fyp23059

# COMP4801 Final Report

## Implementation of HKID/HKU OAuth 2.0 Provider for Real Name Verification

Name: Chiew Lik Seng (3035767487)

Supervisor: Dr. Chan, Hubert T. H.

Date of Submission: 26/04/2024

# Acknowledgement

I would like to express my gratitude to the supervisor of this project, Dr. Chan, Hubert T. H.

# Abstract

This report addresses the evolving landscape of online identity verification and proposes a solution using the OAuth 2.0 framework. The increasing demand for real-name verification in various online services, driven by concerns over security and public order, prompts the need for a more efficient and secure authentication system. The report explores the potential integration of OAuth 2.0 as an authentication platform, emphasizing its role in simplifying and unifying real-name verification processes. The research delves into the OAuth 2.0 protocol, examining its principles, grant types, and implementation in web and mobile applications. Security considerations and best practices, including potential weaknesses and countermeasures, are thoroughly investigated. The literature review focuses on the OAuth 2.0 framework, highlighting its deployment as an authorization framework for secure access to resources. The project's aims involve developing an OAuth 2.0 provider for real identity verification and contributing to the fields of authentication, cybersecurity, and delegated access control. Objectives include understanding the OAuth 2.0 protocol, selecting the most suitable grant type, integrating OAuth 2.0 into applications, and addressing security concerns. The report outlines the products and deliverables of the OAuth HKID provider, including the authorization server, resource server, and the authorization server login page. Technologies such as Node.js, Express.js, React.js, and MongoDB are chosen to implement the backend, frontend, and database components. Vercel and Heroku are selected as the cloud platforms for deployment, ensuring scalability and reliability. The project was completed with the three main deliverables, HKID Frontend Login Application, HKID Backend OAuth Server, and a HKLIB client application shipped to production, providing visible results. Magic links are utilised to enhance the security measure of the system. Possible future enhancements could be added to the project to enhance the system performance. Risk management involves identifying potential risks such as resource over-utilization, incorrect implementation of OAuth 2.0, and inappropriate database design. Mitigation strategies include using AWS Elastic Load Balancing, careful implementation following documentation, and consulting MongoDB experts. The report concludes by emphasizing the project's objective to deliver a secure and efficient OAuth provider for real identity verification.

# List of Tables


| Table | Page number |
|---|---|
| Risk Assessment Matrix | 12 |
| Non-OHS Risk Assessment | 12 |


# List of Figures


| Figure | Page number |
|---|---|
| Abstract Protocol Flow | 3 |
| Authorization Code Flow | 5 |
| Screencap of HKID Login Page | 7 |
| Screencap of HKID Authorization Request Page | 8 |
| Code snippet of Authorization Endpoint | 9 |
| Code snippet of Token Endpoint | 10 |
| Code snippet of Login Endpoint | 11 |
| Code snippet of User Info Retrieval Endpoint | 12 |
| Screencap of HKLIB Login Page | 13 |
| Screencap of HKLIB Home Page | 14 |
| Screencap of Magic Link Example | 20 |
| Authorization Code Interception Attack Flow | 22 |
| Abstract Authorization Code Flow with PKCE | 23 |
| Code Snippet of receiveCodeChallenge middleware | 24 |
| Code Snippet of verifyCodeChallenge middleware | 24 |
| Project Gantt Chart | 27 |

## Table of Contents


Acknowledgement ..... i

Abstract ..... i

List of Tables ..... ii

List of Figures ..... ii

1. Introduction ..... 1
2. Aims & Objectives ..... 2
   - 2.1. Research Question ..... 2
   - 2.2. Aims ..... 2
   - 2.3. Objectives ..... 2
3. Literature Review ..... 3
   - 3.1. OAuth 2.0 Framework ..... 3-5
4. Products & Deliverables ..... 6
   - 4.1. HKID Frontend Application ..... 6
     - 4.1.1 HKID Login Page ..... 6
     - 4.1.2 HKID Authorization Grant Request Page ..... 7-8
   - 4.2. HKID OAuth Backend Server ..... 9
     - 4.2.1 Authorization Endpoint ..... 9
     - 4.2.2 Token Endpoint ..... 9-10
     - 4.2.3 Login Endpoint ..... 10-11
     - 4.2.4 User Info Retrieval Endpoint ..... 11-12
   - 4.3. HKLIB Frontend Application ..... 12-14
5. Implementation Details ..... 15
   - 5.1. Frontend Tech Stack ..... 15
     - 5.1.1 React.js ..... 15-16
     - 5.1.2 Tailwind CSS ..... 16-17
   - 5.2. Backend Tech Stack ..... 17
   - 5.3. Database ..... 18
   - 5.4. Application Deployment ..... 18-19
   - 5.5. Magic Link User Authentication Method ..... 19-20
6. Difficulties Encountered ..... 21-25
7. Future Improvements ..... 25
   - 7.1. Passkey Authentication ..... 25
   - 7.2. Rate Limiting ..... 25-26

8. Project Scope & Project Timeline ..... 27
8.1. Project Scope ..... 27
8.2. Project Plan & Timeline ..... 27
9. Risk Management Plan ..... 28
9.1. Risk Assessment Matrix ..... 28
9.2. Non-OHS Risk Assessment ..... 28
10. Conclusion ..... 29
11. References ..... 30

# 1 Introduction

In this era of globalisation and rapid technology advances, Internet is accessible to most of the world's total population [1]. The widespread proliferation of internet access throughout many societies worldwide has facilitated not only socially positive practices, but also detrimental ones. There are countless users on the web that remain their anonymity and participate in online discussions without fear of reprisal in the physical world. It is just the nature of Internet. But in recent years, the norm has gradually been broken. It can be observed that more and more applications or services require real name verification before we can access the service for instance the registration process of student Octopus card, booking process of community sports centre, etc. Besides that, countries like China even imposed the real-name registration rules to stabilize public order [2]. The Republic of Korea, the nation with the highest rate of high-speed internet access in the world, created its own "Real Name Verification System" as well by forcing Korean citizens to verify their identities on a public institution website [3].

In Hong Kong, users are generally required to scan their HKID (Hong Kong Identity Card) by their phone camera in these real identity verification procedures. It is annoying that the process is not essentially accurate in times since factors like lighting, shadow and camera resolution could affect the outcome of the verification process. Other than that, the same process is repeated whenever real name verification is required on any other web services. Thus, some users just quit attempting to use certain services.

To better facilitate the real identity verification process, OAuth framework has come across my mind as an auth account basically represents a user's identity in the scope of client application and HKID is no different from an auth account in real life. Each citizen would have his/her own user account on a resource server regulated by a trusted authority which would be the government and all the personal information regarding the account owner is contained within the resource server. Whenever users log into any third part web applications via OAuth 2.0 protocol, by providing the correct login credentials to the government server, an acknowledgement would be returned by the server and it could serve the same purpose as the HKID and extend to even more possibilities. The real-name verification processes could also be simplified and unified by one authentication service platform.

# 2 Aims & Objectives

In this section, the questions that are to be kept in mind during the research process are covered in subsection 2.1 Research Question. As for the aims and objectives of project, they are being further discussed in subsection 2.2 Aims and subsection 2.3 Objectives accordingly.

## 2.1 Research Question

The research questions for this project are as follows:

1. What is OAuth 2.0 protocol and how does it work in principle?
2. What are the differences between different OAuth 2.0 grant type and which one would be the best choice for our situation?
3. How can OAuth 2.0 protocol be incorporated into web and mobile applications to act as the authentication system?
4. What are the common security weaknesses in implementation of OAuth 2.0 server and what are the countermeasures?

## 2.2 Aims

This project aims to develop an OAuth 2.0 provider for real identity verification and investigate the possible extension and contribute to the advancement of knowledge in the field of authentication, cybersecurity, and delegated access control.

## 2.3 Objectives

The objectives of this project are:

1. To understand the OAuth 2.0 protocol and its underlying principles.
2. To study different OAuth 2.0 grant type (e.g. Authorization Code Flow, Implicit Flow, Resource Owner Password Credentials Flow, and Client Credentials Flow) and pick the most suitable one for the project.
3. To explore the integration of OAuth 2.0 with existing web and mobile applications by incorporating SSO capabilities and delegated access.
4. To inspect the security considerations and best practices in OAuth 2.0, for instance secure token handling, client authentication and protection against common attacks.
5. To deliver a developer-friendly authentication tool.

# 3 Literature Review

As the authentication system is built on top of a mature, secure authorization framework, this project relies heavily on analysis and implementation of the OAuth 2.0 Framework. Thus, a brief introduction of the OAuth protocol is discussed in subsection 3.1 OAuth 2.0 Framework.

## 3.1 OAuth 2.0 Framework

Open Authorization (OAuth) is a widely deployed authorization framework that allows users (known as "resource owner") to grant limited access to their resources on one website (known as the "resource server") to another website or application (known as the "client") without sharing their credentials [4]. In this way, users do not need to create an auth account for every single new client application. The risk of leaking users' login credentials is also minimized as OAuth relies on short-lived access tokens and third-party client application will only be communicating with the resource server via an access token. In general, there will be not less than 2 endpoints in the authorization server. The two mandatory endpoints are authorization endpoint and token endpoint. The authorization endpoint communicates with the resource owner and returns an authorization grant/code whereas the token endpoint interacts with the client application where client presents the authorization grant to exchange for the access token. The scope of resource access is embedded onto the access token which means third-party client application has restricted access to the resource data of the resource owner.

The basic OAuth protocol flow is illustrated in Fig.1 below.

```
+--------+                               +---------------+
|        |--(A)- Authorization Request ->|   Resource    |
|        |                               |     Owner     |
|        |<-(B)-- Authorization Grant ---|               |
|        |                               +---------------+
|        |
|        |                               +---------------+
|        |--(C)-- Authorization Grant -->| Authorization |
| Client |                               |     Server    |
|        |<-(D)----- Access Token -------|               |
|        |                               +---------------+
|        |
|        |                               +---------------+
|        |--(E)----- Access Token ------>|    Resource   |
|        |                               |     Server    |
|        |<-(F)--- Protected Resource ---|               |
+--------+                               +---------------+
```

Fig. 1: Abstract Protocol Flow. [4]

OAuth 2.0 defines four main authorization grant types: authorization code, implicit, resource owner password credentials, and client credentials. As for now, authorization code flow would be our choice of authorization grant as it provides an additional layer of security compared to other grant types. It involves an extra step of

exchanging authorization code between the client and the authorization server which better enhance the security of the process [4]. Besides that, the user is redirected to the authorization server's login page as they are providing their login credentials. This makes sure that users' login credentials are never exposed to the client application and in turn reassure users and earn their trusts to select the OAuth login method.

The Authorization Code Flow (see Fig. 2 below) below includes the following steps:

(A) The flow starts with the client directing the resource owner' user-agent (in this case the browser) to the authorization endpoint in authorization server. The client identifier, requested scope and a redirection URI is sent along.

(B) The resource owner is authenticated by the authorization server via the browser and resource owner is asked for its permission to grant the client's access request.

(C) Once access is granted by the owner, the user-agent is redirected back to client following the redirection URI mentioned earlier and the authorization code is included in the URI.

(D) The client then makes a request for the access token by presenting the appropriate authorization code and the redirection URI to the authorization server.

(E) The authorization server authenticates the client by validating the authorization code and verifying that the redirection URI is the one used to exchange for authorization code in the previous steps. The authorization server returns an access token if the authentication is valid.

```
+----------+
| Resource |
|   Owner  |
|          |
+----------+
     ^
     |
    (B)
+----|-----+          Client Identifier      +---------------+
|         -+----(A)-- & Redirection URI ----->|               |
|  User-   |                                 | Authorization |
|  Agent  -+----(B)-- User authenticates --->|     Server    |
|          |                                 |               |
|         -+----(C)-- Authorization Code ---<|               |
+-|----|---+                                 +---------------+
  |    |                                         ^      v
 (A)  (C)                                         |      |
  |    |                                         |      |
  ^    v                                         |      |
+---------+                                      |      |
|         |>---(D)-- Authorization Code ---------'      |
|  Client |          & Redirection URI                 |
|         |                                            |
|         |<---(E)----- Access Token -------------------'
+---------+       (w/ Optional Refresh Token)

Note: The lines illustrating steps (A), (B), and (C) are broken into
two parts as they pass through the user-agent.
```

Fig. 2. Authorization Code Flow. [4]

The communication route to the authorization server is built upon Transport Layer Security (TLS) hence it greatly enhances the security of the framework.

# 4 Products & Deliverables

There are mainly three products that we delivered for this project including a HKID frontend application, an OAuth backend server and a supplementary HKLIB frontend application. Our OAuth provider follows the standard RESTful client-server architecture hence it comprises a frontend application and a backend server. The HKLIB application is a nearly empty application that serves the bare minimum functionalities for demonstration purposes.

## 4.1 HKID Frontend Application

The HKID frontend application is the client-sided application that interacts with the users and communicates with the backend OAuth server when necessary. It consists of two pages including the HKID Login Page and the HKID Authorization Grant Request Page.

### 4.1.1 HKID Login Page

The HKID Login Page serves as an interface for users to input their login credentials and performs the user authentication process. Here we utilized the magic link authentication method as the user authentication method instead of the traditional password approach. In this way, the user authentication process becomes much more resilient to online attacks as magic link authentication eliminates the need for users to create and remember complex passwords, reducing the likelihood of weak passwords, password reuse, and credential theft. Instead of relying on static passwords that can be susceptible to various attacks such as brute force, phishing, and dictionary attacks, magic links generate unique, time-limited URLs that serve as one-time login tokens and they are sent to the user's registered email account. This dynamic authentication mechanism enhances security by reducing the attack surface and mitigating the risk of unauthorized access. A more detailed explanation of magic link authentication is provided in the later section 5.5.

As shown in Fig. 3 below, after user inputting their email into the textbox and clicking on the "Generate a Magic Link" button, the button will be disabled after the button text changed to "Magic Link Sent to your Email!" so that the action of repeated generation of magic links for a single user is prevented. The user is then required to check out their email inbox and click on the magic link included in the email. It will then redirect the user's browser to the HKID Authorization Request Grant Page.

As for user registration process, clicking the blue link below will redirect users to the Hong Kong Immigration Department official website because ideally the user account is equivalent to the digital version of the HKID card so it should be distributed by the Immigration Department to avoid any fraudulent user account.

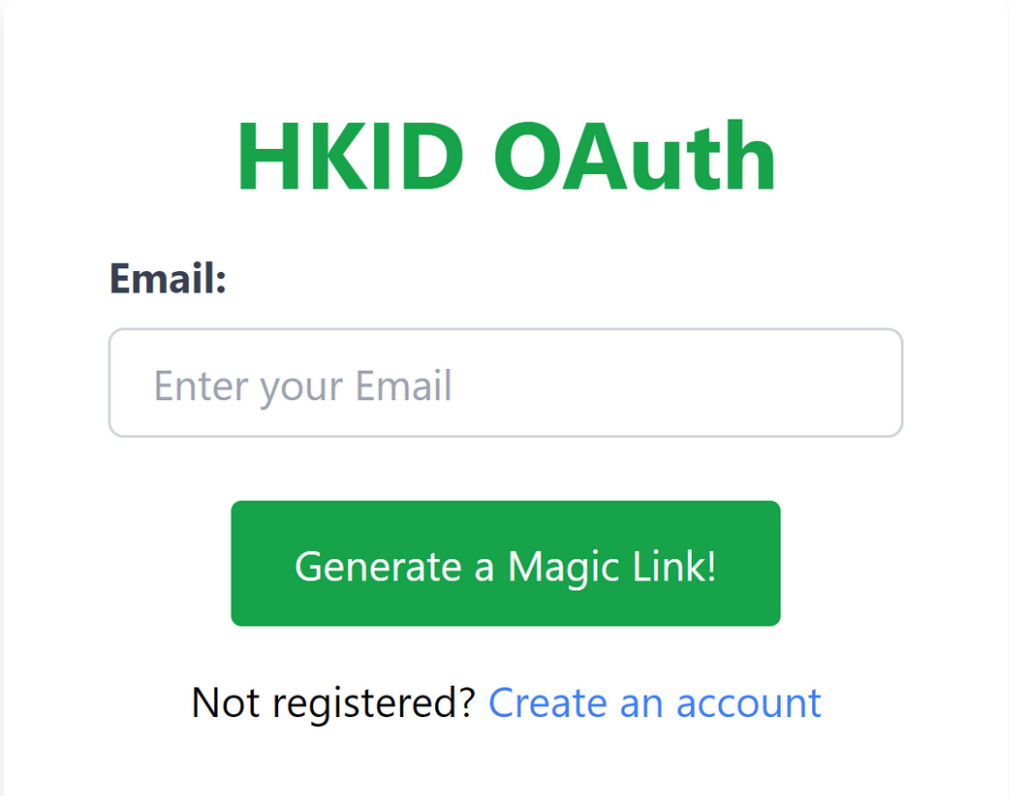


Fig. 3. Screencap of HKID Login Page. [5]

#### 4.1.2 HKID Authorization Request Grant Page

This page informs users about the scope of user info that the third-party client application can access and asks for user's consent. On the top of the webpage, the name and domain of the third-party client application are first clearly displayed to reassure the user that the client application that they attempt to login via our HKID OAuth provider is a registered client under our provider. Next in the subsequent lines, the scope of user info that the client application can and cannot access will be shown clearly with the aids of coloured icons. On the left-hand side of each attribute of user info, a green tick icon or a red cross icon is added whereas the green tick icon represents attribute that is accessible to the client and the red cross icon represent attribute that is non-accessible to the client. At the bottom last paragraph, the redirect URL that the client provided during the client registration process will be shown to inform the users that upon giving consent to this action, the browser will be redirected back to this specific URL of the client application. The privacy policy and terms of service are attached below the redirect URL as well. Lastly there are two buttons with one "Cancel" button on the left and one "Authorize" button on the right. Clicking on the "Cancel" button terminates the authorization process by redirecting user's browser to the login page of HKID and invalidates the completed user authentication process by removing the authentication token cookie

from the browser. On the other hand, clicking on the “Authorize” button entail the user consent for the authorization action and hence the process will be proceeded to the next stage of requesting for authorization code and in turn the access token. The interface of the HKID Authorization Grant Request Page is shown in Fig. 4 below.

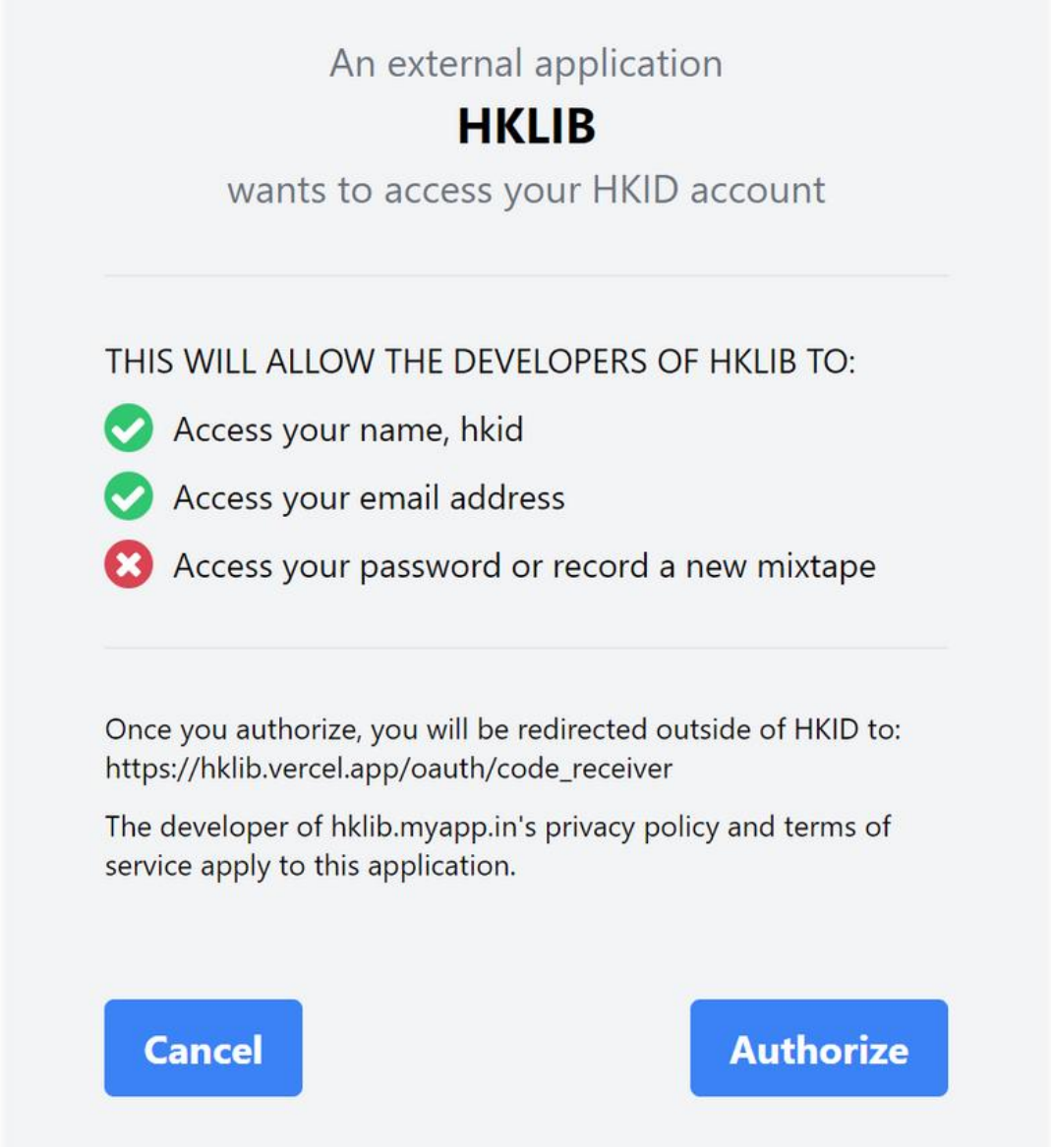


Fig. 4. Screencap of HKID Authorization Request Page. [5]

## 4.2 HKID OAuth Backend Server

It acts as an OAuth Server that provides REST APIs for HKID frontend application to call upon. There are two main API endpoints including the authorization endpoint which returns the auth code and the token endpoint which returns the access token. The authorization endpoint and the token endpoint handle the authorization logic. As for other functionalities such as user authentication, retrieval of user info and user account registration, they are handled by other API endpoints. Since the server mostly consists of API endpoints, so code snippets will be shown as the products in this section.

### 4.2.1 Authorization Endpoint

The Authorization Endpoint is an API route handler that handles the authorization code request. It requires the incoming authorization request to be a HTTP GET request with projectID, redirectURL, scope, code_challenge and code_challenge_method as the request query parameters and the auth_token embedded in the request header. Fig. 5 below shows the code snippet of the authorization endpoint.

The projectID, redirectURL and scope are required to be passed as arguments into the projectMiddleware to verify that this project exists in our database and that the redirectURL and scope match with the corresponding field of the existing document. The term "project" here refers to the client application that is registered under our provider beforehand. Meanwhile, the verifyAuthToken middleware reads the auth_token from the header to verify the user's identity and the receiveCodeChallenge middleware takes in the code_challenge and code_challenge_method as function arguments and handles the implementation of PKCE details which will be discussed in the later section. Upon receiving and validating the input details, the authorization endpoint generates an Authorization Code which is basically a JSON Web Token and returns it to the client by including it in the response body.

```
router
  .route("/code")
  .get(projectMiddleware, verifyAuthToken, receiveCodeChallenge, async function (req, res) {
    try {
      var code = await req.user.generateOAuthCode(req.project);
      redirectURL = `${req.query.redirectURL}?code=${code}`;
      return res.send({ redirectURL });
    } catch (e) {
      console.log(e);
      res.status(500).send({ message: "Unknown Error", code: 500 });
    }
  });
```

Fig. 5. Code snippet of Authorization Endpoint

### 4.2.2 Token Endpoint

The Token Endpoint is an API route handler that handles the access token request. It requires the incoming authorization request to be a HTTP GET request with projectID, redirectURL, scope, projectSecret, code, code_verifier as the request query parameters. Here the projectMiddleware is

explained in the previous subsection of Authorization Endpoint. The verifyAuthToken function is the middleware function that reads the value of “code” query parameter, verifying and decoding the authorization code JSON web token. The project details are also encoded into the authorization code and thus the decoded authorization code can be compared with the project details sent along this request to ensure the details match. Next, the verifyCodeChallenge with the code_verifier value will be discussed in the PKCE section later. For the projectSecret query parameter, it is required because the authorization request and token request are two separate requests. To ensure the sender of the authorization request and the token request is the actual owner from the client application, a client secret named projectSecret is required to provide along the token request as this client secret is supposed to be stored safely and only known by the client application so as to prove the sender’s identity.

The Token Endpoint returns an access token when the valid authorization code is sent along with the correct projectSecret. A screencap of code snippet of the Token Endpoint is provided as Fig.6.

```
35  router
36    .route("/token")
37    .get(projectMiddleware, verifyOAuthCode, verifyCodeChallenge,async function (req, res) {
38      if (req.project.projectSecret != req.query.projectSecret) {
39        return res
40          .status(400)
41          .send({ code: 400, message: "Mismatch ProjectID and Secret" });
42      }
43      user = req.user;
44
45      user
46        .generateAccessToken(req.decoded.scope)
47        .then((token) => {
48          return user.removeToken(req.token).then((e) => {
49            return token;
50          });
51        })
52        .then((token) => {
53          res.send({ access_token: token });
54        })
55        .catch((e) => {
56          res
57            .status(400)
58            .send({ message: "Error while generating access token" });
59        });
60    });
```

Fig. 6. Code snippet of Token Endpoint

### 4.2.3 Login Endpoint

The Login Endpoint is an API route handler that handles the magic link authentication process. The code processes the request body to extract the email, projectID, redirectURL, and scope using Ramda's pick function. It then attempts to find a user with the provided email. If successful, an

authentication token is generated for the user and an email containing a link for user authentication is sent to the user's registered email account using the nodemailer library. Finally, it responds with a success message and status 200 if the email is sent successfully, or an error message and status 400 if any errors occur during the process.

In summary, the main function of this code is to handle user login requests by generating an authentication token for the user and sending an email containing a magic link for user authentication. The code snippet can be seen in Fig. 7 below.

```
31 router.route("/login").post(async function (req, res) {
32   var body = R.pick(["email", "projectID", "redirectURL", "scope"], req.body);
33
34
35   try {
36     var user = await User.findByCredentials(body.email);
37     var token = await user.generateAuthToken();
38
39     const info = await transporter.sendMail({
40       to: body.email,
41       from: 'hkidoauth@gmail.com',
42       subject: 'Magic Link: User Authentication of HKID OAuth',
43       text: `Login to your HKID OAuth account with this link: \n https://hkid-frontend.vercel.app/
         authorization?token=${token}&projectID=${body.projectID}&redirectURL=${body.redirectURL}&
         scope=${body.scope}`
44     });
45
46     return res.status(200).send({"message": "Successfully generated and sent email!"});
47
48
49   } catch (e) {
50     console.log(e);
51     res.status(400).send({ code: 400, message: e });
52   }
53 });
```

Fig. 7. Code snippet of Login Endpoint

### 4.2.4 User Info Retrieval Endpoint

This API endpoint is responsible for processing the user info retrieval requests from the client applications. The high-level workflow of the endpoint is illustrated in Fig. 8 below. A scopeMapping object that maps different access levels (scopes) to an array of fields is first initialized when server application starts running. When the server receives a /userinfo request, it first verifies and validates the access token. If the token is valid, the token is decoded and the scope embedded into the token is then extracted from the token. At last, according to the specific scope that this token is allowed to access, the server returns a response body with the corresponding user info.

```
const scopeMapping = {
  full: ["_id", "name","hkid", "email", "phone"],
  default: ["_id", "name", "hkid"],
  email: ["_id", "name", "email"],
  phone: ["_id", "name", "phone"],
};

router
  .route("/userinfo")
  .get(verifyAccessToken, async function (req, res) {
    token = req.decoded;
    user = req.user;
    res.send(R.pick(scopeMapping[token.scope], user));
  });
```

Fig. 8. Code snippet of User Info Retrieval Endpoint

### 4.3 HKLIB Frontend Application

It is a supplementary frontend application that acts as an example of third-party client application to showcase the interaction between a client application and our HKID OAuth provider. It consists of 3 pages including the HKLIB Login Page, the Code Receiver Page, and the Home Page. The Login Page as shown in Fig. 9 below only consists of static HTML elements without any interaction logic implemented except for the light blue button with the text “Continue With HKID”. This button represents the link to login via our HKID provider and clicking it would redirect the user to the HKID Login Page.

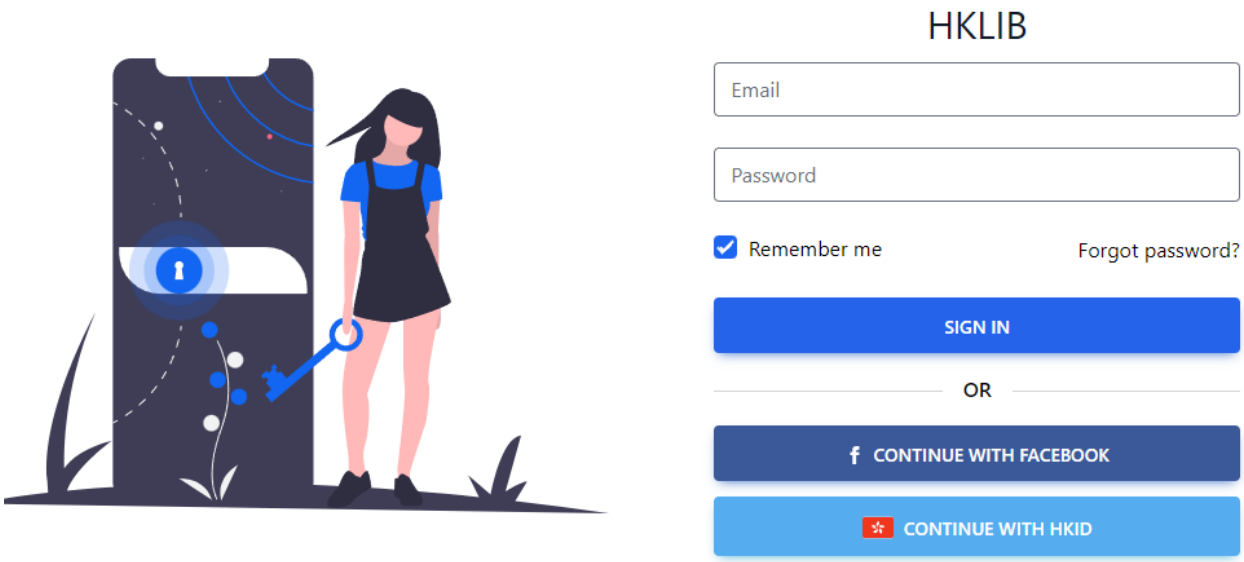


Fig. 9. Screencap of HKLIB Login Page

Next, there is no visible user interface in the Code Receiver Page as no information is displayed on it. The page mainly functions as a route to safely retrieve the generated Authorization Code from the OAuth server and subsequently proceed to exchange the Authorization Code for the Access Token and in turn retrieves the user info by making the respective API calls.

Lastly, there exists the HKLIB Home Page that is protected such that only authenticated and authorized users can access to it. Otherwise, users will be redirected back to the HKLIB Login Page. The Home Page exhibits the name and HKID number retrieved from the OAuth server and the list of books available for users to read. A logout button is located on the top right corner of the webpage and user will be signed out upon pressing the logout button. The interface of the HKLIB Home Page is shown as below in Fig. 10:

| Book | Author | Publication Date |
|---|---|---|
| Book 0 | Author 0 | Publication Date 0 |
| Book 1 | Author 1 | Publication Date 1 |
| Book 2 | Author 2 | Publication Date 2 |
| Book 3 | Author 3 | Publication Date 3 |
| Book 4 | Author 4 | Publication Date 4 |
| Book 5 | Author 5 | Publication Date 5 |
| Book 6 | Author 6 | Publication Date 6 |
| Book 7 | Author 7 | Publication Date 7 |
| Book 8 | Author 8 | Publication Date 8 |
| Book 9 | Author 9 | Publication Date 9 |
| Book 10 | Author 10 | Publication Date 10 |
| Book 11 | Author 11 | Publication Date 11 |
| Book 12 | Author 12 | Publication Date 12 |
| Book 13 | Author 13 | Publication Date 13 |

Fig. 10. Screencap of HKLIB Home Page

As of now, the list of books are merely dummy data as HKLIB application and the list of books are not the emphasis of this specific project. It was built to be barely sufficient enough to demonstrate the capabilties of integrating the HKID provider into a client application.

# 5 Implementation Details

The content below covers the technology framework utilized, tools chosen for various parts of the project, and reasonings behind the implementation approaches.

## 5.1 Frontend Tech Stack

The technology frameworks that are chosen to build the HKID frontend application and the HKLIB client application are discussed and explained in the subsections below.

### 5.1.1 React.js

React.js, commonly referred to as React, is a JavaScript library for building user interfaces. Developed and maintained by Facebook, React has gained widespread popularity among developers for its simplicity, efficiency, and performance. React.js is renowned for its component-based architecture, which aligns seamlessly with the modular nature of modern web development. By breaking down the user interface into reusable components, React.js facilitates the creation of complex applications with ease. For the HKID Frontend Application and HKLIB client application, this modular approach enables us to design interactive user interfaces comprising distinct components such as login forms, authentication workflows, and user dashboards. Each component can be developed, tested, and maintained independently, fostering code reusability, and enhancing development efficiency.

Secondly, React.js leverages a virtual DOM (Document Object Model) to optimize rendering performance, mitigating the need for frequent updates to the actual DOM. This efficient rendering mechanism results in faster page loads and smoother user interactions, crucial for applications requiring real-time data updates and seamless navigation. In the context of the HKID Frontend Application and HKLIB client application, the virtual DOM ensures responsive user interfaces, particularly during authentication processes and data retrieval operations. Users can experience swift transitions between application screens and seamless interaction with authentication prompts, enhancing overall user satisfaction.

Other than that, React.js provides robust state management capabilities, empowering developers to manage application state efficiently across different components. With stateful components and context API, React.js facilitates the propagation of data and state changes throughout the application, ensuring consistency and synchronization across UI elements. This state management paradigm is particularly beneficial for applications like the HKID Frontend Application and HKLIB client application, where user authentication state, session information, and application settings need to be maintained and updated dynamically. React's state management mechanisms enable seamless integration with authentication flows, allowing for real-time updates and notifications to users.

Lastly, React.js facilitates the development of cross-platform applications, allowing developers to target multiple platforms with a single codebase. Through frameworks like React Native, React.js

extends its capabilities to mobile app development, enabling the creation of native iOS and Android applications using JavaScript and React principles. While the HKID Frontend Application primarily targets web browsers, the supplementary HKLIB client application may benefit from cross-platform compatibility, enabling users to access authentication services seamlessly across desktop and mobile devices. By leveraging React.js for both web and mobile development, developers can achieve consistency in UI design, code structure, and user experience, enhancing usability and accessibility for end users.

In summary, React.js offers a compelling set of features and advantages for building the HKID Frontend Application and the supplementary HKLIB client application. From its component-based architecture and efficient rendering to its robust state management and vibrant ecosystem, React.js provides developers with the tools and capabilities needed to create modern, responsive, and user-friendly authentication solutions. By harnessing the power of React.js, developers can deliver seamless authentication experiences across web and mobile platforms, catering to the diverse needs of users in the digital age.

### 5.1.2 Tailwind CSS

Tailwind CSS is a utility-first CSS framework that has gained significant traction among developers for its unique approach to styling web applications. Tailwind CSS follows a utility-first approach, wherein styling is achieved through a set of pre-defined utility classes rather than writing custom CSS rules. This approach offers unparalleled flexibility and efficiency in styling user interfaces, as developers can apply styles directly within HTML markup. For the HKID Frontend Application and HKLIB client application, this approach streamlines the styling process, enabling developers to rapidly prototype and iterate on UI designs without the need for extensive CSS customization.

Besides that, Tailwind CSS promotes modularity and reusability by encapsulating styling rules into small, composable utility classes. These classes can be combined and reused across different components, facilitating a consistent design language and codebase organization. By leveraging Tailwind's modular architecture, developers can maintain cleaner and more maintainable code, reducing redundancy and improving code readability.

Nonetheless, Tailwind CSS offers built-in support for responsive design, allowing developers to create adaptive layouts that seamlessly adapt to different screen sizes and devices. By utilizing responsive utility classes such as “sm:”, “md:”, and “lg:”, developers can define styles for specific breakpoints, ensuring optimal user experience across desktop, tablet, and mobile devices. This inherent responsiveness simplifies the development process and eliminates the need for media queries, resulting in cleaner and more maintainable code. For the HKID Frontend Application and HKLIB client application, responsive design capabilities are essential for delivering a seamless user experience across various devices and screen resolutions.

Last but not least, the most significant advantage of developing with Tailwind CSS is that it prioritizes performance optimization by generating minimal and optimized CSS output. Unlike traditional CSS

frameworks that include large sets of predefined styles, Tailwind generates only the CSS classes that are explicitly used in the application. This results in smaller file sizes and faster loading times, contributing to improved performance and user experience. For applications like the HKID Frontend Application and HKLIB client application, which require efficient resource utilization and fast page rendering, Tailwind's performance optimization features are highly advantageous.

In short, Tailwind CSS offers a multitude of advantages that make it a compelling choice for styling the HKID Frontend Application and the supplementary HKLIB client application. From its utility-first approach and modularity to its responsiveness, and performance optimization, Tailwind CSS provides us with the tools and capabilities needed to create visually appealing, responsive, and maintainable user interfaces that meet the diverse needs of modern web applications.

## 5.2 Backend Tech Stack

The technology framework that we utilized to implement the backend OAuth server is Node.js runtime. Node.js is built on a non-blocking, event-driven architecture, making it highly efficient for handling I/O-bound operations. This asynchronous nature allows the server to handle multiple concurrent connections without blocking the execution thread, thereby optimizing resource utilization and improving scalability. For the backend OAuth server, which may need to handle numerous authentication requests concurrently, Node.js' asynchronous I/O model ensures optimal performance and responsiveness, enhancing the server's ability to handle high traffic loads efficiently.

Moreover, Node.js is renowned for its ability to scale horizontally and vertically, making it well-suited for building highly scalable and performant applications. By utilizing clustering and load balancing techniques, Node.js enables the distribution of incoming requests across multiple CPU cores, maximizing throughput and concurrency. This scalability ensures that the backend OAuth server can accommodate increasing numbers of concurrent users and handle spikes in traffic without sacrificing performance. Furthermore, Node.js' lightweight runtime and efficient event loop contribute to reduced memory overhead and faster response times, enhancing the server's overall performance.

In addition, Node.js comes with a lightweight and flexible web application framework called Express.js. Express.js simplifies the process of building web servers, including the creation of routes, handling HTTP requests, and managing middleware. Express.js is highly extensible, allowing us to extend its functionality through the use of third-party middleware and plugins. The NPM (Node Package Manager) ecosystem offers a wide range of Express.js middleware modules that provide additional features and integrations, such as authentication, session management, caching, compression, and logging. By leveraging these middleware modules, we can enhance the capabilities of our OAuth server without reinventing the wheel. Thus, it provides a solid foundation for building OAuth servers.

## 5.3 Database

Our choice of database would be MongoDB. MongoDB is a NoSQL database that employs a flexible document-based data model. Unlike traditional relational databases, MongoDB stores data in JSON-like documents, allowing for a more natural representation of complex data structures. This flexibility is advantageous for an OAuth server, as it accommodates the diverse and evolving nature of user authentication and authorization data. OAuth-related entities such as users, access tokens, refresh tokens, authorization codes, and client applications can be stored as JSON documents, enabling seamless storage and retrieval of data without the need for complex schema migrations.

MongoDB is designed to scale horizontally, allowing developers to distribute the OAuth server across multiple machines to handle high traffic loads and concurrent requests that are typically associated with an OAuth server. It employs a distributed architecture that enables seamless data partitioning and replication across multiple nodes, allowing the database to scale out as the application grows. This scalability ensures that the OAuth server can handle increasing user traffic, client applications, and authentication requests without sacrificing performance or availability.

In our use case, MongoDB's schema-less design eliminates the need for predefined schemas and rigid data structures, providing flexibility and agility in data modelling as compared to relational SQL databases. This is particularly beneficial for an OAuth server, where the data schema may evolve over time to accommodate new requirements or extensions to the OAuth protocol. MongoDB allows developers to store heterogeneous data types within the same collection, enabling seamless integration of new attributes or properties without disrupting existing data. This schema-less design accelerates development cycles and facilitates iterative development of the OAuth server, enabling us to adapt to changing requirements quickly.

In summary, MongoDB offers a compelling set of features and capabilities that make it a suitable choice for serving as the database for our HKID OAuth server. Its, scalability, high performance, and schema-less design make it well-suited for storing and managing OAuth-related data effectively and efficiently. By leveraging MongoDB, a robust and scalable HKID OAuth server that do not compromise for security, performance, and scalability is implemented.

## 5.4 Application Deployment

For the HKID Frontend Application and the supplementary HKLIB client application, Vercel is chosen as the cloud platform for production deployment. Vercel's serverless architecture eliminates the need for managing infrastructure and server maintenance tasks, allowing developers to focus solely on building and deploying their applications. With serverless deployment, our applications can leverage Vercel's cloud platform to automatically scale resources based on demand, ensuring optimal performance and cost efficiency. This serverless approach reduces operational overhead and accelerates time-to-market for the applications, enabling developers to deliver features and updates more rapidly.

Another attractive point of Vercel is its automatic deployment and continuous integration. Vercel provides seamless integration with version control systems such as Git, enabling automated deployment workflows for the HKID Frontend Application and HKLIB client application. With Vercel's continuous integration (CI) capabilities, developers can trigger automatic deployments whenever changes are pushed to the repository, ensuring that the latest version of the applications is always deployed to production. This automated deployment process streamlines development workflows, reduces manual intervention, and facilitates rapid iteration and testing of new features and updates.

Furthermore, Vercel supports custom domain configurations and provides built-in SSL certificate management, allowing us to map custom domains to HKID Frontend Application and HKLIB client application deployments with ease. By using custom domains, a branded and professional online presence could be established for the applications, enhancing brand recognition and user trust. Additionally, Vercel's automatic SSL certificate provisioning ensures that all communications between users and the applications are encrypted and secure, protecting sensitive data and ensuring compliance with security best practices.

As for the OAuth backend server, it is deployed on Heroku. The main reason choosing Heroku deploy the backend server is due to the ease of deployment. Similar to Vercel, Heroku provides a streamlined deployment process for Node.js applications, allowing developers to deploy their OAuth server with just a few simple commands. Heroku's Git-based deployment model enables pushing code changes directly to the Heroku Git repository, triggering automatic builds and deployments. This ease of deployment accelerates the development lifecycle and enables us to iterate and ship features more quickly.

## 5.5 Magic Link User Authentication Method

After thorough consideration and contemplation, the original user authentication method of using the traditional password-based approach is eliminated and replaced with the magic link authentication method. Replacing the traditional password-based approach with the implementation of magic link authentication brings numerous benefits and significantly enhances security against online attacks, particularly phishing attempts. Magic links offer a more robust and secure user authentication method by eliminating the need for users to remember complex password, which are often prone to being compromised. In this way, the likelihood of weak or reused credentials is greatly reduced. As a result, the attack surface for password-based attacks is significantly reduced as well. Magic links operate by sending a unique, time-limited URL to the user’s registered email address. This URL serves as a one-time login token, effectively replacing the need for a password. These tokens are valid for a short period of time, after which they expire and become useless. This time-bound nature adds an extra layer of security by reducing the risk of stolen or intercepted tokens being misused. Even if an attack manages to obtain a magic link, it loses its validity before they can exploit it, rendering their efforts futile. Furthermore, magic links provide an added layer of protection against phishing attacks [7]. Phishing attempts typically involve tricking users into disclosing their password on malicious websites that mimic legitimate ones. Since magic links are delivered directly to the user’s trusted

email, the likelihood of victims falling to such attacks is greatly reduced. Even if a user unknowingly clicks on a phishing link, the magic link authentication process ensures that the attacker cannot gain unauthorized access without possessing the unique URL sent to the user's trusted device. In summary, the adoption of magic link authentication method over the traditional password-based approach significantly strengthens security against online attacks, particularly phishing attempts. By eliminating the need from passwords and employing time-limited, unique URLs, magic links provide a more resilient authentication method that reduces the risk of weak passwords, password reuse, and phishing attacks for our HKID OAuth provider. This enhanced security ensures greater protection for user accounts and sensitive information, establishing a more robust foundation for our OAuth server. This is crucial as the user accounts are meant to represent the digital version of their HKID hence the security of the server is of our primary consideration. An example of the magic links sent to users' email inbox is exemplified in Fig. 11 below.

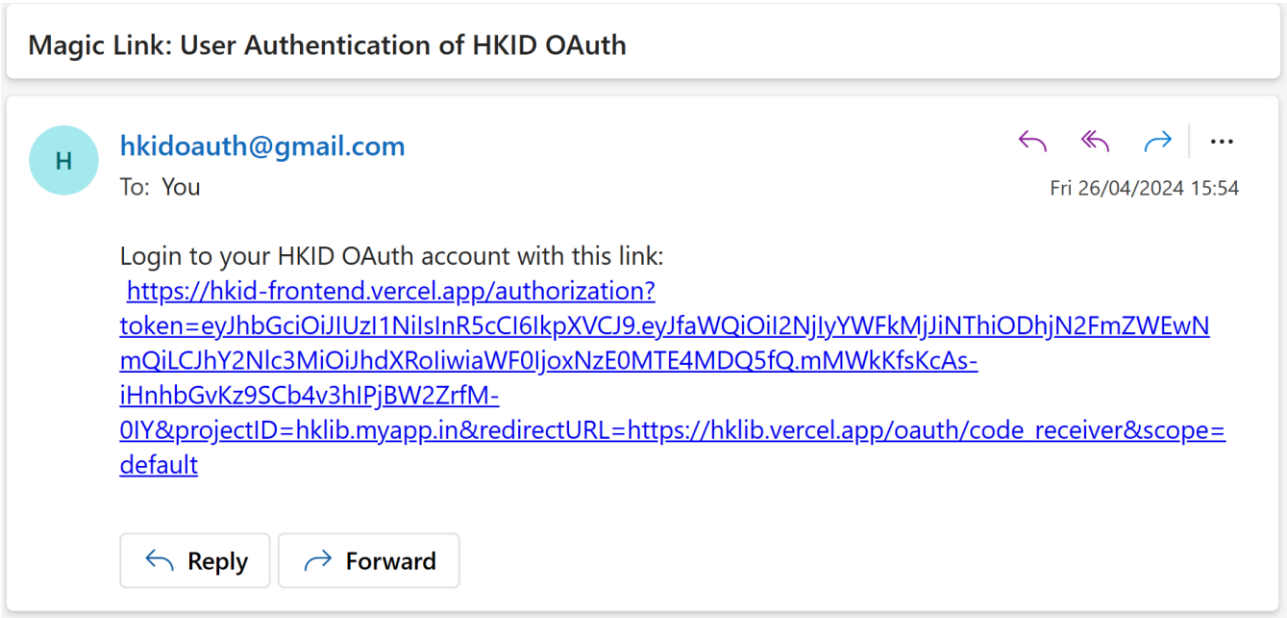


Fig. 11. Screencap of Magic Link Example

# 6. Difficulties Encountered

As mentioned in section 3.1, the authorization grant type that we implemented was the Authorization Code Flow. After the whole OAuth workflow was implemented, more in-depth extensive research was conducted on the security level of the workflow. It was only discovered that this approach is susceptible to a security threat called Authorization Code Interception.

In the Authorization Code flow, the client application initiates the authentication process by redirecting the resource owner's user agent (e.g., web browser) to the authorization server's authorization endpoint. The client includes its client identifier and the requested scope of access as parameters in the request. Upon successful authentication, the authorization server issues an authorization code to the client via the resource owner's user agent, typically as a query parameter in the redirection URI.

The vulnerability arises during the transmission of the authorization code from the authorization server to the client application via the resource owner's user agent. Since the authorization code is typically transmitted as part of the URI query parameters, it is susceptible to interception by malicious actors. If an attacker gains access to the user agent or intercepts the HTTP request/response traffic between the user agent and the authorization server, they can potentially capture the authorization code and misuse it to obtain access tokens unauthorizedly.

In a typical Authorization Code injection/interception attack scenario, an attacker may exploit vulnerabilities in the client application or the user agent to intercept the authorization code during transmission. This interception can occur through various means, including man-in-the-middle (MITM) attacks, cross-site scripting (XSS) vulnerabilities, or compromised client devices. Once the attacker obtains the authorization code, they can use it to impersonate the legitimate client application and exchange it for access tokens without the resource owner's consent.

An illustration of the possible cyberattack mentioned is given in Fig. 12.

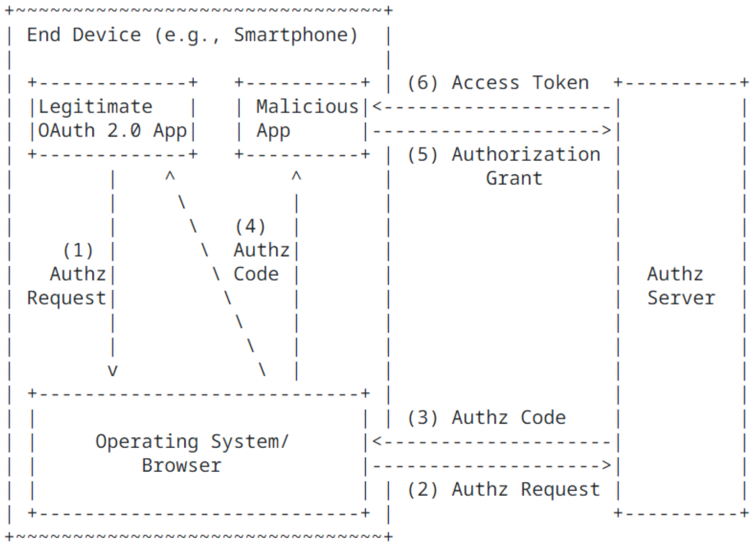


Fig. 12. Authorization Code Interception Attack Flow. [5]

For instance, if a malicious application is planted on the client device, when the authorization code is sent back to the client application via the browser in step (4), the malicious application could intercept the authorization code and in turn request and obtain an access token by going through steps (5) and (6) respectively. As the communication path in steps (1) and (2) is directed to the authorization server, if OAuth Protocol is implemented correctly with TLS version, the transport channel is secure from any attempts to intercept [6].

With that being said, although our OAuth server is HTTPS enabled, the client application is not guaranteed to be supporting the HTTPS protocol that encrypts the data communication process. The redirection URL that the client provides could be an URL that only follows the HTTP protocol. As an OAuth provider, it is unreasonable and too expensive to force every client application to obtain a signed digital certificate.

To mitigate the risk of Authorization Code injection/interception attacks, an extension to the Authorization Code flow called Proof Key for Code Exchange (PKCE), pronounced as ‘pixey’ is utilised. The abstract flow utilizing PKCE practice is visualized in Fig. 13.

```
                                          +-------------------+
                                          |   Authz Server    |
+--------+                                | +---------------+ |
|        |--(A)- Authorization Request ---->|               | |
|        |       + t(code_verifier), t_m  | | Authorization | |
|        |                                | |   Endpoint    | |
|        |<-(B)---- Authorization Code -----|               | |
|        |                                | +---------------+ |
| Client |                                |                   |
|        |                                | +---------------+ |
|        |--(C)-- Access Token Request ---->|               | |
|        |          + code_verifier       | |    Token      | |
|        |                                | |   Endpoint    | |
|        |<-(D)------ Access Token ---------|               | |
+--------+                                | +---------------+ |
                                          +-------------------+
```

Fig. 13. Abstract Authorization Code Flow with PKCE. [5]

PKCE basically creates a randomly-generated cryptographical key called “code verifier” for every authorization request and then the code verifier is transformed into a value called “code challenge” [5]. The “code challenge” derived from the “code verifier” and the hashing method are then sent to the authorization endpoint of the authorization server along with the request for the authorization code. These values are then stored securely in the server, grouping them up with the corresponding authorization code. Later on, the “code verifier” is then sent along with the access token request to the token endpoint. Afterwards, the authorization server hashes the “code verifier” it just obtained and compares it with the “code challenge” that it stored previously to verify that the client requesting for access token is the same as the one asking for the authorization code. This could effectively mitigate the interception attack as the attacker would not know this one-time key as it is sent over TLS.

In our backend authorization server, the PKCE workflow was implemented as two middleware functions: receiveCodeChallenge and verifyCodeChallenge. The receiveCodeChallenge middleware was integrated into the Authorization Endpoint as shown before in Fig. 14 and the verifyCodeChallenge middleware was included in the Token Endpoint as shown in Fig. 15. The PKCE workflow discussed was thoroughly implemented in the code snippets below:

```
99  // PKCE methods
100 var receiveCodeChallenge = function (req, res, next) {
101     var codeChallenge = req.query.code_challenge;
102     var codeChallengeMethod = req.query.code_challenge_method;
103     if (codeChallenge && codeChallengeMethod && codeChallengeMethod == "S256") {
104         req.session.codeChallenge = codeChallenge;
105         req.session.codeChallengeMethod = codeChallengeMethod;
106         next();
107     } else {
108         res.status(400).send({
109             code: 400,
110             message: "Invalid code_challenge or code_challenge_method",
111         })
112     }
113 }
```

Fig. 14. Code snippet of receiveCodeChallenge middleware [5]

```
115 var verifyCodeChallenge = function (req, res, next) {
116     var codeChallenge = req.session.codeChallenge;
117     var codeChallengeMethod = req.session.codeChallengeMethod;
118     // var codeVerifier = req.query.code_verifier;
119     var codeVerifier = "MQ";
120     console.log(codeChallenge, codeChallengeMethod, codeVerifier)
121     if (codeChallenge && codeChallengeMethod && codeVerifier) {
122         var hash = crypto.createHash("sha256");
123         hash.update(codeVerifier);
124         var hashedVerifier = hash.digest("base64");
125         console.log(hashedVerifier)
126         if (hashedVerifier == codeChallenge) {
127             next();
128         } else {
129             res.status(400).send({
130                 code: 400,
131                 message: "Invalid code_verifier",
132             })
133         }
134     } else {
135         res.status(400).send({
136             code: 400,
137             message: "Invalid code_challenge or code_challenge_method",
138         })
139     }
140 }
```

Fig. 15. Code snippet of verifyCodeChallenge middleware [5]

Here the hashing method of code challenge is explicitly required to be the SHA256 hashing algorithm.

SHA-256 is part of the SHA-2 (Secure Hash Algorithm 2) family, which is designed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST) [8]. It provides a high level of cryptographic security against various attacks, including pre-image attacks, collision attacks, and birthday attacks. The 256-bit output size of SHA-256 makes it computationally infeasible to reverse engineer the original input data from the hash value, ensuring data integrity and confidentiality.

Not only it is highly secure, SHA-256 is optimized for efficient computation on modern hardware architectures, including CPUs, GPUs, and dedicated cryptographic accelerators. Its algorithmic design allows for parallel processing and optimized implementations, resulting in fast and efficient hashing performance across a wide range of platforms and devices. This performance efficiency is crucial for real-time applications, cryptographic protocols, and systems with stringent latency requirements.

## 7. Future Improvements

This section states the possible enhancements to be added into the project in the future where subsection 7.1 introduces the Passkey Authentication and subsection 7.2 introduces the rate-limiting mechanism.

### 7.1 Passkey Authentication

Currently, the only possible security weakness that could endanger the authentication system would be the user authentication process. Although the user authentication method has been replaced by magic links and the resistance to online attacks has been greatly increased compared to the traditional password-based approach, email accounts that receive the magic links are still safeguarded by passwords set by users in general. One of the most common cyber threats is credential stuffing, where attackers use leaked username and password combinations to gain unauthorized access to user accounts. Malicious attackers could authenticate as the user at anywhere in the world without being around the user as long as they obtained the passwords or they got access to the user's email account. The solution to this issue is a passwordless authentication scheme called Passkey Authentication.

Passkey authentication is a robust and secure method of user authentication that relies on the use of cryptographic keys, also known as passkeys, to verify the identity of individuals accessing a system or service [9]. Unlike traditional password-based authentication methods, which involve memorized secrets that can be vulnerable to various attacks such as brute force, phishing, and credential stuffing, passkey authentication offers a higher level of security and resilience against unauthorized access.

In passkey authentication, each user is issued a unique cryptographic key pair consisting of a public key and a private key. The public key is freely distributable and can be shared with others, while the private key is

kept confidential and known only to the user. When a user attempts to authenticate to a system or service, they present their public key as proof of identity.

The authentication process typically involves a challenge-response mechanism, where the system generates a random challenge and sends it to the user. The user then signs the challenge using their private key and returns the signed response to the system. The system verifies the signature using the user's public key and grants access if the signature is valid.

Passkey authentication offers a seamless and user-friendly authentication experience. Users do not need to remember passwords or undergo cumbersome authentication procedures, resulting in faster and more convenient access to systems and services.

Overall, passkey authentication represents a secure and reliable method of user authentication that addresses many of the shortcomings of traditional password-based authentication methods. Integrating passkey into HKID provider makes the system completely resilient to online phishing attack which is one of the most common ways that users leak their passwords.

### 7.2 Rate limiting

Rate limiting is a crucial component of any OAuth server implementation, serving as a fundamental safeguard against various security threats and ensuring the stability, reliability, and fair usage of the authentication service. It helps mitigate the risk of brute force attacks, where attackers systematically attempt to guess valid access tokens or authorization codes by making a large number of authentication requests. By imposing limits on the number of requests that can be made within a specified time frame, rate limiting makes it significantly more difficult for attackers to launch successful brute force attacks and gain unauthorized access to user accounts.

Considering the numerous instances of Denial of Service (DoS) assaults that have occurred recently, an OAuth server that does not use rate limiting is susceptible to DoS attacks, in which an attacker overwhelms the server with a large number of requests, rendering it inaccessible or unresponsive. By capping the number of incoming requests, rate limiting ensures that the server can manage genuine traffic while blocking malicious efforts to overload its resources, hence lessening the impact of DoS assaults.

Even in this case of legitimate users, rate limiting brings great benefits to the OAuth server as well. Rate limiting helps prevent resource exhaustion and degradation of server performance by limiting the rate at which clients can make authentication requests. Without rate limiting, legitimate users may inadvertently or intentionally consume excessive server resources, leading to declined server performance, increased latency, and potential service disruptions for all users. Rate limiting ensures equitable distribution of server resources and maintains optimal performance for all clients.

# 8 Project Schedule & Current Progress

## 8.1 Project Scope

The scope of this project includes:

- Development of an OAuth 2.0 with PKCE server
- Development of a supplementary third-party client application

The scope of this project does not include:

- Client registration process with the authorization server
- User registration process

## 8.2 Project Plan & Timeline

A Gannt Chart is produced to ensure that the project progresses smoothly and can be monitored easily. The tasks allocated as well as the timelines are defined in the Gantt Chart show in Fig. 16.

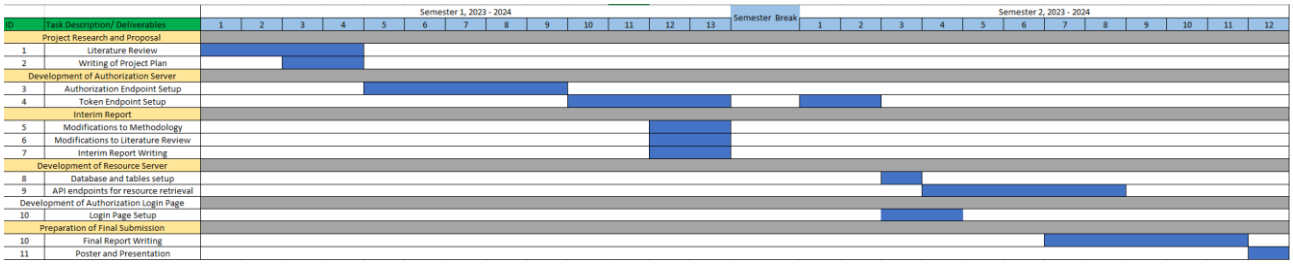


Fig. 16. Project Gantt Chart that displays the timeline of the project in a simple and more readable format

Throughout the semester year 2023/24, all the scheduled tasks were finished in ahead of schedule. Extra tasks such as implementing magic link authentication and was added into the schedule and completed successfully.

# 9 Risk Management Plan

Table 1 from subsection 9.1 and Table 2 from subsection 9.2 below combine to provide an intuitive way for audience to understand the possible underlying risks and consequences of the subsequent risks.

## 9.1 Risk Assessment Matrix

| Consequence and Likelihood | Insignificant | Minor | Serious | Disastrous | Catastrophic |
|---|---|---|---|---|---|
| Rare | L | L | L | L | M |
| Unlikely | L | L | M | M | S |
| Possible | L | M | M | S | H |
| Likely | L | M | S | H | E |
| Almost Certain | M | S | H | E | E |

Table 1: Risk Assessment Matrix which provides a scale to assess the risk of the project based on the likelihood and the serious level of the risks

L = Low Risk, M = Medium Rish, S = Substantial Risk, H = High Risk, E = Extreme Risk

## 9.2 Non-OHS Risk Assessment

| Project Risk | Risk | Likelihood | Consequence | Risk Level | Mitigation | Residual Risk |
|---|---|---|---|---|---|---|
| Unbearably high traffic load | Users fail to be authenticated | Possible | Serious | M | Utilize Elastic Load Balancing (ELB) Service to allow flexible adaptions to intensive CPU usage | Lost users' trust in our OAuth application |
| Incorrect implementation of OAuth 2.0 protocol | Weak security, susceptible to malicious attack | Possible | Disastrous | S | Careful implementation following research documentation and thorough integration testing | Users' data leaked and reputation harm |
| Inappropriate database design | Non-scalable when userbase grows, server operations become irresponsive | Possible | Disastrous | S | Consulting MongoDB experts to better model the tables | All existing database structures are required to be erased |

Table 2: Non-OHS Risk Assessment which lists out the potential project risks, its corresponding risk level, and the mitigation techniques

# 10 Conclusion

In conclusion, this comprehensive report navigates the intricate landscape of online identity verification, offering a robust solution anchored in the OAuth 2.0 framework. The escalating need for stringent real-name authentication across various online platforms underscores the urgency for a streamlined and secure verification process. By exploring the potential integration of OAuth 2.0 as an authentication platform, this project not only addresses the immediate demand for enhanced security but also envisions a future where authentication processes are simplified and standardized.

The research conducted delves deeply into the OAuth 2.0 protocol, meticulously dissecting its principles, grant types, and implementation nuances across web and mobile applications. In doing so, it identifies key security considerations and best practices, recognizing potential vulnerabilities and offering strategic countermeasures to mitigate risks effectively.

Moreover, the literature review underscores the significance of OAuth 2.0 as an authorization framework, highlighting its pivotal role in ensuring secure access to resources in online environments. By harnessing the power of OAuth 2.0, this project endeavours to revolutionize authentication paradigms, contributing significantly to the realms of cybersecurity, delegated access control, and user privacy.

Central to the project's objectives is the development of an OAuth 2.0 provider tailored for real identity verification, manifested in the creation of a frontend login application, an authorization server, and a resource server. Leveraging cutting-edge technologies such as Node.js, Express.js, React.js, and MongoDB, the project adopts a forward-thinking approach to backend, frontend, and database development, ensuring scalability, reliability, and performance optimization.

Furthermore, the successful deployment of the OAuth HKID provider to production marks a significant milestone, with tangible deliverables including the authorization server login page, resource server, and frontend login application. The incorporation of Magic links enhances the system's security measures, laying a robust foundation for future enhancements and performance optimizations.

In essence, this report culminates by reaffirming the project's overarching objective: to deliver a secure, efficient, and future-proof OAuth provider for real identity verification. By embracing innovation, adhering to best practices, and prioritizing security, the project sets a precedent for transformative advancements in online authentication paradigms, heralding a new era of trust and reliability in digital interactions.

# 11 References


[1] "Digital Around the World", https://datareportal.com/global-digital-overview

[2] J. Lee, C. Liu, "Real-Name Registration Rules and the Fading Digital Anonymity in China," Washington International Law Journal, Vol 25, No. 1, 2016. https://digitalcommons.law.uw.edu/cgi/viewcontent.cgi?article=1717&context=wilj

[3] L. John, "Identifying the Problem: Korea's Initial Experience with Mandatory Real Name Verification on Internet Portals," Journal of Korean Law, Vol. 9, 83-108, December 2009. https://s-space.snu.ac.kr/bitstream/10371/85159/1/4.%20Identifying%20the%20Problem%20Korea%E2%80%99s%20Initial%20Experience%20with%20Mandatory%20Real%20Name%20Verification%20on%20Internet%20Portals.pdf

[4] D. Hardt, "RFC 6749 The OAuth 2.0 Authorization Framework," Internet Engineering Task Force (IETF), October. https://datatracker.ietf.org/doc/html/rfc6749

[5] N. Sakimura, J.Bradley, and N. Agarwal, "RFC 6749 The OAuth 2.0 Authorization Framework," Internet Engineering Task Force (IETF), October. https://datatracker.ietf.org/doc/html/rfc7636

[6] T. Lodderstedt, M. McGloin, P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, http://www.rfc-editor.org/info/rfc6819.

[7] "Clerk", https://clerk.com/blog/magic-links

[8] "BitcoinWiki", https://bitcoinwiki.org/wiki/sha-256

[9] "FIDO Alliance", https://fidoalliance.org/specifications/