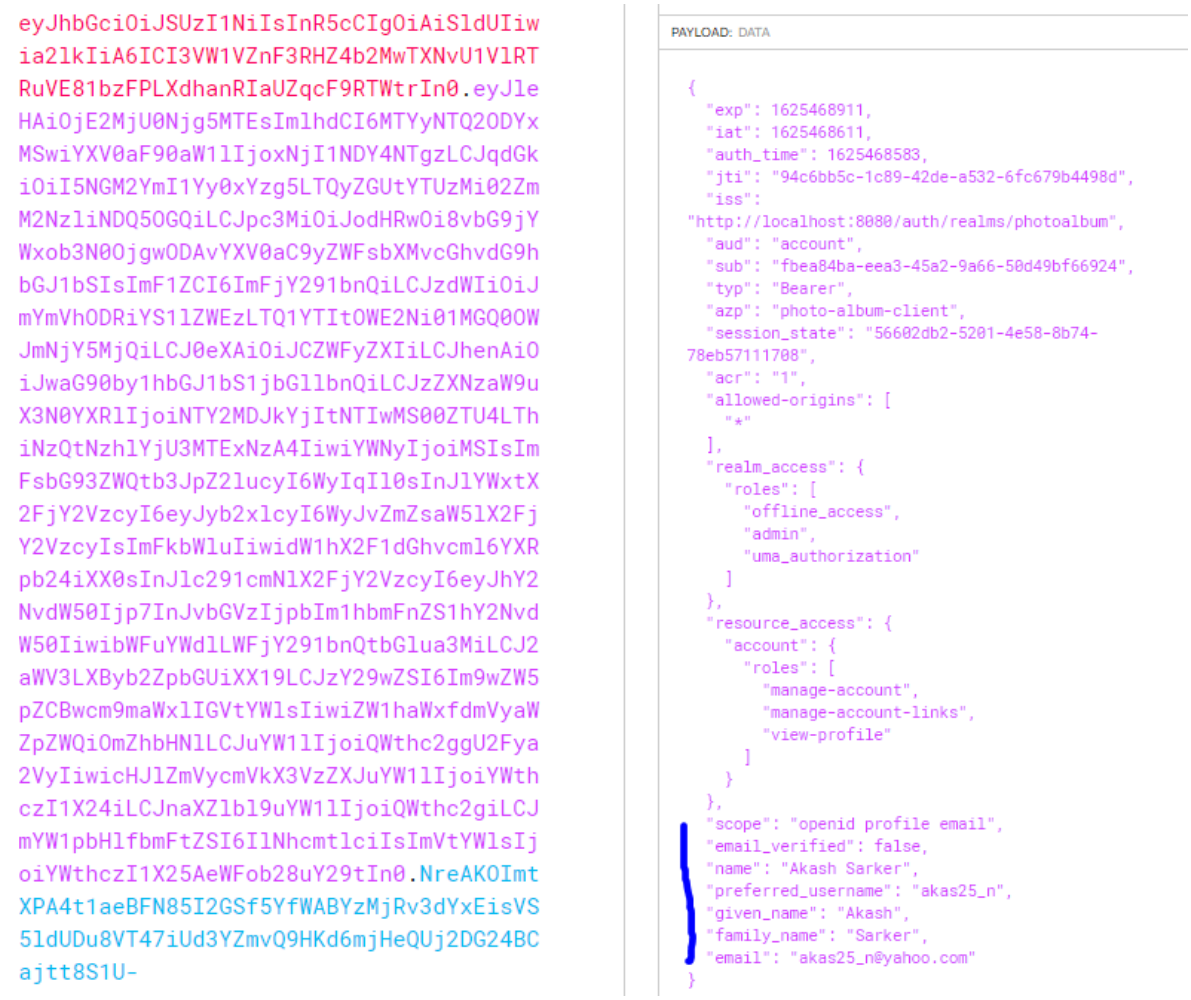


Figure 6.1.1.b: Example of access token with its payload

Here on the left side is the generated access token and on the right side is the payload of the access token. In the bottom of the payload section, it can be seen that the access token contains the currently authenticated user credentials. Once the user is authenticated, the user can access the protected resources of the system according to authorization user have.

### 6.1.2 User Authorization

User authorization is another key functionality of the developed system. For this project, three different roles were created which are guest, admin and super admin. From the Figure 6.1.1.b the payload section of the access token contains a realm access. Inside the realm access there is block names roles. It can be seen that the admin role is in the roles block. Which means the currently logged in user has admin authorities. As a result, the user is allowed to view, edit, create new albums in the system. The admin user does not have the authority to delete an album from the album list. Now if the admin user tries to delete an album, the user gets 403-Forbidden error message.

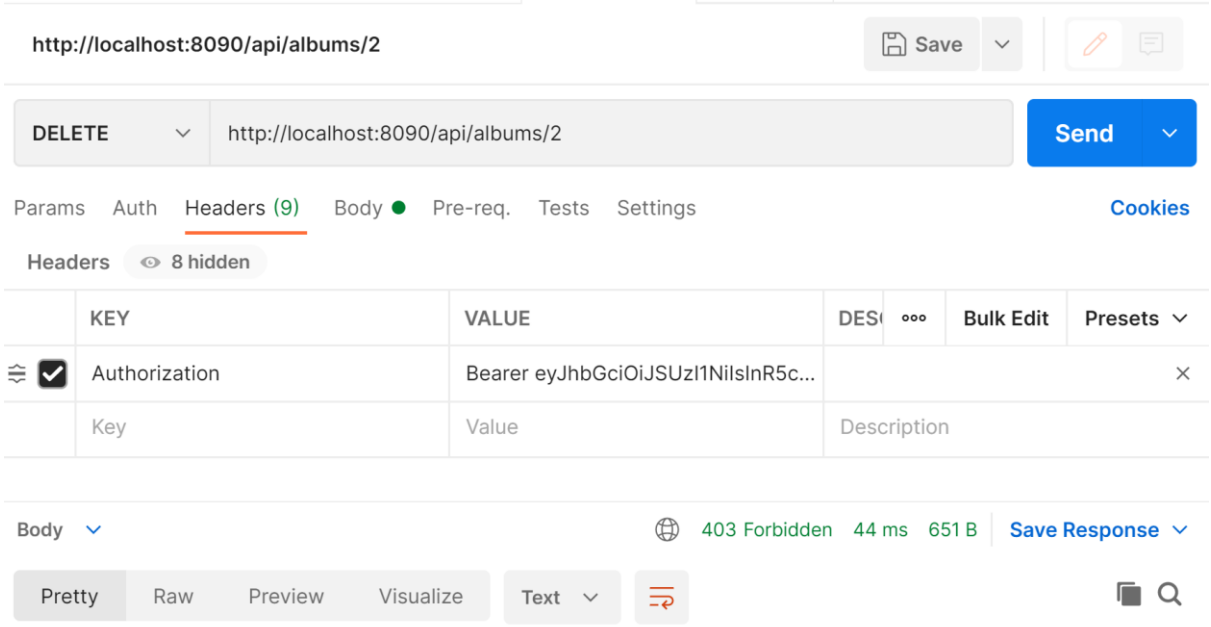


Figure 6.1.2: Screen of postman, 403 forbidden

### 6.1.3 User Can Logout

The authorization server also handles the request to sign out. All microservices that request the user's status are told that the currently logged-in user is no longer active when the user logs out using the Auth-logout Server's function. When the user logs in into the system, on the top right corner of the homepage, the user finds the logout button. When the user presses the logout button, the authorization server ends the current session.

### 6.1.4 Data Manipulation

The Prototype application fulfilled this requirement. It has an option that allows you to create new albums and store them in the database.

The system can also modify the data after it has been saved if necessary. A user with the admin and super-admin roles, for example, can update an album and save the changes to the database.

The database's data can also be deleted by the authorized user. A user with the super-admin role, for example, has the ability to delete an album from the database.

### 6.1.5 Non-functional Requirements

The system's implementation may be validated based on the functional requirements that were met. The non-functional needs are then evaluated in terms of quality in the following stage.

The system's modifiability is assured by the design decision that was made. It is feasible to make modifications to the system more simply by splitting the different pieces. Individual subsystems can be changed and updated without requiring to "touch" the overall system.

In terms of security, care has indeed been implemented to protect that unauthorized access to the system is not possible. The Authorization Server was created for this reason. This secures accessibility to the program's remaining components. Because OAuth2 and OpenId Connect have been used as security requirements, it's important to remember that the system is only protected to the level that OAuth2 and OpenId Connect permit.

The system's reliability might be demonstrated by examining the use cases. The testing process has been shown that the system can behave correctly at any time. If an issue occurs due to an erroneous request, the module rejects it and sends a reply message to the user.

Splitting the separate components can also help to ensure highly scalability. Isolation allows for the easy replacement of different pieces in the event of failures or problems.

The system has been implemented following the criteria in terms of usability. The user is unaware of the processes that are operating in the background. This conduct promotes the user's ease, quickness, and error avoidance.

# 7 Conclusion

The authentication and authorization process of an application developed utilizing the microservice architecture pattern was implemented in this work. The requirements of the prototype were gathered at the beginning of the project. Both requirements, functional and non-functional, were determined. The requirements ranged from basic to more complicated functionality. The next stage was to create a system design that was appropriate for the requirements. This was done through a thorough analysis of the entire system. The authorization server was initially set up. The resource servers and the other components of the backend system were then implemented, followed by the client application. Finally, the entire system performed as planned.

# Bibliography


Kharenko, A. (2015, 09). *Monolithic vs. Microservices Architecture*. Retrieved 4 2021, from https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59

Len Bass, P. C. (2012). *Software Architecture in Practice.* Pittuburgh: Addison-Wesley Professional, - ISBN 978-0-321-81573-6

Richardson, C. (2019). *Microservices Patterns*. New York: Manning Publications Co. - ISBN 9781617294549

Singh, J. (2018, 06 07). *The What, Why, and How of a Microservices Architecture*. Retrieved 2021 04, from https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9

Walls, J. (2017). *Spring Microservices in Action.* New York: Manning Publications Co. - ISBN 9781617293986

Wolff, E. (2016). *Microservices: Flexible Software Architecture.* Boston: Addison-Wesley Professional. - ISBN 978-0-134-60241-7

Walls, C. (2016). *Spring Boot in Action.* New York: Manning Publications Co. - ISBN 9781617292545

Spilcă, L. (2020, 10). *Spring security in Action.* New York: Manning Publications Co. Retrieved from Manning: https://www.manning.com/books/spring-security-in-action

Senthilvel, G. (2017). *Actors in OAuth*. (O'Reilly Media Inc) Retrieved 05 2021, from https://www.oreilly.com/library/view/enterprise-application-architecture/9781786468888/47ccd111-4d78-4d6b-9eae-12c99e6d3a8d.xhtml

Justin Richer, A. S. (2017). *OAuth 2 in Action*. Retrieved 05 2021, from PDF Drive: https://www.pdfdrive.com/oauth-2-in-action-e167377225.html

Wilken, J. (2018). *Angular in Action.* New York: Manning Publications Co.

*Angular applications: The essentials*. (2018). Retrieved 05 2021, from Angular: https://angular.io/guide/what-is-angular

*Securing Applications and Services Guide*. (2019, 11 07). Retrieved 06 2021, from Keycloak: https://www.keycloak.org/docs/4.8/securing_apps/#_spring_boot_adapter

*Server Installation and Configuration Guide*. (2019, 11 07). Retrieved 06 2021, from Keycloak: https://www.keycloak.org/docs/4.8/server_installation/#_standalone-mode

Chapman, P. (2014, 07 14). *Microservices with Spring*. Retrieved 06 2021, from Spring Blog: https://spring.io/blog/2015/07/14/microservices-with-spring

Jaeckel, A. (2018, 11 5). *Konzeption und Umsetzung einer Portalseite und Integrationsumgebung für die HAWAI Microservices*. Retrieved 05 2021, from https://reposit.haw-hamburg.de/handle/20.500.12738/8494

Wohlgethan, E. (2018, 08 03). *SupportingWeb Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue.js*. Retrieved 06 2021, from https://reposit.haw-hamburg.de/handle/20.500.12738/8417

Minkowski, P. (2020, 10 9). *Spring Cloud Gateway OAuth2 with Keycloak*. Retrieved 06 2021, from Piotr's TechBlog: https://piotrminkowski.com/2020/10/09/spring-cloud-gateway-oauth2-with-keycloak/

Piomin. (2020, 01 22). *Microservices Security with Oauth2*. Retrieved 06 2021, from Github: https://github.com/piomin/sample-spring-oauth2-microservices

*Configuring Eureka Server*. (2018, 02 15). Retrieved 05 2021, from Github: https://github.com/Netflix/eureka/wiki/Configuring-Eureka

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

| _____ | _____ | _____ |
|---|---|---|
| Ort | Datum | Unterschrift im Original |