A Design Specification Document for the course CS5617 (Software Engineering)

Design Specification Document

by

UX Team

Team Lead

Neel Kabra (111901057)

Team Members

Jasir K (111901025)

Parichita Das (111901039)

Under the supervision of

Ramaswamy Krishnan Chittur

INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD

Department of Computer Science and Engineering

Kerala, India - 678557

Authentication Module

Parichita Das

(111901039)

---

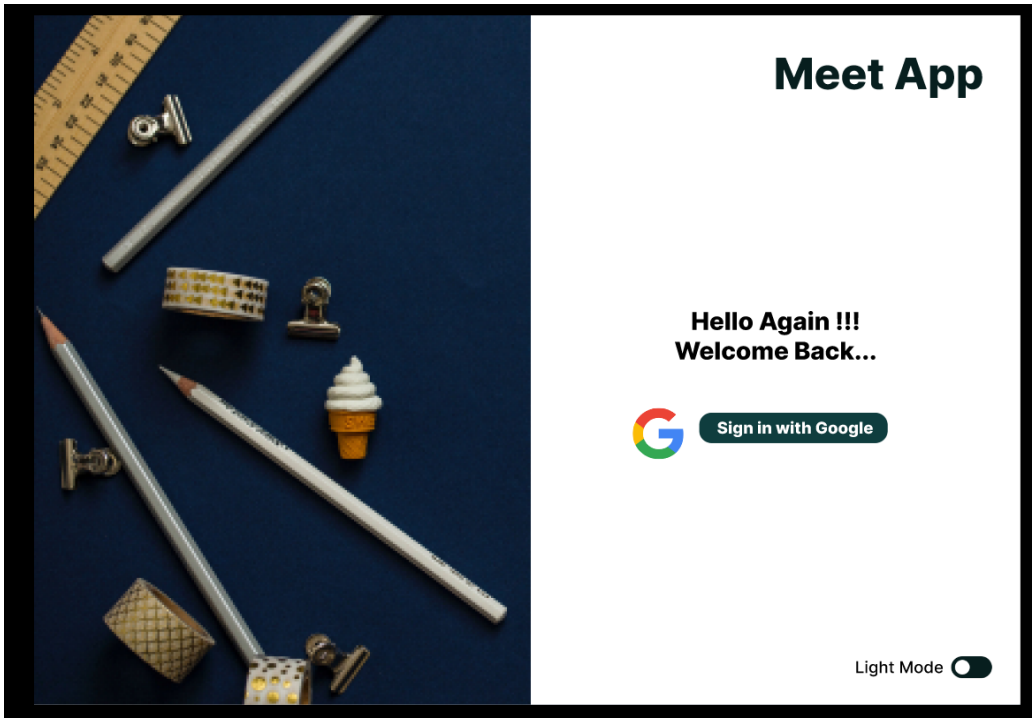
1. Overview

The UX team for this project is responsible for joining all the UX components of all other teams. In this document, you will find how the **Authentication screen** works. On starting up the application, after the splash screen, this is the first screen that a user (host/client) sees. This screen is mainly focussing on the authentication of the user trying to use this application.

1.1 Google OAuth 2.0

We wish the UI to look as following for Google Authentication:

- On the screen, we have a Google Sign In button. Clicking on this will take the user to the google page where they can enter their Google email. We will only accept IIT Palakkad *smail* users.
- This will give us information about the *Name, Email ID, Roll Number* (if any), and *Profile Picture (*if any), from the user's google account. We would store this information in some form of class Object/ Struct.



1.2 Manually entering identity

We wish the UI to look as following for Manual Authentication:

- We will have row wise tabs to enter *Name, Roll Number / ID Number, Email ID* and option to upload *Profile Picture.* Name, ID and Email ID **must** be entered.
- We will have a *Sign In* button at the bottom. Clicking on this, we store the information provided in the form of class Object/Struct.

The UX team's target is to achieve the first method, the second method can be used as a backup.

2. Objective

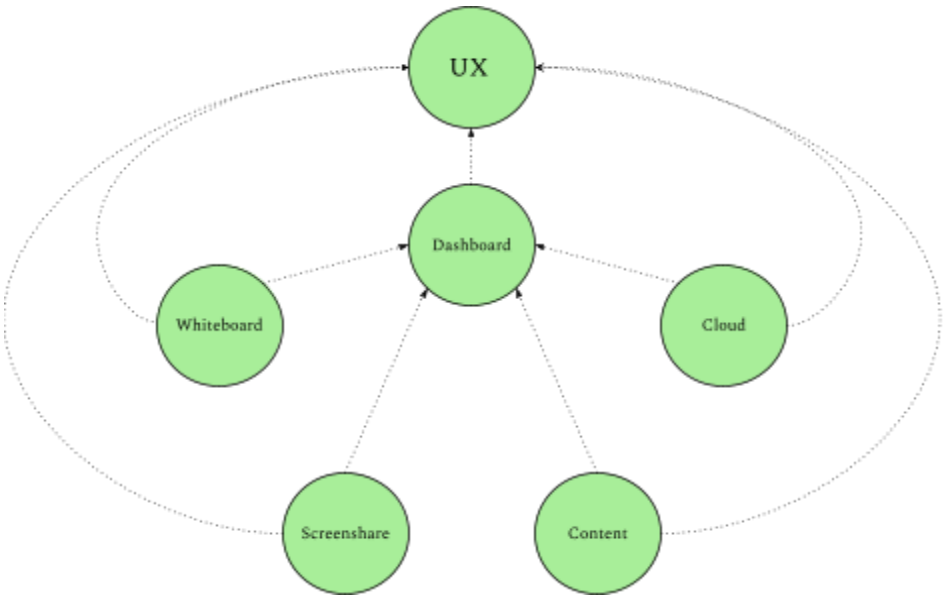
The main objectives of the Authentication Screen are listed below:

- We wish to give entry to our application only to users who belong to our organization. Using Google Authentication we can reject entry to users without an smail ID.
- We also wish to capture the different information of the user without having to manually enter it. We want to further send this information to the *Dashboard Module*.
- We want to provide a friendly interface for all the mentioned features.
- In case of Manual Authentication, we just ensure the email ID entered belongs to our organization. (There is no way to authenticate the user though)
- Implementing an *AuthView* and AuthViewModel which will be used for binding the click button to retrieve the information and authenticate it.

3. UML

In this section we will discuss the different types of interactions between different sections of the entire codebase.

3.1 Module Diagram

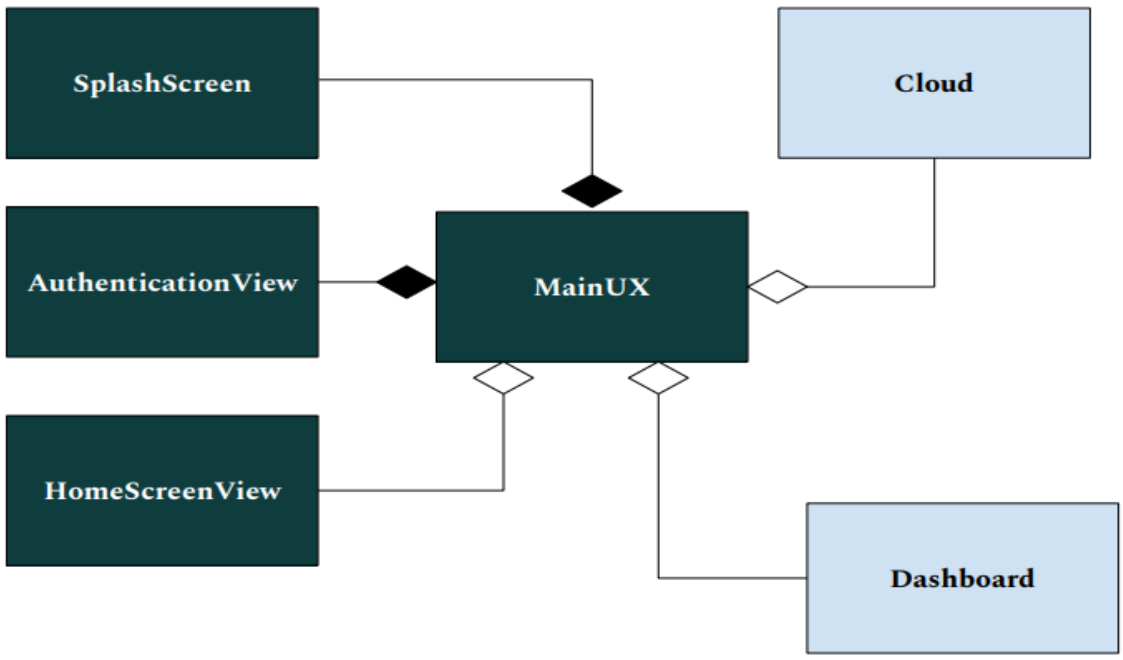


In the above diagram, as we can see, the UX module is dependent on all the other modules except Networking. Dashboard is responsible for creating the other modules once it gets instantiated. The different modules then send us their respective UIs which we will display in a *Navigation Tab Layout.*

3.2 Class Diagram

There are two class diagrams that we will be discussing here. The first one is for the overall UX, and the second one is specific to the *Authentication Module.*

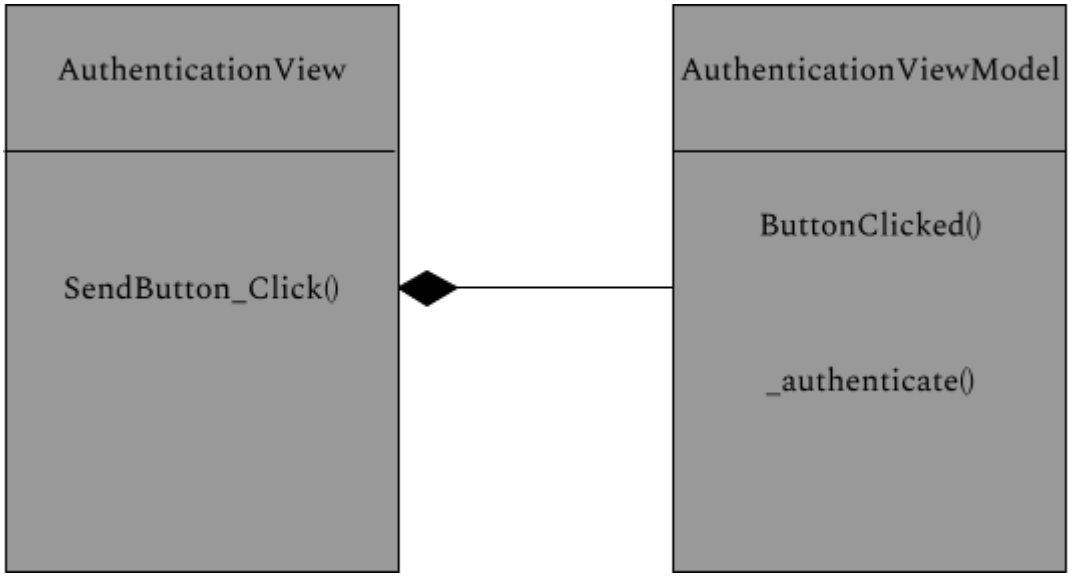
3.2.1 Main UX Class Diagram



In the above diagram, it is clear that MainUX is composing the *SplashScreen* and *AuthenticationView* as both of them must be constructed if *MainUX* gets instantiated. Once Authentication is done, it returns the struct/object with User Information to the *MainUX*. It might happen that the user does not get authenticated and hence the

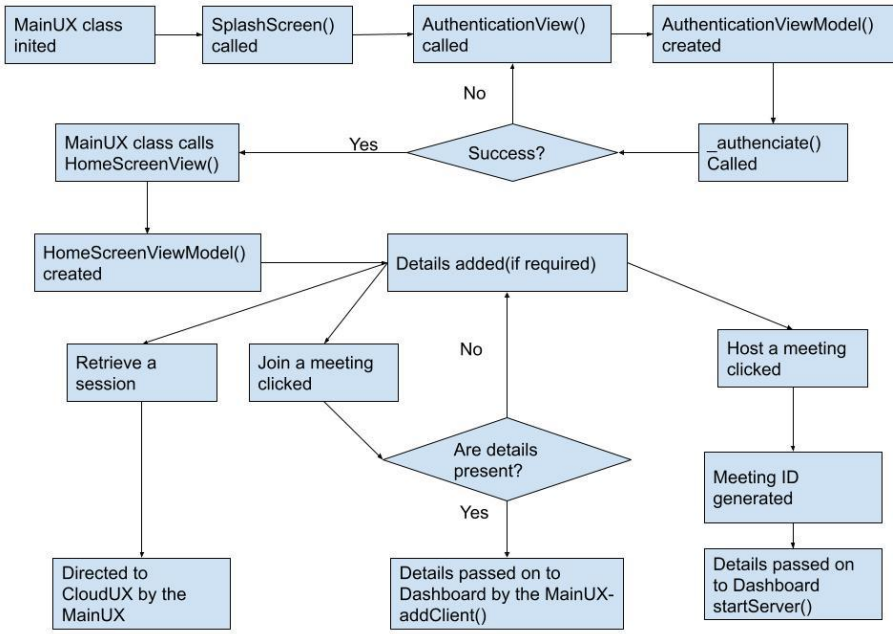
MainUX will not proceed from here onwards. In the other case the MainUX will now create a *HomeScreenView* object and pass user information to it. The Home Screen is now in charge of what the next steps from here will be. In case of Join/Host a new meeting, the Home Screen module sends back respective information to the *MainUX* asking it to create a *Dashboard* object, otherwise in case of retrieving an old session, the *MainUX* is redirected to create a *Cloud* object. This way *MainUX* aggregates *HomeScreen*, *Dashboard*, and *Cloud* modules.

3.2.2 Authentication Module Class Diagram



As we are using an MVVM (Model-View-Viewmodel) design pattern, for the authentication module, we only have a view and a view model. The view must compose the View Model, and likewise the View Model must bind to the View.

3.3 Activity Diagram



In the above diagram we can clearly understand the flow of the UX Module which includes the Authentication and the Home Screen modules.

4. Design

We are using an MVVM design pattern as it gives *extensibility (*We can easily extend our project), *testability* (We can easily test or debug our codes) and *clear understanding* (We have a very transparent understanding of the connections between different modules).

4.1 View and View Model

We have the XAML file containing the button for 'google sign in' in the *AuthenticationView*. In this class we will be binding the *AuthenticationViewModel* class object to itself.

```
public AuthenticationView() {
	AuthenticationViewModel viewModel = new AuthenticationViewModel();
	this.DataContext = viewModel;
}
```

We do not need to create any model in our case as we just need to send our data back to the *MainUX.*

In this we have a *SendButton_Click* to notify the ViewModel that we need to authenticate.

```
private void SendButton_Click(object sender, RoutedEventArgs e)
{

	AuthenticationViewModel viewModel = this.DataContext as
AuthenticationViewModel;
	viewModel.ButtonClicked = true;
}
```

In the view model we have a *_authenticate* function call to take care of the OAuth.

```
public bool ButtonClicked
{
	return _authenticate();
}
```

Now once authentication is done, we store all the information of the user in a struct such as this:

```
struct User
{
	public string Name;
	public string Email;
	public string UUID;
	public string Picture_Address;
```

```
}
```

Now we simply call the endpoint function provided by *HomeScreen* to send this struct and continue the further processes.

4.2 Using OAuth 2.0

Now we are trying to use Google to authenticate the sign in. On clicking the above mentioned button on the frontend, in the backend we are redirected to a google authentication method. For doing so we will be using **Google OAuth 2.0 Client.**

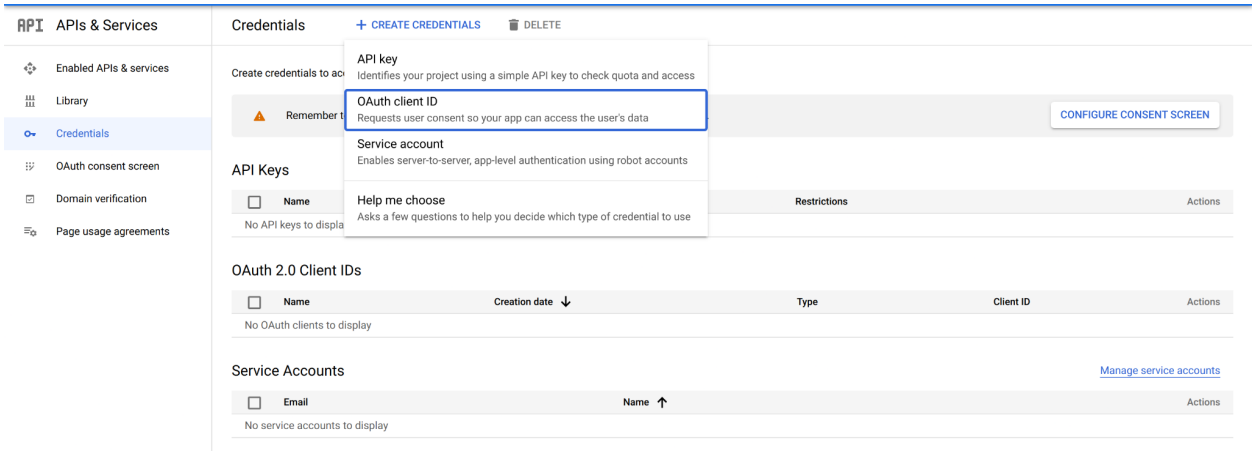
OAuth is useful when two completely unrelated sites or services are trying to accomplish something on behalf of their common users. When a user has already signed in to one of the said services, and is now trying to access the other site/service, the following things happen:

- The first site connects to the latter using the user's identity. The second site generates a one-time **request token** and **secret** for the initiating user's client software which is unique to the transaction and both parties involved.
- This token is then provided to the authority, who then authenticates the user.
- Once authenticated, the client is asked to approve the transaction to the second website. The request token is now an **approved access token**.
- Now this token is given to the second website as a proof of authentication on behalf of the user.
- Now the transaction is completed and the first website will be accessing the second on behalf of the user.

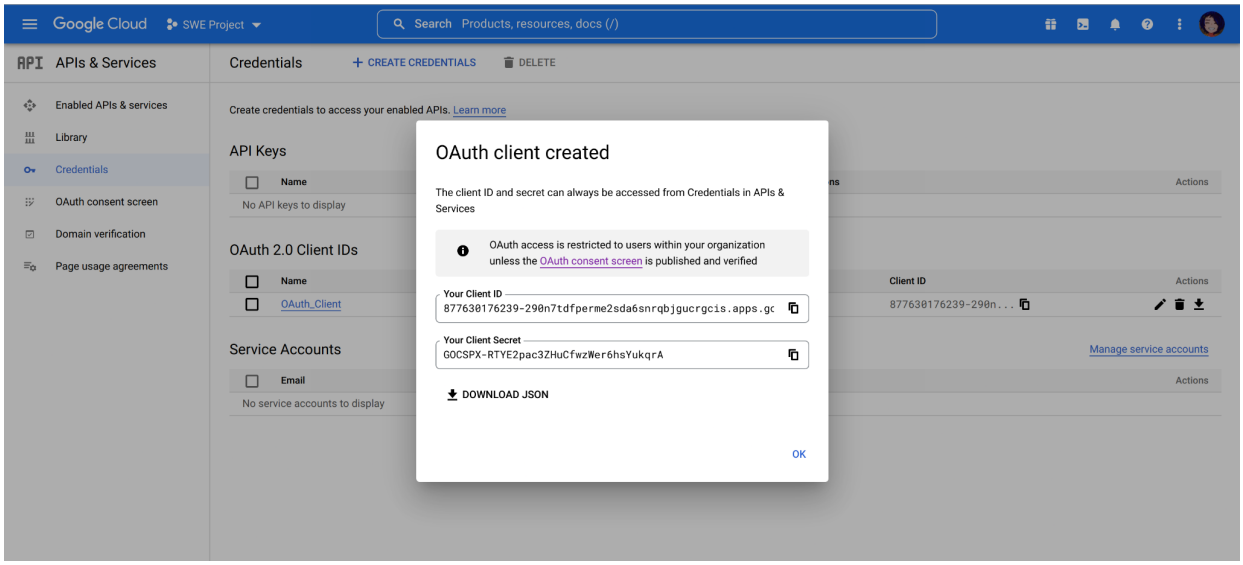
In order to use OAuth 2.0 to access Google API, we must have the correct permissions for the application in Google's OAuth 2.0 server.

For authorizing the application, we must follow the following steps:

- Go to credentials page of Google Cloud, and click **Create Credentials** and create **OAuth client ID:**



- After following the steps mentioned on the screen, we can see our client_id and client_secret_key:



- We now store these keys in our function and call *AddAuthentication:*

```
var builder = WebApplication.CreateBuilder(args);
var services = builder.Services;
var configuration = builder.Configuration;
```

```
services.AddAuthentication().AddGoogle(googleOptions =>
{
    googleOptions.ClientId = configuration["Authentication:Google:ClientId"];
    googleOptions.ClientSecret =
configuration["Authentication:Google:ClientSecret"];
});
```

Subsequent calls to AddAuthentication override previously configured AuthenticationOptions properties. Once authentication is complete, we will be ready to send the required information to the next module.

5. Interfaces

We do not use any *Interfaces* for Authentication.

6. Analysis

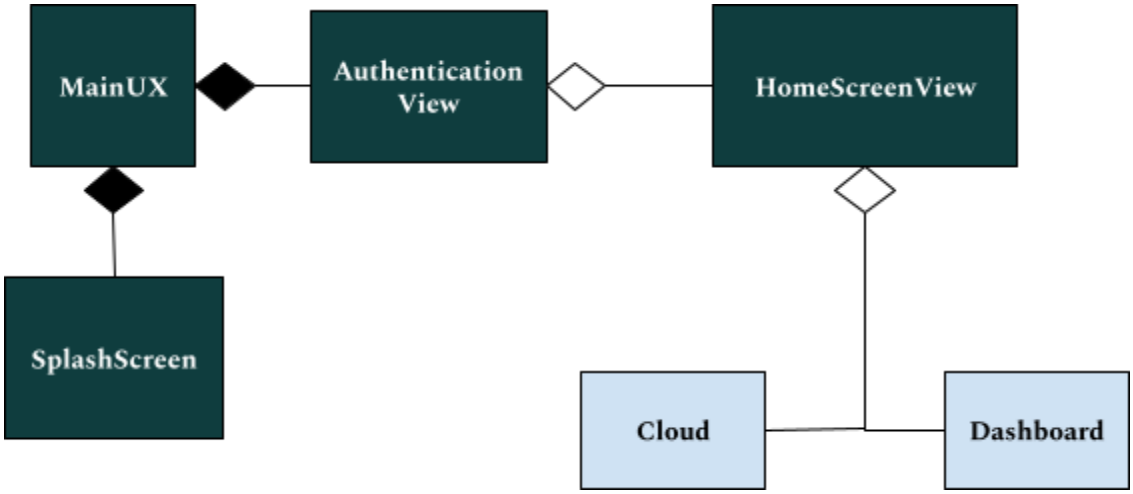
First let us analyze some of the designs we used and why.

6.1 Design Pattern

MVVM gives testability and extensibility. The code modules we write makes it easy to debug. That is why we are preferring it over other design patterns.

6.2 Class Diagram

We had thought of one more way to handle endpoints in our module other than the one already mentioned. It is as follows:



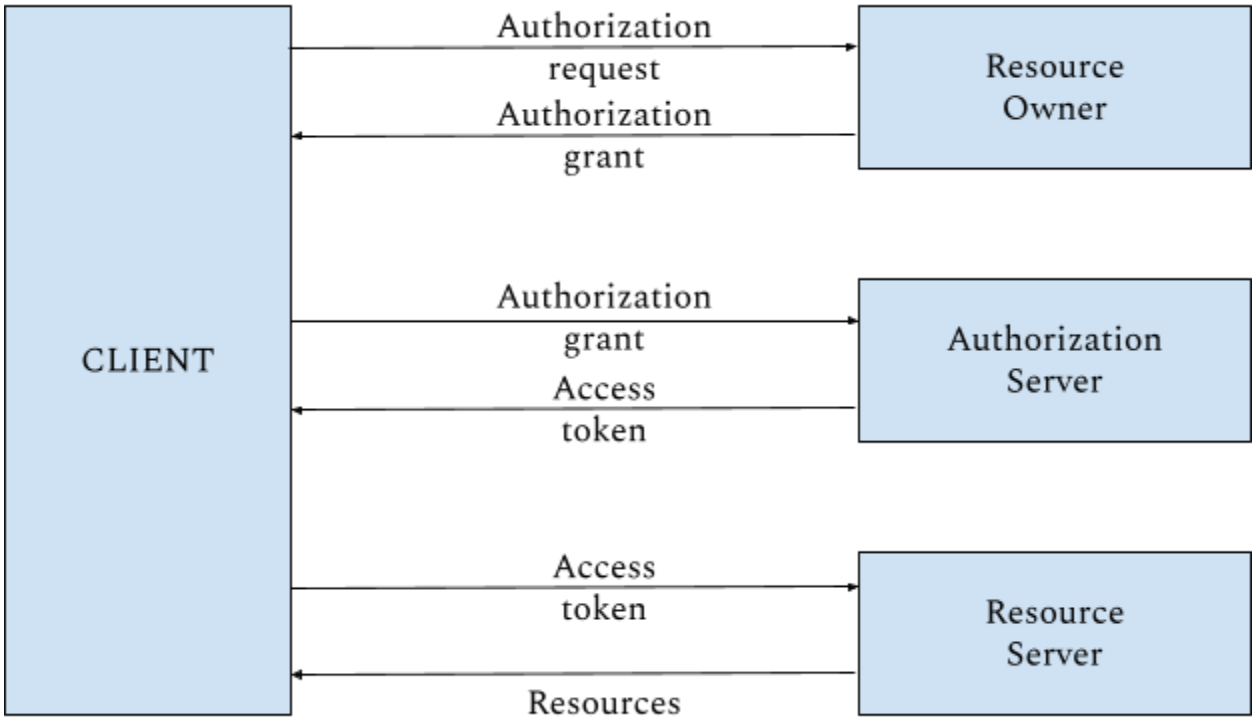
In this the MainUX creates the Authentication Module and then the Authentication Module directly instantiates the Home Screen Module, which in turn directly calls the Cloud or the Dashboard.

We however use the aforementioned design in which the MainUX is the endpoint for all other modules. This way the MainUX has full control and functions as a separate module for the other modules.

6.3 Google OAuth 2.0

We now analyze all the features implemented above. Authentication module does not use any specific data structure as it is more concerned with one user at a time. So our analysis will be based on OAuth performance.

6.3.1 Performance



This is the generic flow of OAuth. As we can see it is a two way process between two endpoints, one always being the client. OAuth 2.0 uses client credentials only when it gains authorization from the user. Rest all API calls are done using only the resulting token from the above mentioned step. This means that API servers never get to know about client credentials and they can validate only using access tokens.

6.3.2 Security

Due to lack of intrinsic security binding, a rogue website can phish a user's valid credentials at the stage of the process where the user is needed to verify themselves to the authorization provider. For instance, suppose a user is utilizing the first service and selects a feature that triggers an OAuth transaction to a second service. It is feasible for

the first website to impersonate the second website, where user authentication is frequently performed. The rogue website can then steal the user's authentication credentials and pretend as though the OAuth transaction was successful.

OAuth expects implementers to use an outside protection protocol like TLS (Transport Layer Security) to provide many of such safeguards, but it is up to the implementers on all sides to make it mandatory. Coders and users should check that OAuth is functioning under TLS protection. Developers can write code to compel the usage of TLS, and users should be informed that TLS is being used whenever they are prompted to submit authentication credentials (just like they should anytime they are entering in credentials).

The protection is for users to ensure that when requested for credentials, they submit them in the authentic second website's domain and avoid dubious first websites. There is no totally safe, generally approved SSO that works on all websites, but we're getting closer with OAuth.

6.3.3 Usability

Using the device browser for OAuth requests rather than an integrated web-view may improve app usability significantly as users only need to sign in to Google once per device, increasing sign-in and authorization flow conversion rates in any app. Using embedded browsers for OAuth requires a user to check in to Google each time, rather than using the device's current logged-in session. Because applications may analyze and alter content in a web-view but not anything visible in the browser, the browser increases security.

7. Summary

Enabling Google Sign-in in our application would not only ensure security from outside our organization, but also inside. No foul activities will go undetected as information of all students/staff will always be stored for any session. This also provides an extremely smooth transition without the need to manually enter personal details every time someone is trying to login. This addition, will therefore, be extremely helpful and user friendly.

8. Conclusion

Once Authentication of any user is done, the next module in charge will be the *HomeScreen Module*. Home Screen Module is responsible for users to either join or host a new meeting. Home screen will transfer the control to the *Dashboard* or the *Cloud* team. This way the Dashboard team can create hash codes, validate IP addresses and store the login information of the current user. Cloud team, on the other hand, can load a previous session right from the Home Screen.
