Final Group Report

Group: CI04SI2

Member Name	Student ID	Roles
Loh Jianyang John	1006360	Backend Developer and Tester
Chia Chun Mun	1005934	Frontend Developer - View Subsystem Frontend Tester System Tester
Gizelle Lim Yin Xuan	1006141	Frontend Developer - Homepage Subsystem Sidebar Subsystem Frontend Tester System Tester
Mohammed Ansar Ahmed	1006015	Backend Developer and Tester
Dinh Thao Vy	1006124	Frontend Developer - Form Subsystem Frontend Tester System Tester
Lim Jun Kiat	1005960	Frontend Developer - Login Subsystem Frontend Tester System Tester
Cheng Ee	1004896	Frontend Developer - Account Creation Subsystem Password Management Subsystem Cloud Solutions Engineer, Backend Software Architect, Cloud integration specialist
Wang Jun Long Ryan	1005923	Backend Developer Cloud Solutions Engineer, Backend Software Architect, Cloud Integration Tester,

Requirement

Final Requirements
3 account types: OSL, Fifth Row EXCOs and ROOT
About 1000 characters per section for OSL to comment
Prompt event dates in the EPF when event date is too near the EPF submission date
Account creation functionality
Form validation to ensure fields are filled up properly
A consolidated page to view their forms easily

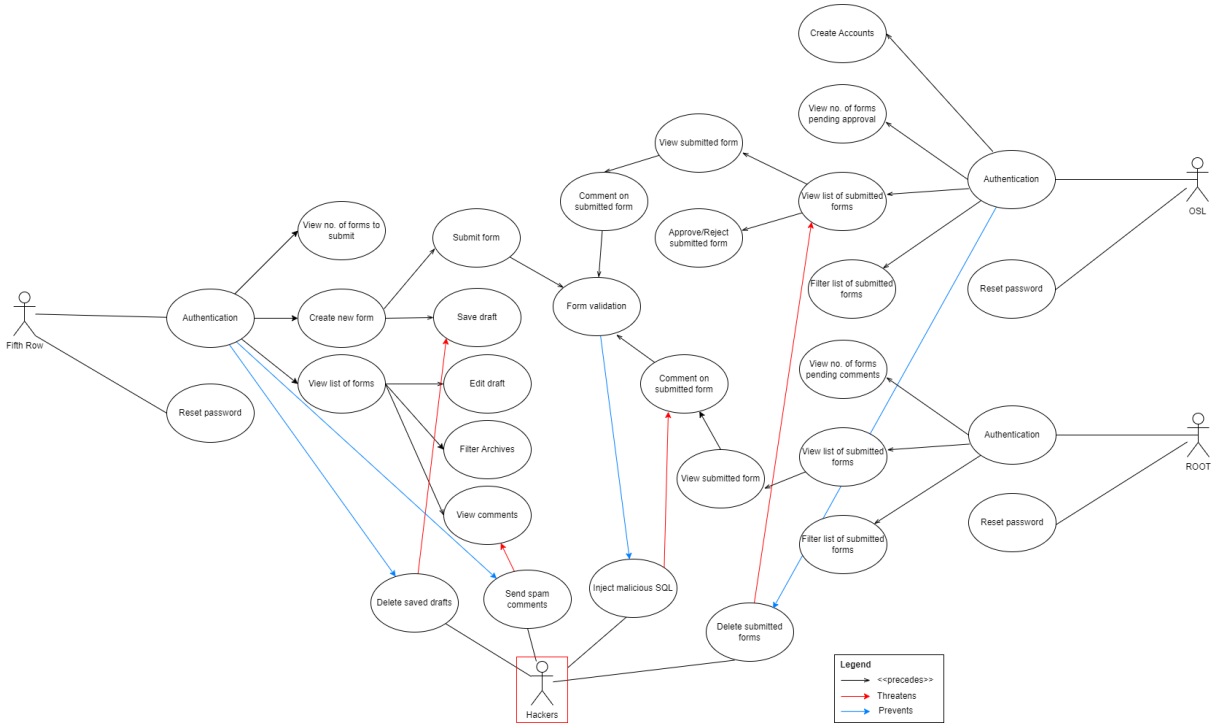
We have dropped the file upload as part of the previous requirements as we have shifted our focus on the AWS implementation instead as part of the agile manifesto.

As for the notification system, we implemented the email service using AWS simple email service and we were using ses sandbox. The sandbox is to prevent people from spamming. The sandbox ses require us to manually verify accounts or own the SUTD domain email. We then tried to request to be removed from the sandbox but aws rejected the request. Thus we were unable to proceed with the notification system.

OSL-Fifth Row App is a web application that aims to improve the event proposal form submission and management process for OSL, Fifth-Row EXCOs and ROOT users.
Our use case diagram showcases our web application system's overall functions and scope: OSL-Fifth Row App. The app has 7 main use cases that users can make use of depending on their roles and the permissions they have:

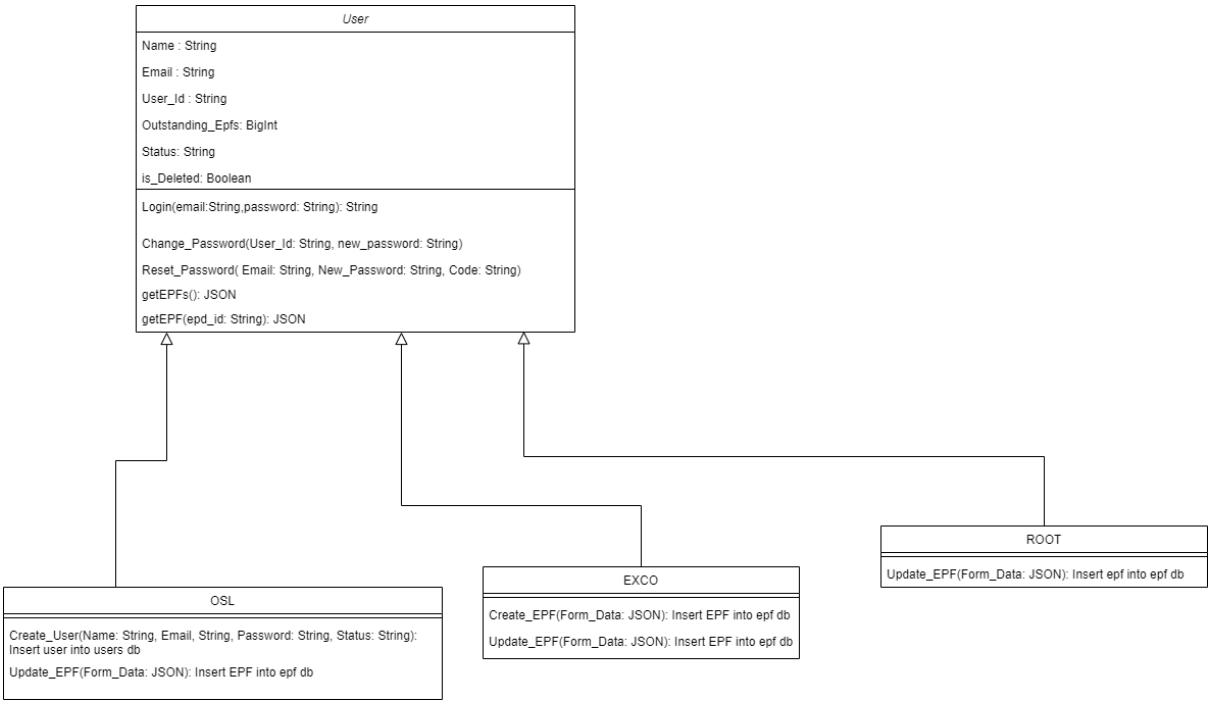
1) Create an Account (OSL)
2) Login (Fifth Row EXCOs, OSL, and ROOT)
3) Reset User Password (Fifth Row EXCOs, OSL, and ROOT)
4) View Form Archive (Fifth Row EXCOs, OSL, and ROOT)
5) Comment on Forms (OSL and ROOT)
6) Approve Forms (OSL)
7) Submit Forms (EXCOs)

The relationship between each use case is demonstrated in the use case diagram below:



Design

Class Diagram



Database Design

EPF

```
epf_id SERIAL PRIMARY KEY,
status VARCHAR,
exco_user_id VARCHAR,
date_created TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

a_name VARCHAR,
a_student_id INT,
a_organisation VARCHAR,
a_contact_number INT,
a_email VARCHAR,
a_comments_osl VARCHAR,
a_comments_root VARCHAR,

b_event_name VARCHAR,
b_target_audience VARCHAR,
b_event_schedule VARCHAR,
b_expected_turnout INT,
b_event_objective VARCHAR,
b_comments_osl VARCHAR,
b_comments_root VARCHAR,

c1_date TEXT[],
c1_time TEXT[],
c1_activity_and_description TEXT[],
c1_venue TEXT[],

c2_date TEXT[],
c2_time TEXT[],
c2_activity_and_description TEXT[],
c2_venue TEXT[],

c3_date TEXT[],
c3_time TEXT[],
c3_activity_and_description TEXT[],
c3_venue TEXT[],

c3_cleanup_date TEXT[],
c3_cleanup_time TEXT[],
c3_cleanup_activity_and_description TEXT[],
c3_cleanup_venue TEXT[],
c_comments_osl VARCHAR,
c_comments_root VARCHAR,

d1a_club_income_fund DECIMAL(10,2),
d1a_osl_seed_fund DECIMAL(10,2),
d1a_donation DECIMAL(10,2),
d1b_revenue DECIMAL(10,2),
```

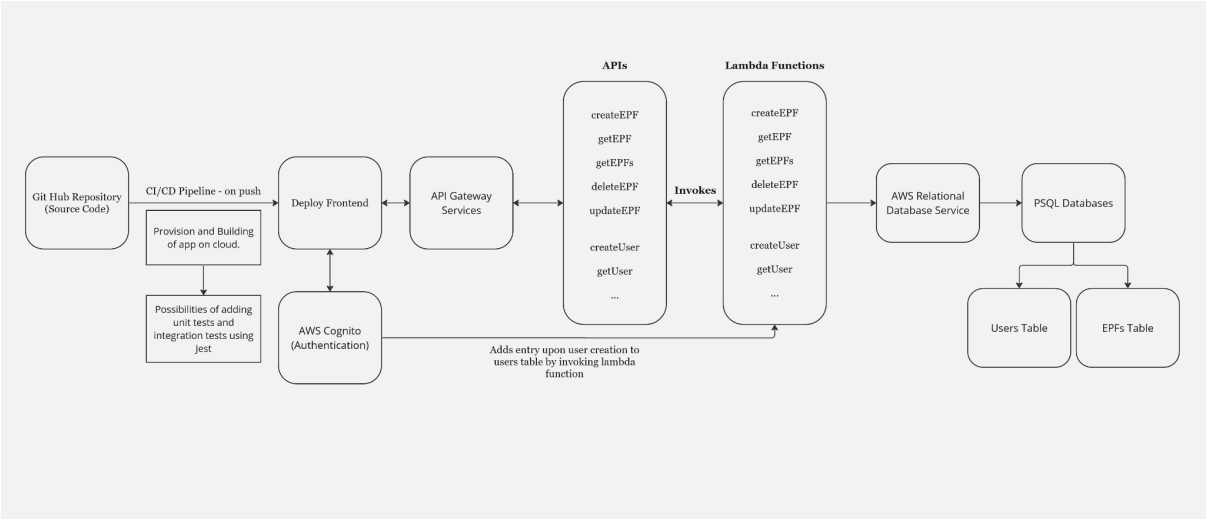
```
    d1b_donation_or_scholarship DECIMAL(10,2),
    d1b_total_source_of_funds DECIMAL(10,2),

    d11_items_goods_services TEXT[],
    d11_price TEXT[],
    d11_quantity TEXT[],
    d11_amount TEXT[],
    d11_total_revenue DECIMAL(10,2),

    d2_items TEXT[],
    d2_reason_for_purchase TEXT[],
    d2_venue TEXT[],
    d2_total_expenditure DECIMAL(10,2),
    d_comments_osl VARCHAR,
    d_comments_root VARCHAR,

    e_personal_data INT,
    e_comments_osl VARCHAR,
    e_comments_root VARCHAR,

    f_name TEXT[],
    f_student_id TEXT[],
    f_position TEXT[],
    f_comments_osl VARCHAR,
    f_comments_root VARCHAR,

    g_1_1 VARCHAR,
    g_1_2 VARCHAR,
    g_1_3 VARCHAR,

    g_2_1 VARCHAR,
    g_2_2 VARCHAR,
    g_2_3 VARCHAR,

    g_3_1 VARCHAR,
    g_3_2 VARCHAR,
    g_3_3 VARCHAR,

    g_4_1 VARCHAR,
    g_4_2 VARCHAR,
    g_4_3 VARCHAR,

    g_5_1 VARCHAR,
    g_5_2 VARCHAR,
    g_5_3 VARCHAR,

    g_6_1 VARCHAR,
    g_6_2 VARCHAR,
```

```
    g_6_3 VARCHAR,

    g_7_1 VARCHAR,
    g_7_2 VARCHAR,
    g_7_3 VARCHAR,
    g_comments_osl VARCHAR,
    g_comments_root VARCHAR,

    is_deleted BOOLEAN DEFAULT false
```

User

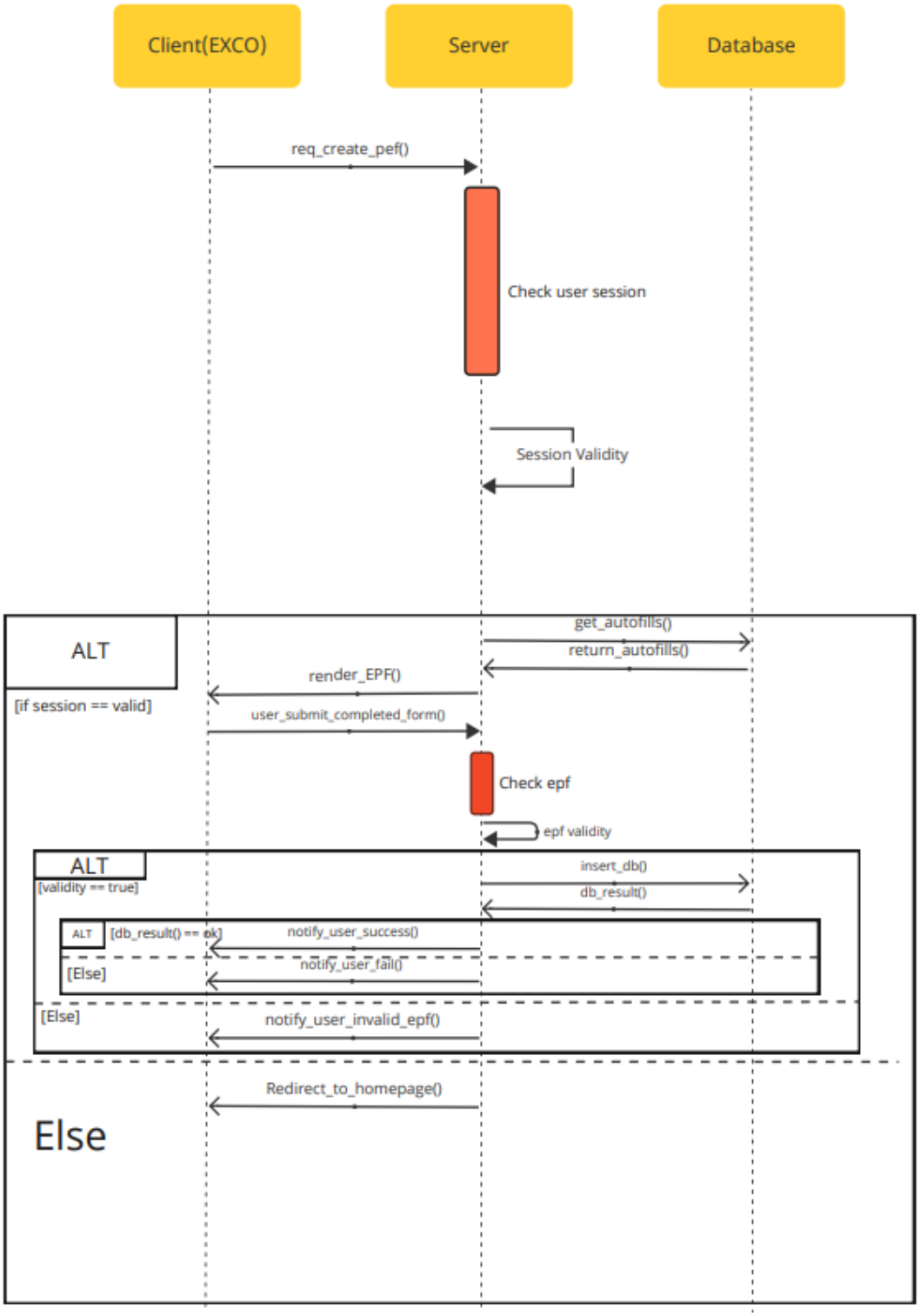
```
    user_id VARCHAR PRIMARY KEY NOT NULL,
    name VARCHAR NOT NULL,
    email VARCHAR NOT NULL,
    user_type VARCHAR NOT NULL,
    outstanding_epf INT,
    is_deleted BOOLEAN DEFAULT false
```

AWS Architecture

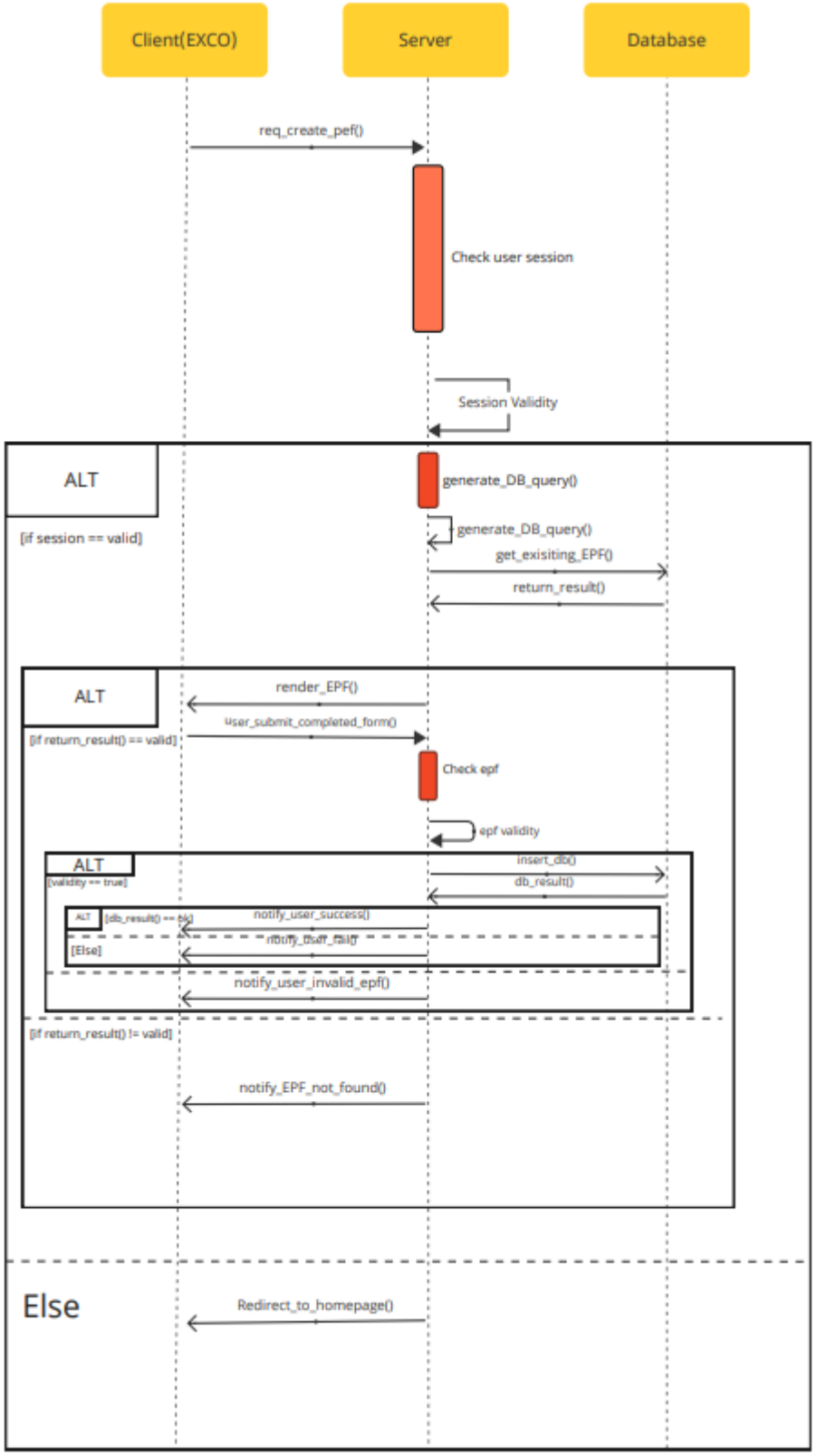


Sequence Diagrams

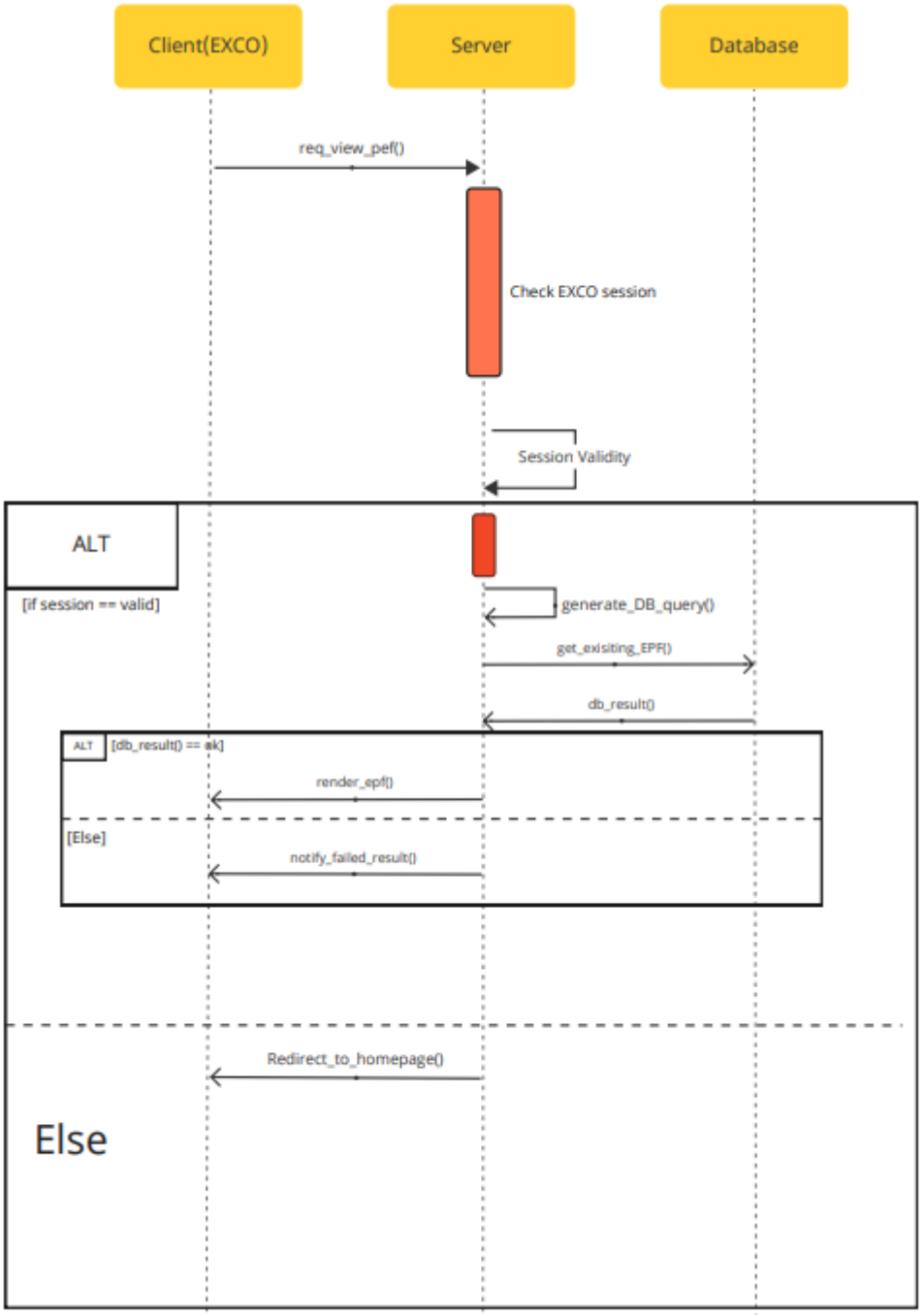
1. EXCO wants to create new EPF



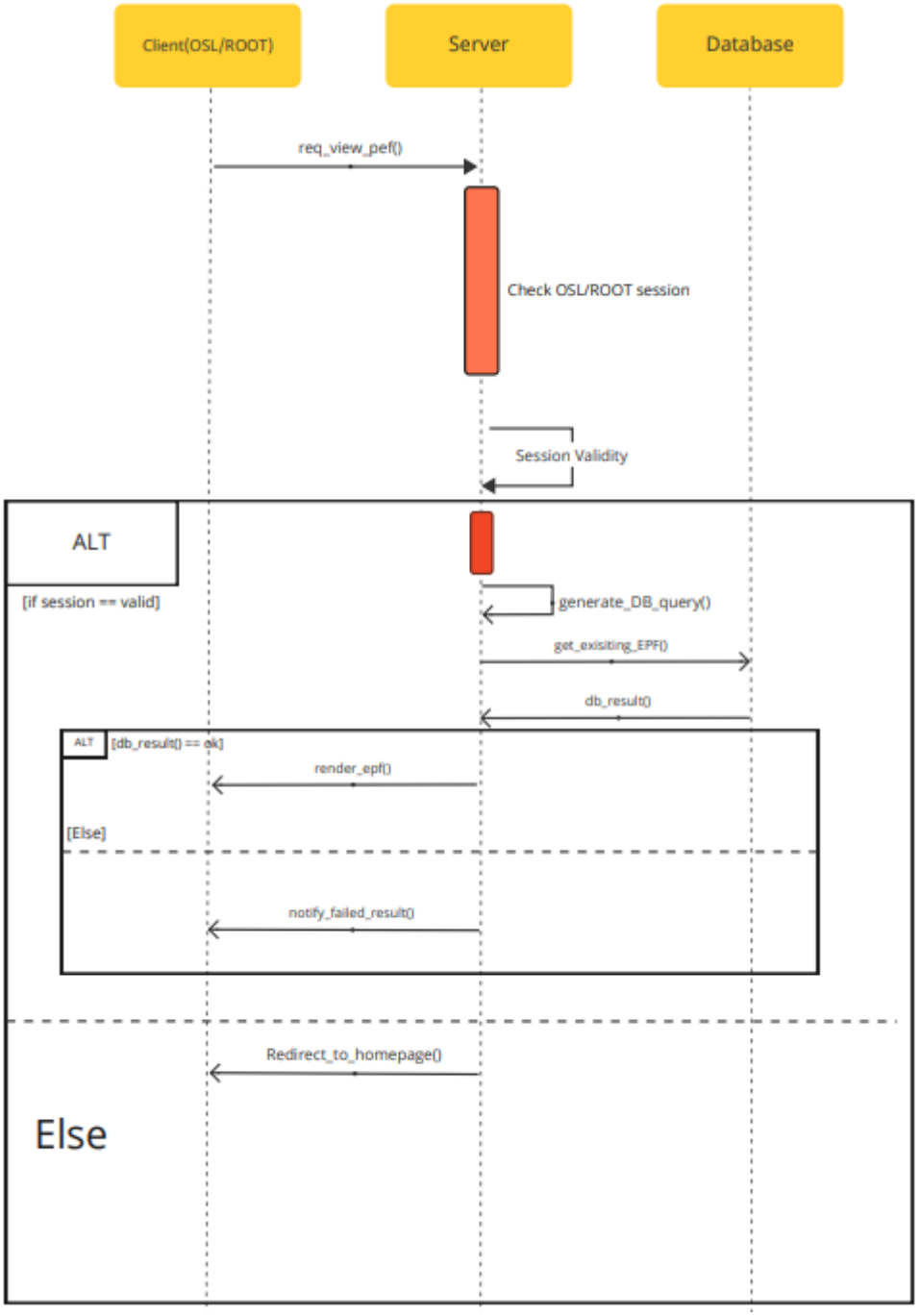
2. EXCO wants to update existing EPF



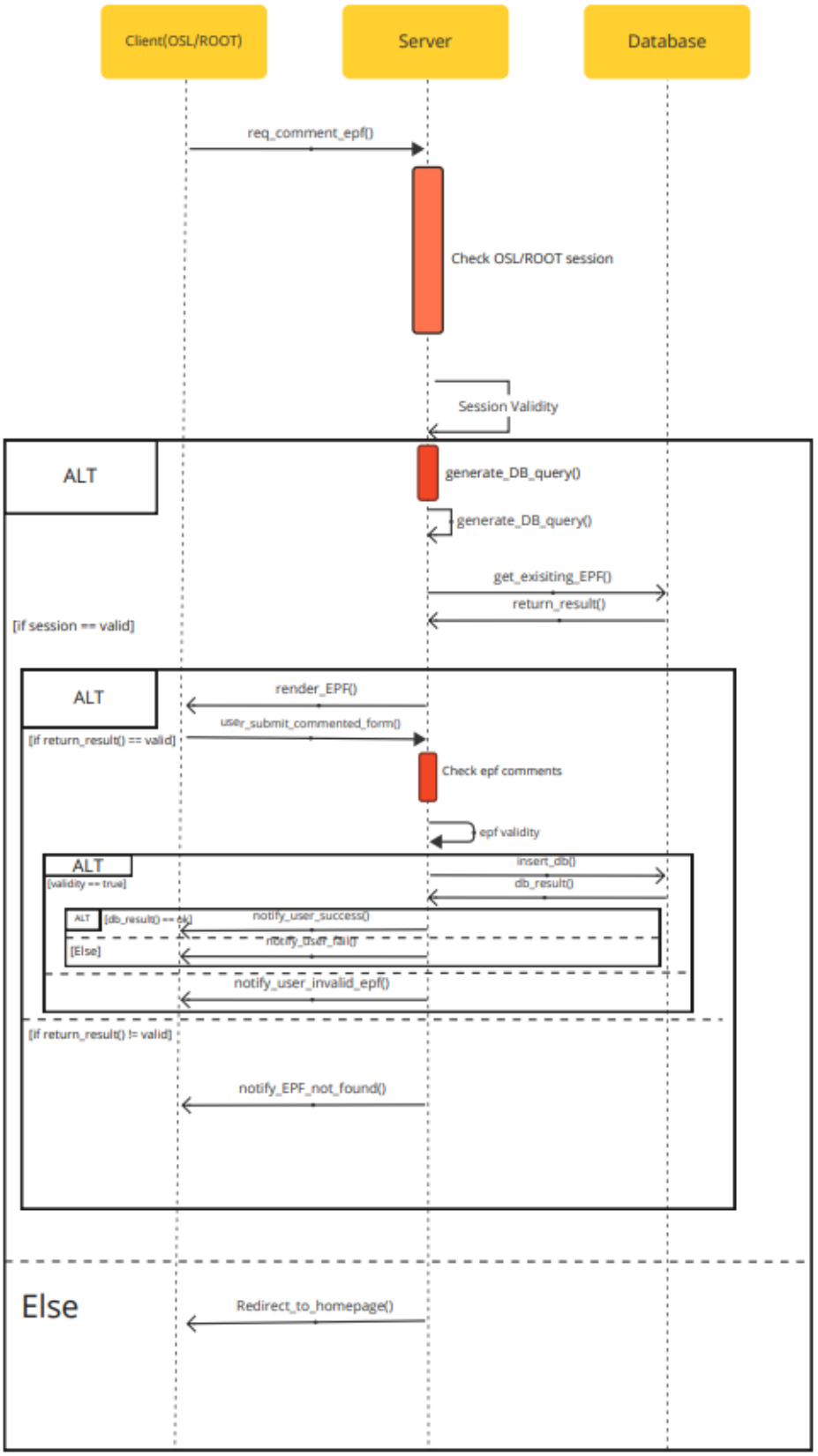
3. EXCO wants to view existing EPF



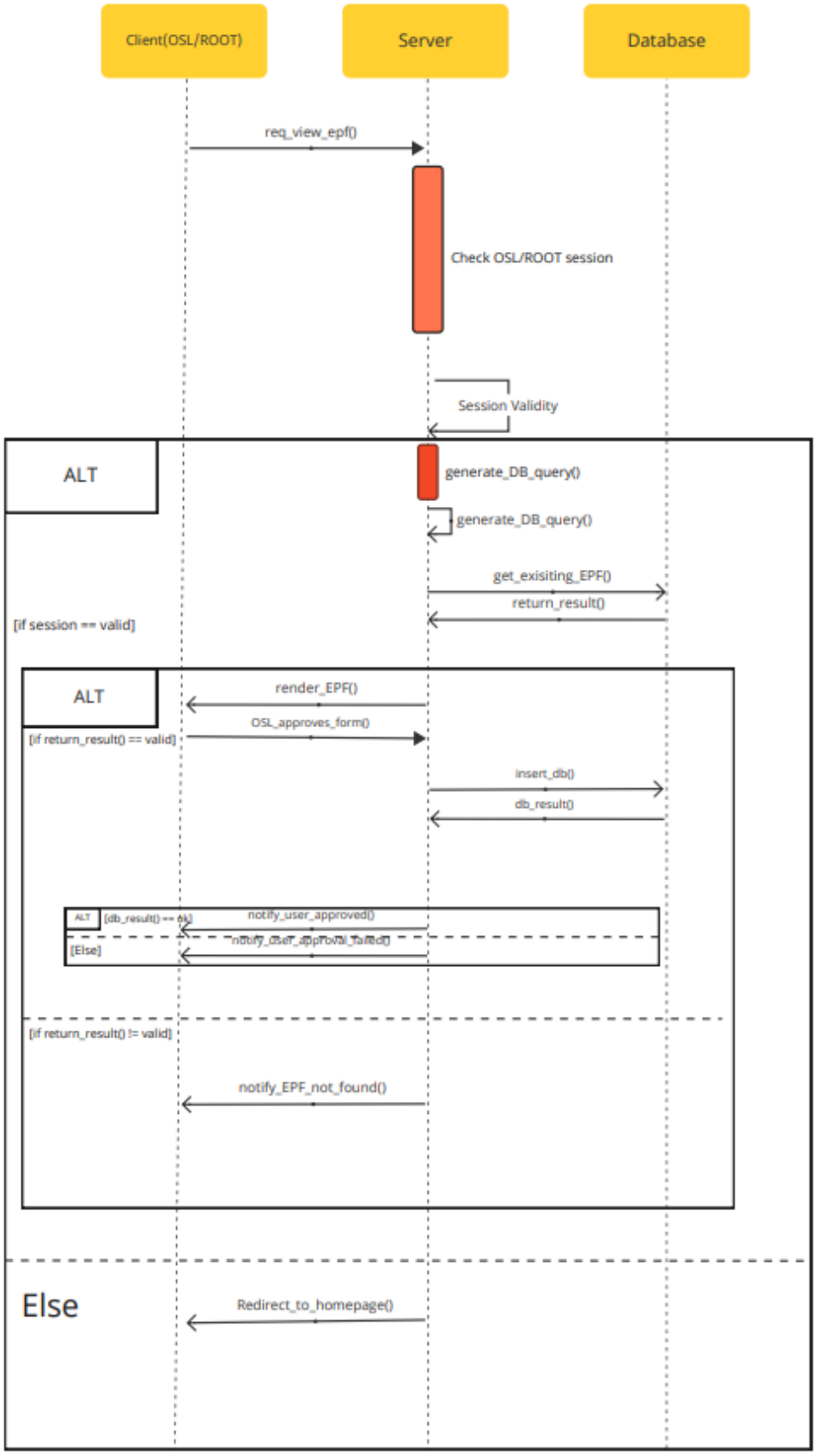
4. OSL/ROOT wants to view EPFs



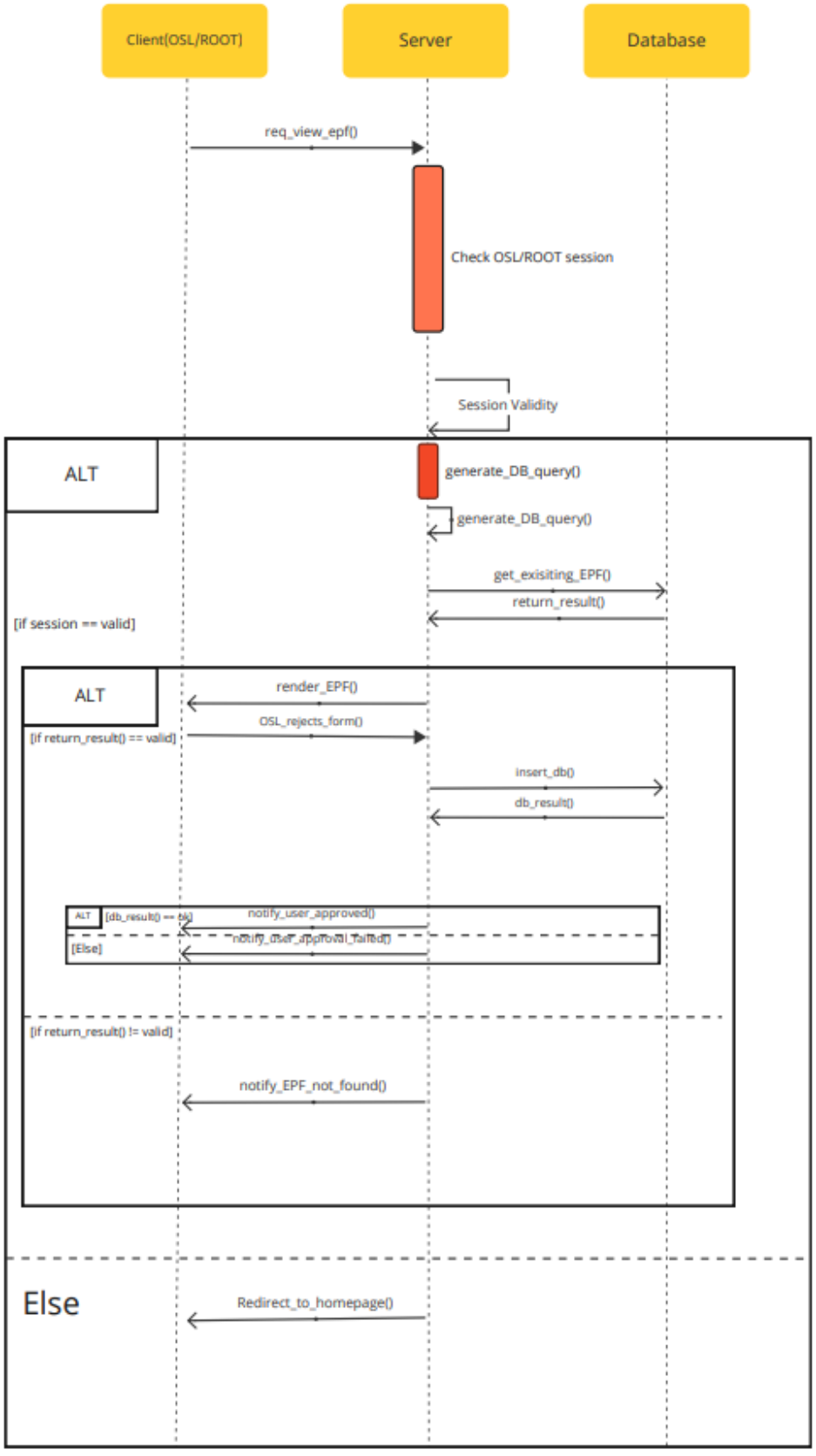
5. OSL/ROOT wants to leave comments in EPF



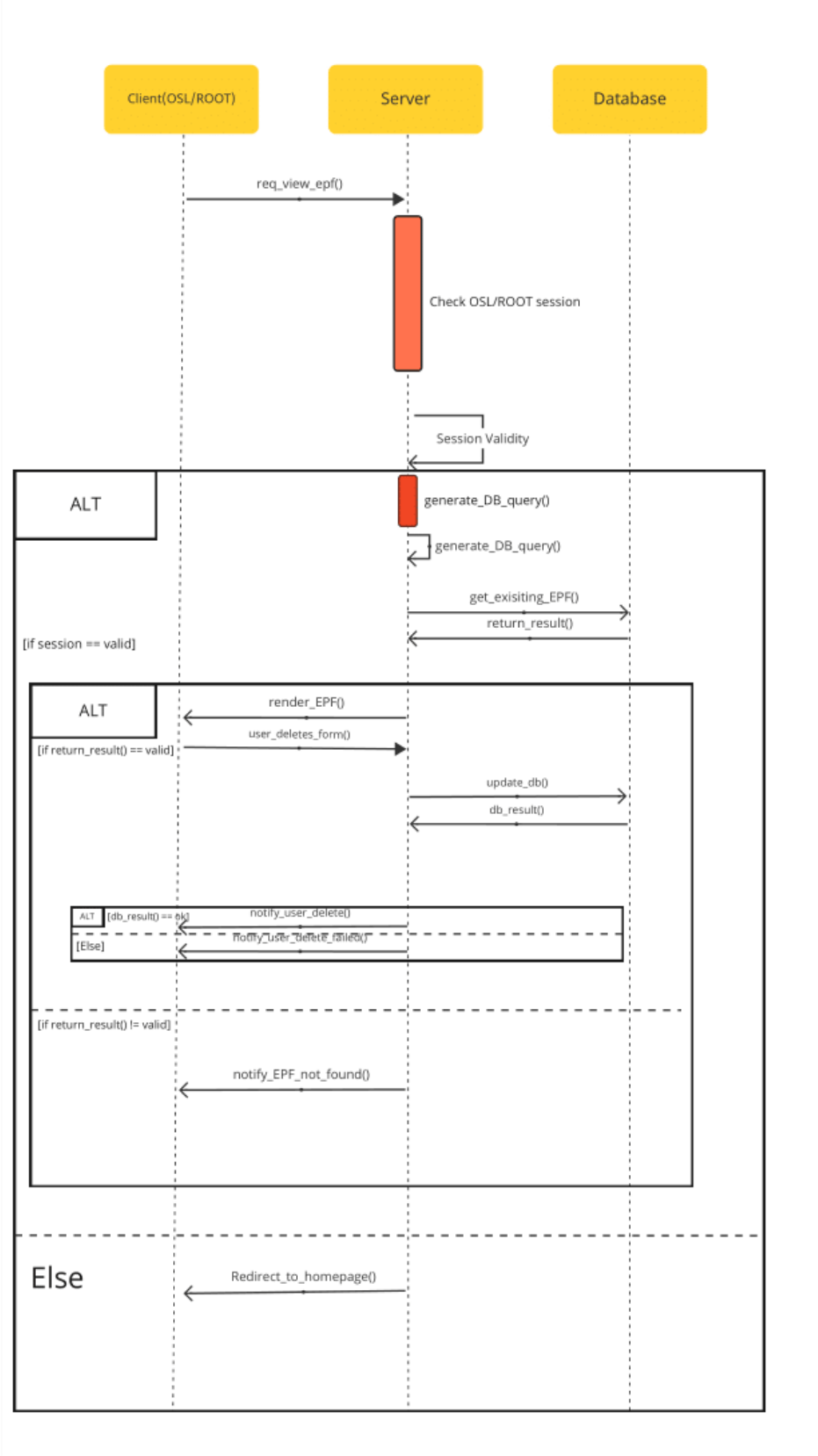
6. OSL wants to approve EPF



7. OSL wants to reject EPF



8. OSL wants to delete EPF



Implementation Challenges

Algorithmic Challenges

Since the app mostly involves just data validation, sorting, filtering, and storage there were minimal algorithmic challenges faced, both on the frontend and backends codebase. However, as more complicated features get onboarded in the future there will be a greater need for complex algorithms to be utilized for implementing those features. For example one of our future features to be implemented is room booking and allocation in which we can use a modified version of the knapsack algorithm to efficiently allocate booking of venues for events.

Engineering Challenges

Backend

1. Invalid Query: Using string formatting directly to build the DB query resulted in errors as string data contains SQL keywords additionally DB is vulnerable to SQL injection or invalid Query

- Solution: Use the parameterized query

2. Race condition: Since async programming allows for tasks to be scheduled concurrently and Postgres DB allows for multithreading, we faced read/write tears for data, especially during the testing phase.

- Solution: Serialization and/or Read Committed isolation and Row-Locking for DB queries

3. Deadlock: Since we introduced locks we also ran into the issue of deadlocks

- Solution: Deadlock detection(provided by posture package) and resolution (A release and retry mechanism implemented in our server)

4. Migration to Amazon Web Services (AWS): The cloud services offered by AWS often demand various code changes that force both the front end and the back end to adapt

accordingly. This is especially evident with the use of API Gateway, Amplify, and Lambda functions.

- Solution: Increased communication between the backend and frontend team to convey what is required from both ends as well as reading the documentation of the tools (AWS services such as AWS Cognito).

Frontend

1. Form - Table: The tables in an EPF form are of different shapes and different minimum rows required, with some having row names and static row values, while some needing options to add new rows/delete existing rows. Both of these can occur in the same table, which means the ‘Table’ component, which only has basic functionalities from external libraries (React/MUI/Vue) cannot be used.

- Solution: With consideration that the app might expand to other forms aside from EPF, build custom components for EPF tables allowing props that specify validation, required, dynamic-ness (add new row/delete row), etc to maximise the modularity and reusability of the code. The custom components use the ‘Grid’ component form MUI to keep leveraging on MUI’s clean css.

2. Form - Load EPF: An EPF form loaded from the server might have some fields disabled depending on the status of the form and the user’s edit permissions (e.g. all input fields such as Event Name are disabled for OSL users).. React Hooks, and in particular, *useForm()* from ‘react-hook-form’ must be called in a React functional component, but asynchronous API calls can only be placed in the lifecycle method *useEffect()*. WIth the new information from the API call consumption in *useEffect()*, fields that must be set to disabled have to be re-registered again, but this will trigger the component to re-render, which in turn trigger the *useEffect()* method again, landing the component in an infinite re-render cycle.

- Solution: Create a wrapper component that initially renders a spinner. Once the API call finishes, it will then render the original EPF form component with the correct disabled/enabled properties.

Testing Challenges

Backend

1. JSON data for testing Event Proposal Forms (EPFs) are very long
   To work around the problem of having several long JSON data inputs in our test suites, we placed the test JSONs into a separate folder in the same directory level as the test suite. Which we shared on

   The numerous fields of EPFs with various data structures also posed a problem for generating test cases. We applied the concept of equivalence class testing to generate fewer test cases that will still provide a broad, comprehensive code test coverage. In the end, we achieved a comprehensive code test coverage of 90.22%.
2. Jest runs test cases concurrently and although the main server code was designed to handle concurrency, the test scripts written were not as they required to set up and tear down for each test case and values such as EPF count and epd_id are interdependent and dynamic. This makes making test cases, especially making test cases that test for exact matches, significantly more difficult. We hence decided to run the test cases InBand (test suites are run sequentially one after the other)

Frontend

First of all, there would be too many test cases available to test for. We can individually test for every single button and field in the forms. In a short period, to cover all the different test case categories, we have to be selective. Thus, we applied the concept of equivalence testing and avoided the repetitive test cases.

Initially, we also faced issues to mock the data in jest and selenium. It kept prompting us errors regarding packages etc. As it was the first time using jest for all of us, we were stuck and went on exploring how to fix the error for 2 days before coming together to sit down and tackle the problem with the information each of us have gathered.

When using Selenium testing, some components of the webpage render quicker than the rest, thus some of the code in Selenium will execute first before waiting for the browser. Thus, this causes the test suite to fail. To counter this, we either await till the element appears before executing the action (e.g. click or enter keys) or we can add a delay between the codes to let the webpage have enough time to render before continuing executing.

We also want our tests to execute successfully on any computer, not just the computer of the developer who wrote the test. As such, instead of hard-coding the values that a form must take e.g. form with id = 2 has the event name “Test Event”, we modified the test setup such that it will create a new form with id = x, with a unique event name using the API createEPF. The test itself then mock-renders the corresponding React component, loads the form with id = x, and checks that the UI is correctly displaying the event name. This is also in line with Jest’s spirit - testing the component rather than the page loaded with the component.

Testing

Backend

Link to backend Test case tables:
https://sutdapac-my.sharepoint.com/:x:/g/personal/mohammed_ansar_mymail_sutd_edu_sg/EfoC2AfGgklPo0kR4Ehka38BPaW5QrVLExjFM7EO_PQn9g?e=Nt2DVQ

The test cases for individual backend API functions were generated using equivalence class testing to come up with fewer, but comprehensive test cases, inclusive of positive and negative testing, that provided a wide code coverage (90.22%).

Testing was done using Jest. Unit testing was performed where each API function was tested. Integration testing was also done by testing the API function's interaction with the database.

The backend API functions that were tested include the following:
EPF

- createEPF
- getEPF
- getEPFs
- getEXCOEPFs
- updateEPF
- deleteEPF
- countOutstandingEpf

User

- createUser
- getUser
- getUsers
- updateUser
- deleteUser

Our unit and integration testing process follows the general recipe:

1. Truncate the users and EPFs database tables in before() or beforeEach() (test case dependent)
2. Set up the necessary prerequisites for test cases in beforeAll() or beforeEach() (test case dependent). This includes:
    - Fifth-Row EXCO user accounts that EPFs need to be linked to
    - EPFs
3. In each test case
    - Retrieve JSON data for DB operation and positive/negative checking (can be found in the testing document), if necessary
    - Execute the test
    - Perform the positive/negative checking

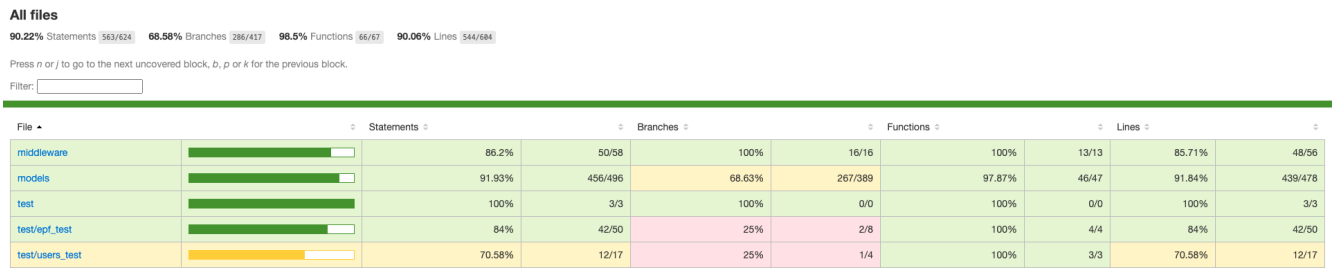
The backend API middleware functions were tested using fuzzy testing, in particular, fuzzing the low-level request - generating random bytes as HTTP requests to test the robustness of the API service. The backend API middleware functions that were tested included the following:

- createEPFmiddleware
- getEPFmiddleware
- updateEPFmiddleware
- deleteEPFmiddleware

Our fuzzy testing process follows the general recipe:

1. Testing positive behaviour using a valid request body (can be found in the testing document)
2. Testing negative behaviour where random bytes, incomplete, incorrect request bodies, or params are sent (can be found in the testing document)

The final results of testing locally on the backend are as follows:



All files

90.22% Statements `563/624` **68.58%** Branches `286/417` **98.5%** Functions `66/67` **90.06%** Lines `544/604`

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File		Statements		Branches		Functions		Lines	
middleware		86.2%	50/58	100%	16/16	100%	13/13	85.71%	48/56
models		91.93%	456/496	68.63%	267/389	97.87%	46/47	91.84%	439/478
test		100%	3/3	100%	0/0	100%	0/0	100%	3/3
test/epf_test		84%	42/50	25%	2/8	100%	4/4	84%	42/50
test/users_test		70.58%	12/17	25%	1/4	100%	3/3	70.58%	12/17

Overall Test Coverage: 90.22%

To help achieve more modularity, a database utilities file was used to create and release connections across the test cases.

Backend AWS

Most of the test cases done locally (backend API middleware functions not included) in the backend were migrated over to AWS as part of our Agile Manifesto.

The testing tools (Jest) and the recipes for the testing process are the same. The main difference in testing with AWS lies in changing the code to use the local database, to the database hosted on AWS via lambda functions.

Frontend

Link to frontend Test case tables:
https://docs.google.com/spreadsheets/d/11ocfVnybG_ygzhj-RaptEct_jUMCW6TEdwZBHasmIwg/edit#gid=23257494

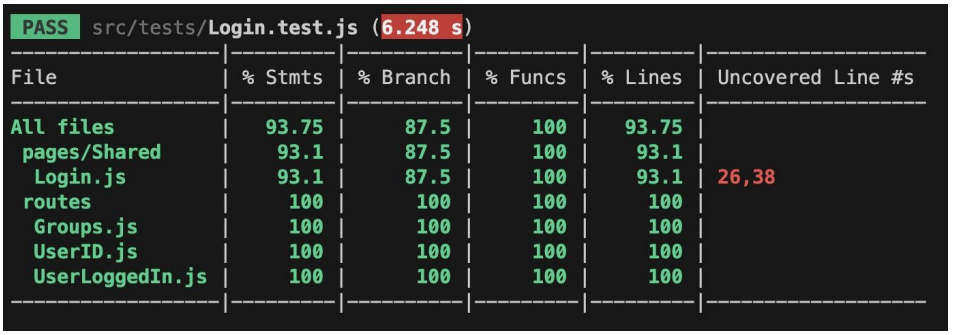
On the frontend's side, unit and system testing was conducted to test the functionality of each frontend component. As the main purpose of the app is to allow users to fill in forms, there naturally were many fields and combinations that we could test. However, due to the lack of time and manpower, we could only perform a limited amount of testing. Hence, we made use of equivalence class testing to come up with fewer, but comprehensive test cases that would simulate users' common responses and mistakes such that the main usage of the form was tested.

Testing was done using Jest and Selenium. Unit testing was done to test the various functions of each frontend component. System testing was also performed to test the essential functionality of each frontend component, the integration between the different frontend components, as well as the integration between frontend and backend components.

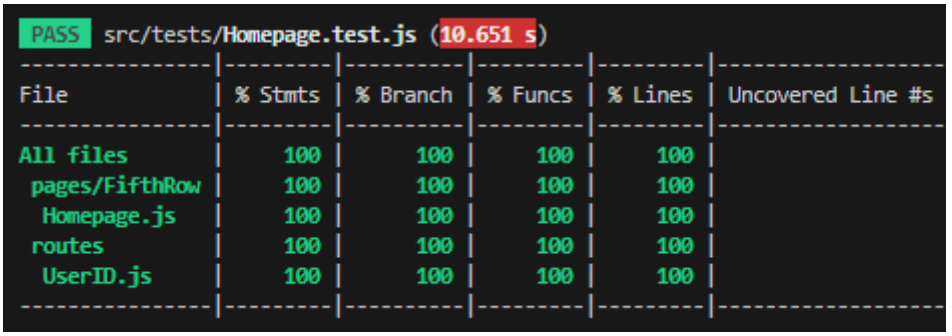
Our unit testing covers the five main components of the app which include login, homepage, sidebar, epf submit, and epf view. Comprehensive tests were done to ensure that there

exists proper rendering and functionality of each component. Testing was also done to simulate API calls from the backend to display the relevant information.

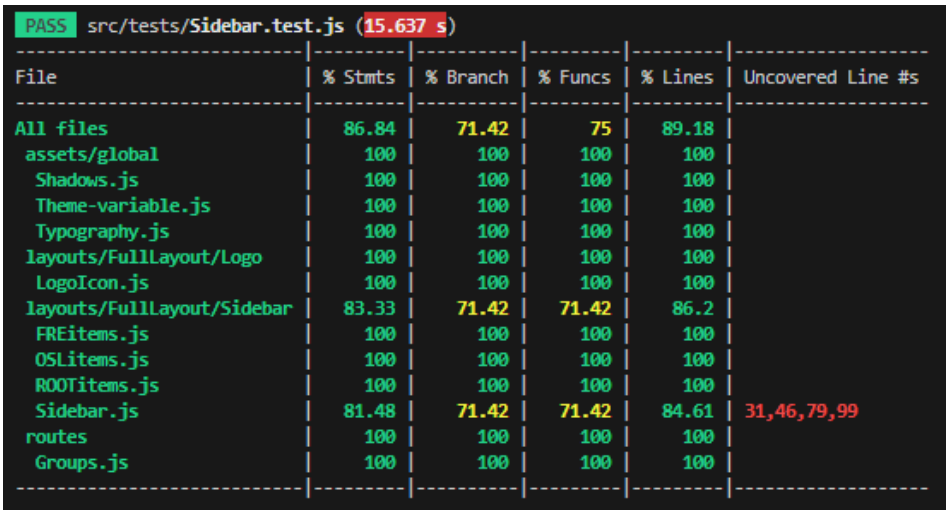
As some of our unit tests were done using Selenium, we were unable to generate a test report for them. Hence, what we have are the unit tests that were done using Jest, the test report for each of these components can be seen here:



```
PASS src/tests/Login.test.js (6.248 s)
-------------------|---------|----------|---------|---------|-------------------
File               | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-------------------|---------|----------|---------|---------|-------------------
All files          |   93.75 |     87.5 |     100 |   93.75 |
 pages/Shared      |    93.1 |     87.5 |     100 |    93.1 |
  Login.js         |    93.1 |     87.5 |     100 |    93.1 | 26,38
 routes            |     100 |      100 |     100 |     100 |
  Groups.js        |     100 |      100 |     100 |     100 |
  UserID.js        |     100 |      100 |     100 |     100 |
  UserLoggedIn.js  |     100 |      100 |     100 |     100 |
-------------------|---------|----------|---------|---------|-------------------
```



```
PASS src/tests/Homepage.test.js (10.651 s)
----------------|---------|----------|---------|---------|-------------------
File            | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------|---------|----------|---------|---------|-------------------
All files       |     100 |      100 |     100 |     100 |
 pages/FifthRow |     100 |      100 |     100 |     100 |
  Homepage.js   |     100 |      100 |     100 |     100 |
 routes         |     100 |      100 |     100 |     100 |
  UserID.js     |     100 |      100 |     100 |     100 |
----------------|---------|----------|---------|---------|-------------------
```



```
PASS src/tests/Sidebar.test.js (15.637 s)
---------------------------|---------|----------|---------|---------|-------------------
File                       | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------------------|---------|----------|---------|---------|-------------------
All files                  |   86.84 |    71.42 |      75 |   89.18 |
 assets/global             |     100 |      100 |     100 |     100 |
  Shadows.js               |     100 |      100 |     100 |     100 |
  Theme-variable.js        |     100 |      100 |     100 |     100 |
  Typography.js            |     100 |      100 |     100 |     100 |
 layouts/FullLayout/Logo   |     100 |      100 |     100 |     100 |
  LogoIcon.js              |     100 |      100 |     100 |     100 |
 layouts/FullLayout/Sidebar |   83.33 |    71.42 |   71.42 |    86.2 |
  FREitems.js              |     100 |      100 |     100 |     100 |
  OSLitems.js              |     100 |      100 |     100 |     100 |
  ROOTitems.js             |     100 |      100 |     100 |     100 |
  Sidebar.js               |   81.48 |    71.42 |   71.42 |   84.61 | 31,46,79,99
 routes                    |     100 |      100 |     100 |     100 |
  Groups.js                |     100 |      100 |     100 |     100 |
---------------------------|---------|----------|---------|---------|-------------------
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered L
All files	87.74	64.35	84.57	87.73	
components/Forms/Custom	87.65	60.26	85.58	86.87	
Form.js	88.23	55.35	84.61	88	134,188,250
Section.js	80	50	60	80	30,38
Table.js	84.82	58.62	86.53	84.25	73,75,77,79
Utilities.js	92.85	83.33	89.28	92.45	37-38,125,1
pages/FifthRow/EPF	97.64	96.42	91.66	98.68	
Submit.js	97.64	96.42	91.66	98.68	444
pages/OSL/EPF	80.55	68.75	80.95	83.07	
Submit.js	80.55	68.75	80.95	83.07	57,455-470,
pages/Root/EPF	82.81	62.5	73.68	81.96	
Submit.js	82.81	62.5	73.68	81.96	57,95-96,45
routes	100	100	100	100	
UserID.js	100	100	100	100	

Our system testing comprises five test suites that cover the 7 use cases mentioned above. These test suites primarily focused on the routing of pages and testing the integration of the frontend components. Hence, what we tested covered the processes of:

- Login to form submission for EXCOs
- Login to the approval of forms for OSL
- Reset user password
- View from the archive for both EXCOs and OSL
- Account creation and login

The final test report for the five test suites tested locally is shown below:

Started: 2023-08-08 01:04:41

Suites (5)	Tests (5)
5 passed	5 passed
0 failed	0 failed
0 pending	0 pending

E:\University\Term 5\Elements of Software Construction\1D Project\osl-fifth-row-app\frontend\src\system\System_CreateAccount.test.js — 20.699s

System - Fifth Row	passed	17.839s

E:\University\Term 5\Elements of Software Construction\1D Project\osl-fifth-row-app\frontend\src\system\System_ResetPassword.test.js — 19.607s

System - Reset Password	passed	19.039s

E:\University\Term 5\Elements of Software Construction\1D Project\osl-fifth-row-app\frontend\src\system\System_FRE_ViewFormsArchives_System.test.js — 65.488s

System - Fifth Row Submit And OSL Approve	passed	62.603s

E:\University\Term 5\Elements of Software Construction\1D Project\osl-fifth-row-app\frontend\src\system\System_Submit.test.js — 41.509s

System - Fifth Row Submit, OSL Comment And OSL Approve	passed	40.6s

E:\University\Term 5\Elements of Software Construction\1D Project\osl-fifth-row-app\frontend\src\system\System_OSL_ROOT_ViewFormsArchives.test.js — 97.538s

System - Fifth Row Submit And OSL Approve	passed	94.655s

Lessons Learnt

The software development process we adopted was Iterative and incremental in the initial phase followed by the Agile method after the main functionalities have been implemented in the software.

In hindsight, it would be ideal to start developing the software on AWS in the beginning to allow ample time to implement additional features. That said, many members were not experienced in AWS and it could also have been a liability to learn an entirely new skill (usage of AWS) in conjunction with applying the concepts from class. Ultimately, one crucial lesson that we learned was to write good documentation (use cases, class diagrams, sequence diagrams, etc.) as well as read the documentation (for the tools we used like AWS, PostgreSQL, jest, etc.).

Value from reading and writing good documentation:

1. Efficiency: well-documented diagrams, specifications, and class relations decrease ambiguity for the software and speed up the development process.
2. Completeness: reading documentation for the tools used enables the development process to be more thorough in covering edge cases or needs. Eg: Accounting for concurrent queries in the Postgresql database - documentation. Without reading the documentation, we would not have been able to handle concurrent queries.
3. Continuity: good documentation not only provides a good framework for the software but also provides a clear direction as well as a platform to build upon for new features in the future.

In addition, we also learned the importance of frequent communication between both the frontend and backend. This is because we adopted the agile method and there were very frequent changes. During the project, we disseminate updates in the telegram chat which could be flooded with messages easily, thus resulting in some information not being passed down to every member. Thus, towards the end, we opted to meet more regularly in physical meaning and do focused sessions to code. This allows better and more efficient communication between the frontend and backend teams.

Deliverables

Frontend Repo: https://github.com/esctmp/osl-fifth-row-app

Backend Repo: https://github.com/a-nnza-r/ESC_backend

Link to demo video:

https://drive.google.com/file/d/1sQpJY8KudrkNaZZT0YPnvjGOF1c7M81x/view?usp=sharing