# Hodophilia

# Software Design

## CSCI-P465/565 (Software Engineering I)

## Project Team

### Akshay Murthy

### Griffin Wall

### Stephen Smith

### Venkata Indu Bhanu Pothineni

## 1. Introduction

### 1.1 System Description

Hodophilia is a web based touring management system which provides tour planning and managing solutions to users at the comfort of their homes. It seeks to provide users with all the tools they will need to select an ideal, personalized tour, book the tour, and follow the tour with minimal oversight during the trip.

### 1.2 Design Evolution

#### 1.2.1 Design Issues

Our team faces a few notable challenges: as compared to a professional team, we collectively have quite a bit less experience in web application development. Added to this is our quite narrow project completion window of time. This means completion time, and thus our ability to understand the technologies we're working with, is of the utmost importance.

Because our project requires a web application, we are required to have an appropriate system design - frontend with vanilla HTML, CSS, & JS (or an extending library/framework) and backend that can sufficiently serve web technologies through HTTP services. Following this, we also need a way of preserving user information on our own systems, thus requiring an appropriate database technology.

#### 1.2.2 Candidate Design Solutions

For our frontend, our most popular consideration was React. We also very briefly considered running a vanilla JS implementation, as well as a few other libraries such as Vue and Svelte. For file structure, we discussed the possibility of using a strict MVC design due to its predictable and strict guidelines. We also considered, for simplicity's sake and to align our structure more closely with a potential React project style, to simply form a distinction between individual components and screens.

For our backend, we most heavily considered Java Spring Boot and Python Flask, though Node.js + Express (using the MERN stack) was also a consideration. For our database, we liked the idea of a NoSQL DB - specifically MongoDB or Firebase - as we noted the object-based organization provided with JSON may be more closely aligned with the types of data we will be storing. Firebase was also alluring as it provided many abstractions for easy access, as well as a cloud storage system removing the hurdle of hosting a database server ourselves. PostgreSQL was also an appealing technology.

#### 1.2.3 Design Solution Rationale

We have decided on using React paired with Java Spring Boot as our frontend and backend. For React, this was due to its popularity, the availability of resources and assistance surrounding it, and Stephen's prior exposure to the library. We believe this option would give the best chance of providing a quality, functional product with all the required features in the shortest amount of time. For Java Spring Boot, it was due to popularity as a backend framework and that it is powered by Java (a language Indu, Griffin, and Akshay are familiar with). We went with PostgreSQL due to its easy integration with Java Spring Boot as well as due to the entire team's familiarity with SQL.

### 1.3 Design Approach

#### 1.3.1 Methods

Our frontend emphasized compartmentalization to divide code into easily reusable components.

Our backend has captured elements of the MVC architectural pattern in order to service the REST API. We use two model classes - Provider and User - to represent their respective systems (Provider: the service that provides the account login). We also use an authentication controller to accept form input from POST calls of login/signup request and prepare a response. We have API’s to handle password reset, and further divided into password reset using email and password reset using security questions. Also, we have allowed the user to use multifactor authentication using Authenticator app. Using the mentioned methods, users can have better control over the profile and security. Likewise, we have few more classes (Search, Search Controller, Search Repository etc.) created to provide JSON response with a list of all places and location details based upon the user’s search.

#### 1.3.2 Standards

We have nested Screens inside the components folder to distinguish smaller components from an assembled page of components. We also take a heavy disposition towards a breadth-first file structure, particularly for the frontend, avoiding deeply nested directories.

We take every precaution to protect user passwords, providing encryption where possible.

Our frontend sticks to Javascript's standard for variable and function names; namely the fixed use of camel case. In addition to this, constant values are always written in uppercase.

#### 1.3.3 Tools

Our primary IDE is Visual Studio Code. We will use this for multiple reasons. First, its excellent git management controls ensures that we will have multiple options for quick, easy version control. Related to this, it provides easy access to multiple types of terminals. This will allow us to quickly view error logs and frontend/backend server status - and give commands to each - all in one place. We will use it alongside recommended Java extensions (notably "Extension Pack for Java") provided via Visual Studio Code's extensions marketplace. This will give us all the support we will need for running, autocompleting, and debugging our Java Spring Boot server. The primary use of Visual Studio Code will, without a doubt, be in managing the front-end due to its fantastic support for web languages. However, we will also be using it to develop our Java Spring Boot application.

We will also be using Node.js and the Node Package Manager to provide and update node modules critical for operating the front-end, such as React.

## 2. System Architecture

### 2.1 System Design

We have chosen to use React in the frontend to communicate with our Java Spring Boot server on the backend with a REST API. This API will retrieve and submit data to/from our database storage with PostgreSQL. We use an encryption algorithm to encrypt/decrypt these passwords before storing in and after retrieving from the database. We then generate a JWT and send it back to the user/client through our REST API between frontend and backend. We are providing list of places and location details on users search in JSON format and then use this JSON response to communicate back to frontend. This search feature doesn’t require user’s authentication. To test these calls before assembling front-end fetch requests, we will use Postman.

System design for user’s profile security:

Further to the login/sign up mechanism provided to the user, we have added features which enable the user to reset password using two means, one is using a valid email used during sign up or by using the answers provided during sign up for the security questions.

The user’s unique token is sent along with a url to his email as a link to reset password. On clicking the link from the email, user can change the password as the user is redirected the password reset page. The token is carried through and the user is identified by the token as we have introduced a field called reset\_token for each user in the DB and each time the user requests for a reset of his password, this token is updated with the one that is used to reset on the front end for validation. The security questions fields are introduced to the DB for each user mandorily to be answered as we store them securely and use these to valid during password reset. These when validated right , will continue to token generation and redirected to password reset page and then the user’s password plus the token is sent to backend where the user is identified by the token and the password is updated.

Multifactor authentication is introduced as a feature and the user has a choice to enable it during sign up, if enabled, the user gets a QR code which is stored on the backend for that user and the user scans the same to add the account to his phone using Authenticator app. This code interacts with Java TOTP manager mechanism to generate unique codes every 30 seconds and used to input into the system for verification. The code entered is verified against the secretImageURI stored as part of the user in the DB which is a random hash generation of the image.

Diagram

Description automatically generated

Diagram

Description automatically generated

### 2.2 External Interfaces

The communication between frontend and backend through REST APIs is included as an external interface where HTTP requests are sent from frontend with a JSON body containing all the necessary information and a response is sent back from backend in JSON format. Backend interfaces with the database through connectors.

Login + Signup routes: Provides a web interface to submit a form, where inputs are converted to a JSON object and submitted via POST to two separate REST APIs. Once data has been submitted, the input will be validated. If validation fails, the user will be prompted to try again, notifying them what errors occurred. The errors will be sent back, alongside an error code, in a JSON response object.

React <Login /> route: This form has a field for **email**, **username**, and **password**. Uses the api/users/login endpoint.

React <Signup /> route: This form has a field for **first name**, **last name**, **email**, **username**, and **password**. Uses the api/users/signup endpoint.

Password reset routes: A HTTP POST request is sent when the user requests for password reset, and further post requests are made to the backend for request through email or reset request through security questions. For MFA, we use Java TOTP and Microsoft Authenticator app for verifying codes obtained through the secret QR code stored for each user.

React<Forgotpassword> route: This page has the user to enter email id .

React<Reset password> route: Once validated, the user is provided with two fields to enter new password and to confirm the new password which is validated.

React<Qrcode> route: this page has the qr code that the user scans if the user wishes to have MFA enabled on the account

React<VerifyCode> route: This page has an input text field to enter for MFA authentication

React<SecurityQuestions> route: This page has input fields that the user enters his answers for security questions.

Search & Explore routes: A HTTP GET requests are sent after performing validation for the search string and a JSON response with all the necessary location details are provided through search APIs.

React<Search /> route: This page has search field for place name. Uses the api/search end point.

React<SearchedCity /> route: This page has category components to display necessary location information like Hotels, Restaurants etc. Uses the api/search/{place} end point.

## 3. Component Design

**Component Name**

User Management: Login/Sign-up component

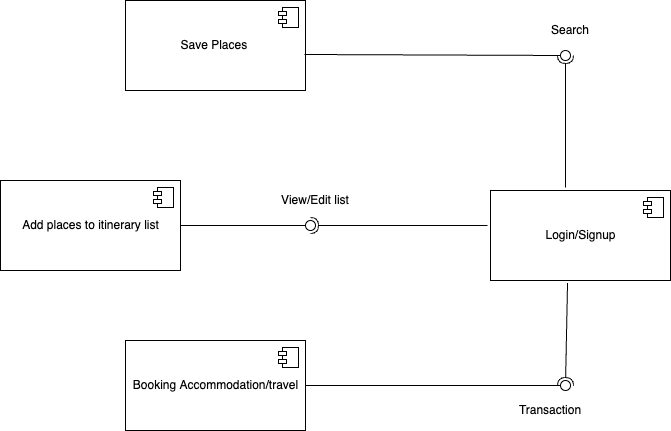
* **Component Description**

This component allows users to login to their account when the user tries to save locations to visit, make a booking or add a place to their itinerary. It provides multiple ways for the user to login to the system through their existing Google/Facebook (using OAuth 2.0) or email accounts and also provides high password encrypted techniques to allow the user to securely login to Hodophilia. This component also allow users to reset password through email, security questions. Also, if the user wishes to , the user can enable MFA on the account and use Authenticator app to have multiple security layers.

* **Responsible Development Team Member**

Akshay Murthy takes the responsibility in both design and implementation of this component.

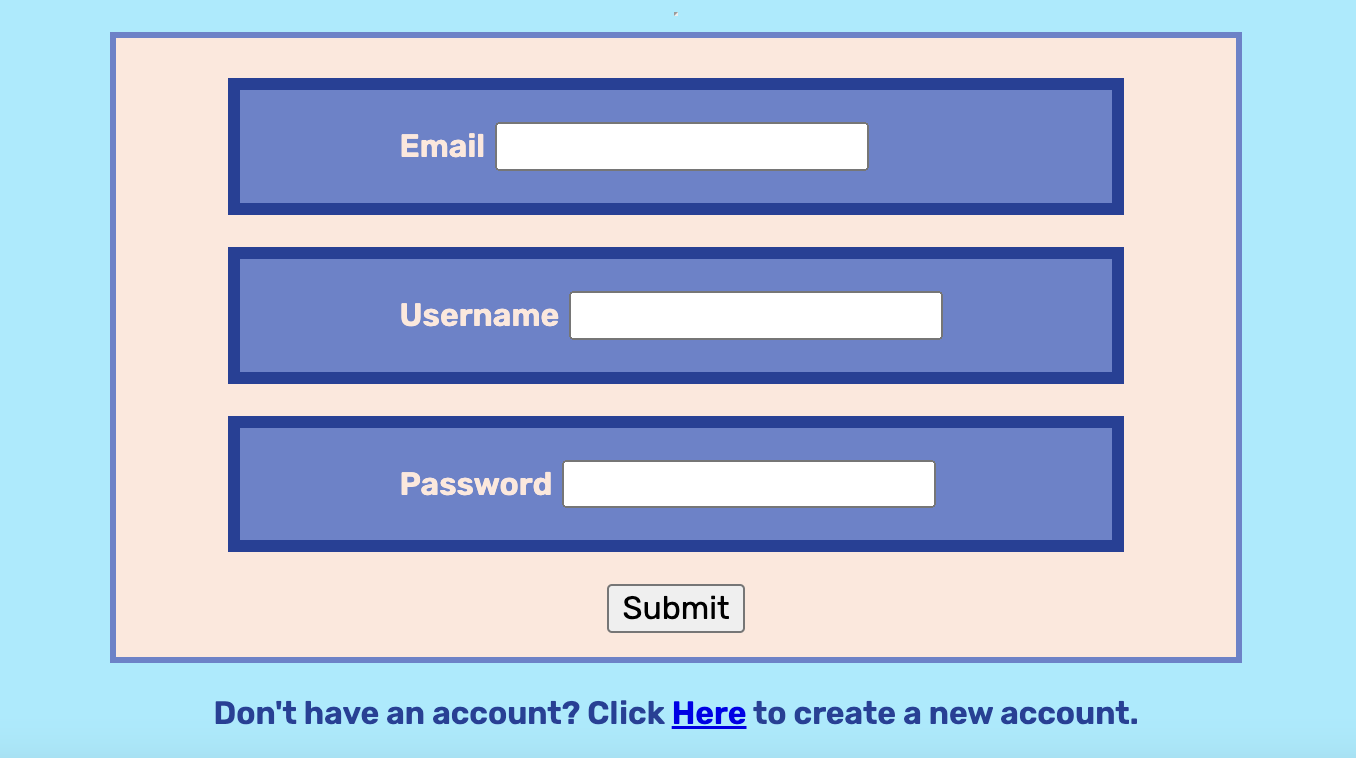
* **Component Diagram**



* **Component User Interface**

The UI of this component includes the login and sign-up pages where the login page allows users to provide Email/Username along with their login password which is further encrypted using Bcrypt hashing method for a safe access and the sign-up page takes the basic inputs from the user and registers an account while storing the user details in the PostgreSQL database.

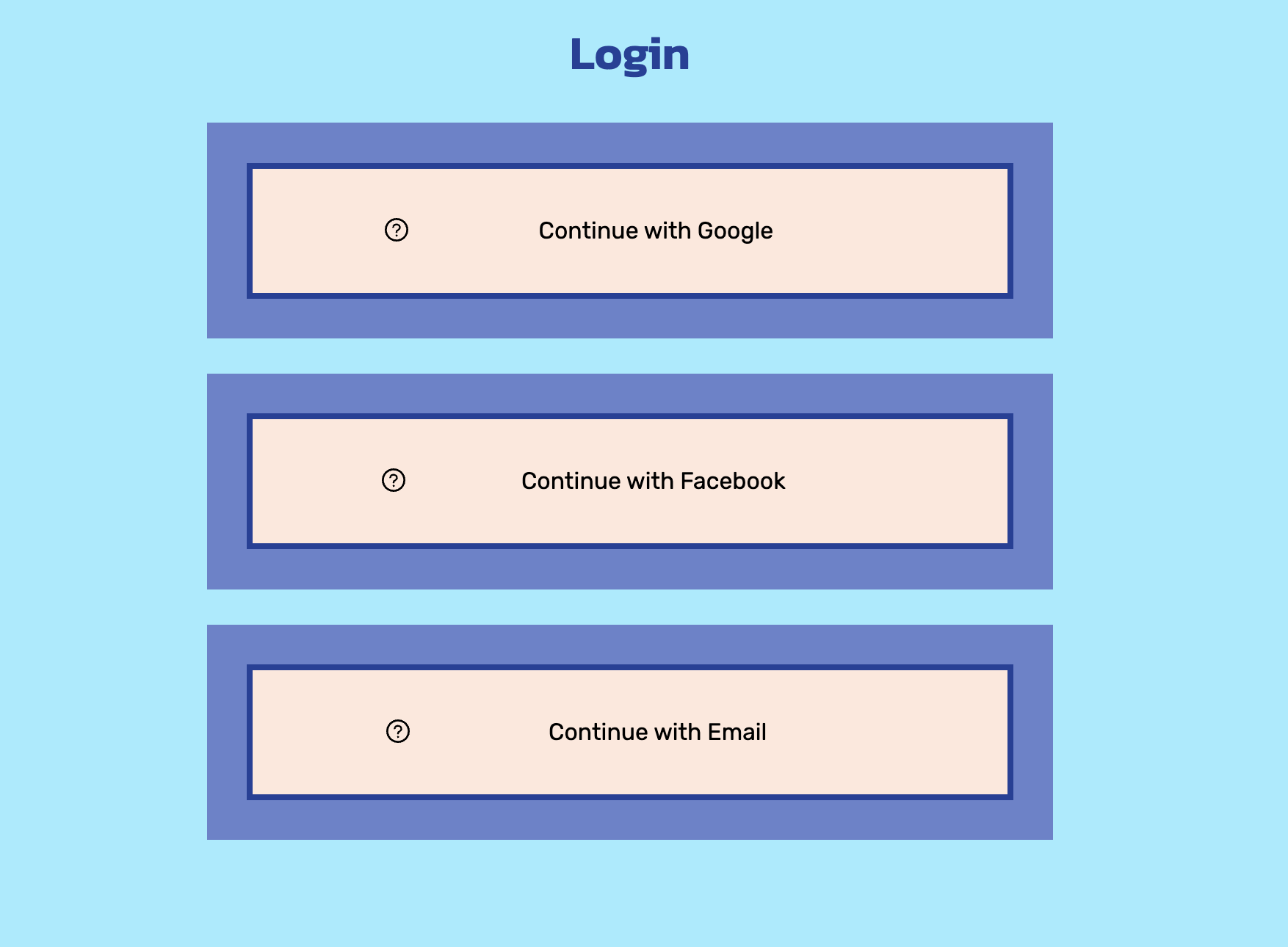
LOGIN PAGE:



SIGN-UP PAGE:



LOGIN WITH GOOGLE/FACEBOOK:



This page allows the users to enter a valid email to recover the account using email reset.

Graphical user interface, application

Description automatically generated

This page allows the users to enter a valid email to recover the account using security questions.

Graphical user interface, application

Description automatically generated

This page allows the users to enter new passwords.

Graphical user interface, application

Description automatically generated

This page allows the user to scan QR code to add account to Microsoft authenticator app for using MFA feature

Qr code

Description automatically generated

This page allows the user to enter code to verify login if MFA is enabled

Graphical user interface, application

Description automatically generated

* **Component Objects**

o The signup page allows the signup API to validate the user information and stores the information under ‘users’ table in PostgreSQL database.

o Login API allows user authentication where the login page provides details required to authenticate the user to the login API and then it validates the user details from database and provides an access token which informs the API that the user is authorized to use the service.

o Both login and signup APIs take user details in json format and throws a status code and response message depending upon the test case.

o Google/Facebook's external API services are used to provide more flexibility to the user to access these services by simply logging in through Google/Facebook if the user already has an existing account in it.

* **Component Interfaces (internal and external)**

Internal interface includes developing Rest APIs for login/signup page and validating user details in backend together with the input form validation of the user to send a valid request. Also, JSON format is used to exchange information in the web application using Rest APIs.

Refer to Section 2.2 for external interfaces.

* **Component Error Handling**

**Error Case 1:** Input Validity Check:

For input validity, we are using Spring boot’s built-in annotations-based methods to check for input errors. We are using @NotBlank annotation, @Size to check for blank user inputs and large size input check. We are using JpaRepository’s existsByUsername to check if the user already exists. JSON message is being sent back with appropriate error status code with a message.

**Error Case 2:** User Authentication Check: User can login through third party providers with the help of OAuth. Spring boot Authentication manager is used to authenticate with username and password, which returns a access token to be stored for a particular session started by the user.

**Component Name**

Search and Explore component

* **Component Description**

This component allows users to search for places and explore more details on the place searched through Hodophilia. The user can either login to their account (through their existing Google/Facebook or email account) or can even explore the website without logging in/signing up to their account. This component further provides information like Hotels, Restaurants, Things to do, Travel forum etc. upon search.

We also provide user an option to search and explore places even before the user login to their account.

* **Responsible Development Team Member**

Stephen Smith takes the responsibility in both design and implementation of this component.

* **Component Diagram**

Graphical user interface, application, Teams

Description automatically generated

* **Component User Interface**

The UI of this component includes the search and explore on search pages where the search page allows users to provide a place where the autocomplete feature is implemented to suggest user with a list of places that matches with the search input given by the user. Upon clicking on a specific location/destination, the user will be able to see a more information on the location searched. Details on Hostels, Things to do, Restaurants etc. are displayed under each category which can further lead to save these place details in users itinerary and make accommodation/travel booking.

SEARCH PAGE:

Graphical user interface

Description automatically generated with medium confidence

EXPLORE ON SEARCH PAGE:

Graphical user interface, website

Description automatically generated

* **Component Objects**
* /forgot\_password API is used to submit request for resetting password.
* /reset\_password API is used to trigger password reset.
* /forgot\_password\_questions is used to submit request for resetting password using security questions
* /verify is used to allow the user to enter the code of MFA to go through authentication.

o The search page allows the (/search) API to retrieve the list of all the places stored under ‘search’ table in PostgreSQL database. This page also provides users with an autocomplete feature for searching and data validation where the search string is validated before making a request to API.

o Explore on search (/search/{place}) API allows user to provide necessary details on a specific location searched through explore page where the user is provided with few necessary categories of the location like Hostels, Things to do, forums etc. based on valid and not-null search request and these details will further be added to itinerary list based on user’s choice of place and things to do.

o Both search and explore on search APIs does not need user’s authentication to get access to these services and provides location details in json format and throws a status code and response message depending upon the test case.

* **Component Interfaces (internal and external)**

Search and explore interfaces externally with PostgreSQL database which has all the location information required on a search. Internal interface includes developing Rest APIs for search/explore pages and validating user input on location to allow the users to send requests with valid and not-null data. Also, JSON format is used to exchange information in the web application using Rest APIs.

Refer to Section 2.2 for external interfaces.

* **Component Error Handling**

**Error Case 1:** Input Validity Check:

For input validity, we are using Spring boot’s built-in annotations-based methods to check for input errors. We are using @NotBlank, @Size annotations to check for blank search inputs and large size input check. Also, @UniqueConstraint annotation is used to define that the column should be unique and duplications to it is not allowed. JSON message is being sent back with appropriate error status code with a message.

1. **Error Case 2:** Search Existence Check: The search page on the frontend allows users to check if the location exists and provides a valid and not-null place to the API request so that any invalid input can be handled even before a request is made to the API service.
2. Error Case 3: User existence check: The forgot password page on the front end allows users to enter any email but checks for a valid user on the backend if the user is valid through API service, /forgot\_password, else triggers a message of invalid user.
3. Error case 4: Token validity check: The reset password feature stores a reset\_token on the DB , which is used at the last step of the process to find the user with the same token from the DB using backend service /reset\_password. This ensures that the token belongs to the user and avoids malicious users to gain access.
4. Error case 5: MFA code validity check: The secret qr code is stored in the backend and the user has multiple codes sent to the app through the randomized hash of the URI and these codes are refreshed every 30 s to ensure the codes are not reused.

## Revision History

|  |  |  |
| --- | --- | --- |
| **Revision** | **Date** | **Change Description** |
| V1 | Oct 4, 2022 | Post Sprint 1 |
| V2 | Oct 18, 2022 | Post Sprint 2 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |