School of Computing

CS3219: Software Engineering Principles & Patterns
AY 2020/2021 Semester 1

Developer Documentation
PlanNUS

Code repository URL: https://bit.ly/2Imh9Ek
Deployed at: https://plannus-cfd18.web.app/

Group 09

Name	Matriculation number	Email
Grant Lee	A0183367W	grant@u.nus.edu
Ng Shi Wei	A0185450E	shiweing@u.nus.edu
Lee Wai Fong	A0189078L	e0324662@u.nus.edu
Low Ee Ter	A0183691W	e0310486@u.nus.edu
Mikhail Faiz	A0184253E	mikhailfaiz@u.nus.edu

Table of contents

About PlanNUS

Introduction

PlanNUS is a web application catered towards students of the National University of Singapore (NUS). As the name suggests, it is a tool for students to plan their academic timetables. However, unlike the popular timetable planner application NUSMods, PlanNUS takes a different approach to timetable planning. It prides itself in its simplicity: the timetables are planned *for* the student, rather than *by* the student. In essence, it is designed to be a companion tool (rather than a replacement) for NUSMods and aims to empower NUS students in their academic decision making process.

PlanNUS requires only two inputs from a student. Firstly, the modules that the student intends to take (of course). Secondly, the scheduling considerations that are deemed most important by the student. And then, voila! The most optimal timetables are swiftly generated for the student. In this set of documentation, we will discuss the various components that made PlanNUS possible, including its foolproof user interface as well as its original and innovative set of backend algorithms.

How to run PlanNUS

Run on Firebase Hosting

1. Navigate your browser to the following URL: https://plannus-cfd18.web.app/

Run locally

1. Clone from the repository (link on the cover page).
2. Navigate to the local repository in a terminal window.
3. Change directory to the view folder: `cd view`
4. Install dependencies with: `npm install`
5. Start running the application with: `npm start`

Project details

In this introductory section, we will highlight some key features that are available to users of PlanNUS. The provided screenshots should give you a better idea of the user experience that is associated with each of the features. The implementation of these features will be discussed in detail in the later sections.

Generating new timetables

The feature that sets PlanNUS apart from other academic timetable planners is its unique ability to generate new timetables for its users. This feature solves a problem that is faced by NUS

students biannually — the hassle of having to plan their timetables for the next academic semester, while trying to find the best combination to suit their needs and desires.

The timetables that PlanNUS produces are catered differently for every user, since every user has a different consideration of what is important (and what is not) to them. This functionality is made possible by a set of backend algorithms that performs an intelligent analysis of each timetable and evaluates the extent to which it aligns with the user's specific priorities.

PlanNUS has been implemented in a way that is open to extension, hence developers can easily expand the list of priorities that are available for users to choose. The current list of supported priorities are as follows:

1. Avoid lessons before (Time) every day.
2. Avoid lessons after (Time) every day.
3. Avoid lessons between (Time 1) and (Time 2) every day.
4. Have a maximum number of free days.
5. Minimise travelling across campus.
6. Minimise breaks between classes.
7. Include a lunch break for (Duration) every day.

User input is performed via a minimalist, foolproof user interface. Instead of having to meddle with too many confusing options, the user only needs to do what is natural — to answer the prompt on the screen. An example of this is shown below.

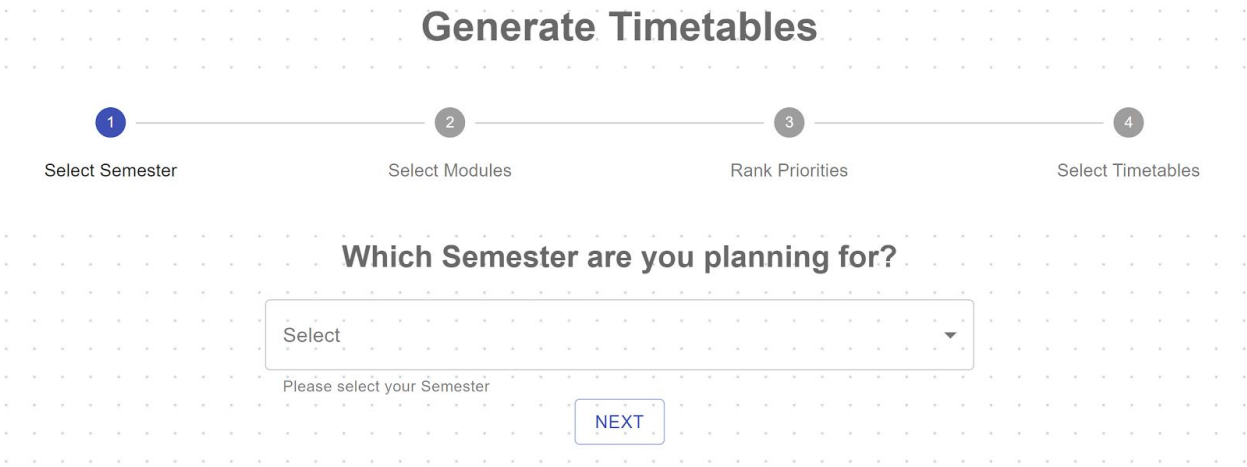

Figure 1: User interface

The first of four steps in the timetable generation tool is the selection of academic semester. As previously mentioned, the minimalist user interface is straightforward and foolproof. In the final step of the timetable generation process, a list of the most optimal timetables are provided.

Saving multiple timetables

When it comes the time to plan their timetable for the upcoming semester, a common practice among NUS students is to keep track of certain alternative timetables. This is due to the unfortunate fact that not all students can eventually secure their most ideal timetable.

PlanNUS seeks to address this issue by providing the functionality of saving multiple timetables for each user. This differs from other academic planners which usually support the saving of just a single timetable. The added functionality of saving multiple timetables grants users the ability to keep track of multiple options in the timetable planning process.

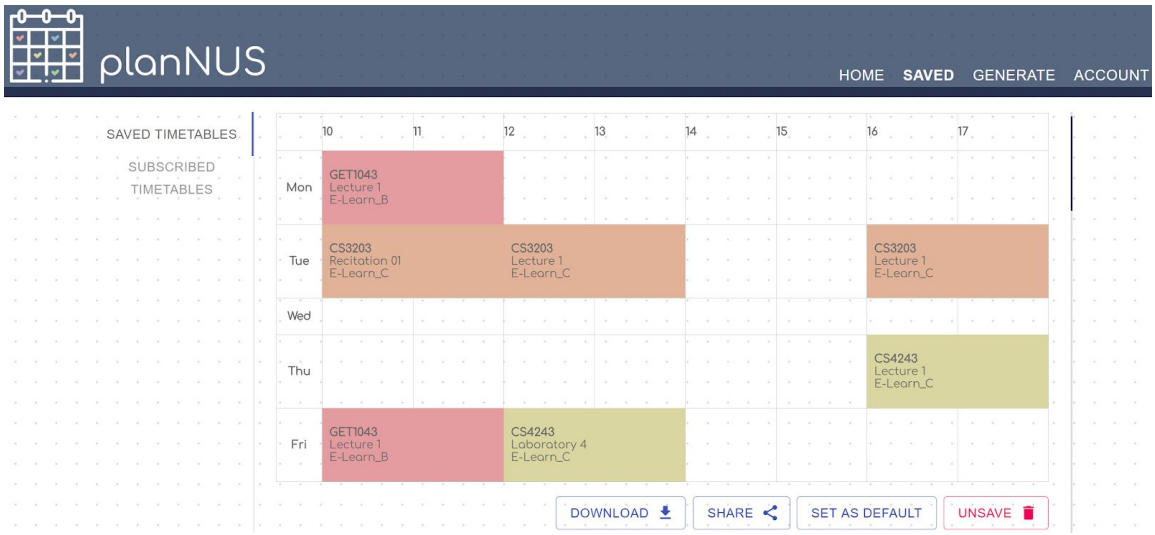

Figure 2: Saved timetables

The screen capture above depicts one of a user's saved timetables. Any timetable that is accessible to the user can be saved (and un-saved). Every timetable, regardless of whether it was generated by the user or shared from a friend, can be saved using the Save button.

Following that, the user can always refer to his/her list of saved timetables within the PlanNUS application. In addition, the user may also save a timetable as his/her default timetable, which will result in the timetable becoming the default view of the user's home page.

Sharing a timetable

Many existing academic planners are designed to be used on a personal basis. However, there are many occasions in which a user may want to share his/her timetable (or one of his/her multiple timetables) with another user. Many existing academic planners are limited in their ability to only export the timetable into an image format. This process of exporting, downloading and sharing the image can be cumbersome for many users.

To address this gap, on top of allowing users to export the timetable as an image file, PlanNUS also provides sharing functionality. The sharing feature provides a unique URL for every saved

timetable, which can be easily copied out into a user's device clipboard. Upon receiving a timetable URL, the user's friend can choose to save or subscribe to the user's timetable.

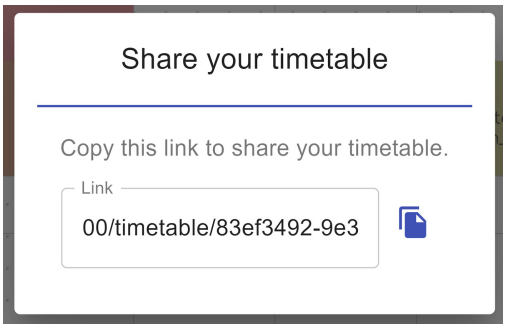

Figure 3: Sharing timetable

Subscribing to timetables

Before we discuss this feature, it would be good to clarify the difference between subscribing to a timetable and saving a timetable. Subscribing to a timetable implies that you are only a viewer of the timetable, and you are subscribing to any updates to the timetable as well. On the other hand, saving a timetable would refer to saving a "local" copy of it, and if the original copy changes or disappears, a user would still have that saved copy.

There are certain scenarios in which a user may prefer to subscribe to another user's timetable, rather than save a copy of it. For example, if a user wants to keep up with any changes in his/her best friend's timetable, he/she should subscribe to the friend's timetable instead of saving it. This is because his/her friend may edit the timetable in the future. (Take note that even though editing timetables is currently not a feature of PlanNUS, it is in the pipeline for future iterations of PlanNUS.)

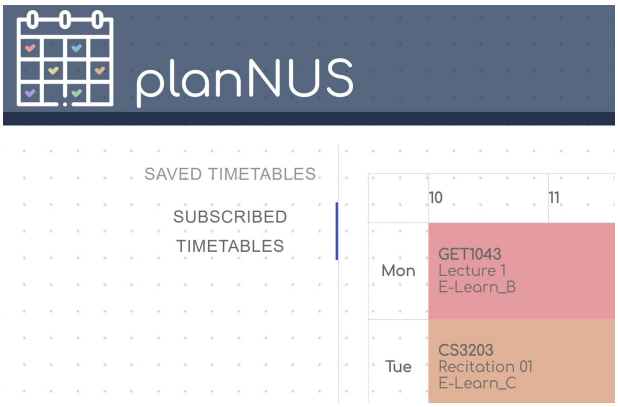

Figure 4: Subscribed timetables

Subscribing to timetables is as easy as hitting the "Subscribe" button. Just like the saved timetables, subscribed timetables can be quickly accessed via the menu as shown above and can also be easily unsubscribed from as well.

Development process

Manpower distribution

To optimise the development process, we divided our team of 5 into 2 sub-teams, the frontend development sub-team as well as the backend development sub-team. In addition, we also assigned a documentation IC. The documentation IC was responsible for ensuring that the project documentation was promptly updated by the team members and that the documentation was coherent.

The table below highlights the sub-teams and additional roles as described above:

Frontend sub-team	Backend sub-team	Documentation
Shi Wei	Ee Ter	Mikhail
Wai Fong	Grant	

Development plan

The general development plan for the entire development phase is as such:

Phase	Week	Goals	Rationale
1	7	Goal setting	To build team dynamics and decide on team logistics
		Development planning	To ensure that every member has a reasonable amount of workload throughout the development phase
	8	GUI mockups	To ensure that the frontend and backend team have a common understanding of how the final product should look and behave
		Setup of authentication system	To ensure that basic authentication for PlanNUS is functional
		Design and setup of database	To ensure that the database design aligns with the software requirements
2	9	Design of backend algorithms	To ensure that the backend algorithms fulfil their required functionalities efficiently
		Basic implementation of GUI	To ensure that the GUI mockups are realised
	10	Implementation of GUI	To connect the view and the controller

		elements	
		Implementation of backend algorithms	To ensure that the backend performs the required functionality as designed
	11	Connection of frontend and backend via APIs	To ensure that API calls are functional and sufficient
	12	Bug fixes	To ensure correct output for each functionality and robustness of system
3	13	Documentation	To finalise documentation about our work for future developers of the project

We then proceeded to plan goals for each phase. These are described as such:

Phase 1: Planning and design

We started the project with Phase 1, which is the planning and design phase. The general aim of this phase was to ensure that the basic project environment was set up for all developers of the team as well as to ensure that everyone had the same understanding with regards to how the eventual product will look and behave.

Phase 2: Implementation

We then moved on to Phase 2, which comprises the bulk of the development process. During this phase, each sub-team worked on the implementation of their components. The goal of this phase was to materialise the visual as well as algorithmic designs from Phase 1. In addition, during this phase, we maintained an open channel of communication to ensure that all team members could clarify any doubts.

Phase 3: Documentation

Finally, we entered Phase 3 of the development process, which was dedicated to uncovering final bugs and patching them, as well as to record our development strategies in our report.

The specific deliverables and development plan can be seen in the following Gantt chart:

Week	7	8	9	10	11	12	13
Set up frontend Firebase proxy							
Setup config							
Setup firebase and authentication context							
Basic authentication				Restructure authentication flow			
Navbar							

Build GUI							
Setup routing							
Login Page							
Build GUI							
Setup Firebase UI							
Redo routing							
Frontend Timetable View							
Build timetable component display						Update presentation logic	
Share function							
Download function							
Save / unsave feature							
Subscribe / unsubscribe feature							
Set default feature							
Account							
Build UI form							
Account verification							
Account detail update function							
Generate							
Stepper for generate timetables							
Select semester feature							
Search for modules feature							
Select modules feature							
Selected modules feature							
Add priorities feature							
Drag and drop feature for ranking priorities							
Select timetables feature							

User management							
User record							
Year, semester							
Modules							
Priorities							
Timetables							
Timetable management							
Generation algorithm and module							
Scoring module							
Generation function							
Documentation							
Introduction							
Software requirements							
Developer plan							
Backend design							
Frontend design							
UI components							
Suggested Enhancements							
Testing							
Integration tests							

Team member contributions

The following table describes the individual contributions of each member to the development of the project:

Member	Individual contributions
Waifong	● GUI Mockups ● Frontend Firebase proxy ● GUI styling and layout ● Generate page ● Select semester component ● Select modules component ● Select timetables component ● Rank priorities component ● Module search bar component and logic ● Requirements documentation ● Design documentation ● Deploy and host on Firebase domain
Shiwei	● Frontend Firebase proxy ● GUI styling and layout ● Frontend context handling ● Timetable component ● Home page ● Routing ● Account page ● Login page ● Nav bar ● Module search bar component and logic ● Implemented `deleteUser` API ● Requirements documentation ● Design documentation ● Deploy and host on Firebase domain
Ee Ter	● Implemented `retrieveModules`, `generateTimetables` API ● Users and Timetables database design ● Module cache database design ● Implemented module cache algorithm ● Timetable generation algorithm ● Optimization of timetable generation algorithm ● Implemented Cloud Functions testing ● Requirements documentation ● Design documentation ● Deploy and host on Firebase domain
Mikhail	● Deploy and host on Firebase domain ● Design documentation ● Requirements documentation ● Manual testing

	● Developer documentation ● Enhancement documentation
Grant	● GUI mockups (using Axure RP) ● Authentication setup ● Users and Timetables database design ● Database setup ● Timetable scoring algorithm ● Implemented `setUserModules`, `setUserPriorities`, `getUserSemester`, `getUserModules`, `getUserPriorities`, `createUser`, `getTimetable`, `setDefaultTimetable`, `getDefaultTimetable`, `getSavedTimetables`, `getSubscribedTimetables`, `saveTimetable`, `unsaveTimetable`, `subscribeToTimetable`, `unsubscribeFromTimetable` APIs ● Requirements documentation ● Design documentation ● Deploy and host on Firebase domain

Software requirements

MOSCOW priority legend

Priority	Meaning
Must	Requirement is essential for the success of the project
Should	Requirement is important but not essential for the success of the project
Could	Requirement is to be considered only if resources permit
Wish	Requirement is to be considered for future versions of the project

Functional requirements

Timetable generation

Sub Category	Code	Priority	Requirement
Semester Selection	FR1.1.1	Must	PlanNUS provides a form for selection of modules
	FR1.1.2	Must	PlanNUS allows users to select any semester from the current academic year
	FR1.1.3	Wish	PlanNUS allows users to select semesters from the upcoming academic year
	FR1.1.4	Should	PlanNUS displays an error message if no semester is selected before proceeding
Priority Selection	FR1.2.1	Must	PlanNUS provides a form for selection of priorities
	FR1.2.2	Must	PlanNUS provides an option to avoid classes before a user-specified time every day
	FR1.2.3	Must	PlanNUS provides an option to avoid classes after a user-specified time every day
	FR1.2.4	Must	PlanNUS provides an option to avoid classes between 2 user-specified times every day

	FR1.2.5	Must	PlanNUS provides an option to maximise the number of free days (i.e. days with no classes)
	FR1.2.6	Must	PlanNUS provides an option to minimise travelling distance across classes
	FR1.2.7	Must	PlanNUS provides an option to minimise breaks between classes
	FR1.2.8	Must	PlanNUS provides an option to include a lunch break for a user-specified duration specified every day
	FR1.2.9	Should	PlanNUS provides a "must have" option that can be toggled ON/OFF for the priorities
	FR1.2.10	Must	PlanNUS will display an error message if no priorities are selected before proceeding
	FR1.2.11	Should	PlanNUS saves the selected priorities in a database
Module Selection	FR1.3.1	Must	PlanNUS provides a form for module selection
	FR1.3.2	Must	PlanNUS fetches all available modules for a selected semester
	FR1.3.3	Must	PlanNUS allows users to search for modules by typing the module code
	FR1.3.4	Should	PlanNUS allows users to search for modules by typing the module name
	FR1.3.5	Should	PlanNUS saves the selected modules in a database
Algorithm	FR1.4.1	Must	PlanNUS generates all timetable permutations based on selected modules
	FR1.4.2	Must	PlanNUS calculates a score for each generated timetable based on how fully it satisfies the selected preferences
	FR1.4.3	Must	PlanNUS sorts the generated timetables using their calculated scores in descending order
	FR1.4.4	Must	PlanNUS displays the top 20 generated timetables
	FR1.4.5	Must	PlanNUS displays an error message if no timetables can be generated based on the preferences selected (i.e. score for every timetable is 0)

Timetable access

Sub Category	Code	Priority	Requirement
Saving	FR2.1.1	Must	PlanNUS allows users who are logged in to save one timetable for a given semester to a database
	FR2.1.2	Should	PlanNUS allows users who are logged in to save more than one timetable for a given semester to a database
	FR2.1.3	Must	PlanNUS allows users to un-save any previously saved timetable(s) from the database they were saved in
Viewing	FR2.2.1	Must	PlanNUS allows users to access and view any of the timetable(s) they have saved
	FR2.2.2	Wish	PlanNUS allows users to edit their saved timetable(s)
	FR2.2.3	Should	PlanNUS allows users to set a timetable as the "default" which will be the timetable shown on the home page each time the application is launched
Sharing	FR2.3.1	Should	PlanNUS allows users to share their saved timetable(s) with other users
	FR2.3.2	Could	PlanNUS allows users to subscribe to timetable(s) that have been shared with them
	FR2.3.3	Could	PlanNUS allows users to save copies of any timetable(s) that have been shared with them
Download	FR2.4.1	Should	PlanNUS allows users to download their timetable as an image file.

User authentication

Sub Category	Code	Priority	Requirement
Account Creation	FR3.1.1	Must	PlanNUS allows users to create accounts
	FR3.1.2	Must	PlanNUS allows users to register for accounts using their emails
	FR3.1.3	Should	PlanNUS allows users to register for accounts using Google sign-in
	FR3.1.4	Must	PlanNUS allows only one account to be registered for each unique email address

	FR3.1.5	Could	PlanNUS allows users to enter their first and last name during registration
	FR3.1.6	Must	PlanNUS allows users to set a password for their account during registration
	FR3.1.7	Could	PlanNUS sends new users a confirmation email upon signing up for an account
Account Management	FR3.2.1	Must	PlanNUS allows users to sign in and out of their accounts
	FR3.2.2	Must	PlanNUS allows users to change the password for their accounts
	FR3.2.3	Could	PlanNUS allows users to delete their accounts

Non-functional requirements

Usability

Sub Category	Code	Priority	Requirement
Written Language	NFR1.1.1	Must	PlanNUS will support standard British English as a language
	NFR1.1.2	Wish	PlanNUS will support Mandarin as a language
	NFR1.1.3	Wish	PlanNUS will support Malay as a language
	NFR1.1.4	Wish	PlanNUS will support Tamil as a language
Tutorial	NFR1.2.1	Could	PlanNUS will provide first-time users a tutorial for usage and navigation
	NFR1.2.2	Could	PlanNUS has a help page containing a tutorial and frequently asked questions (FAQs)
	NFR1.2.3	Should	PlanNUS will show the user which step in the process of timetable generation they are currently in
	NFR1.2.4	Should	PlanNUS will notify the user with an error message when they attempt an action that cannot be performed
Aesthetics	NFR1.3.1	Could	PlanNUS has its own uniquely identifiable logo

	NFR1.3.2	Wish	PlanNUS has an option to switch between dark/light mode
	NFR1.3.3	Wish	PlanNUS has an option to choose/change color theme of timetables

Security

Sub Category	Code	Priority	Requirement
Authentication	NFR2.1.1	Should	PlanNUS will enforce a 6 character minimum length when users are setting a new password for their accounts
	NFR2.1.2	Should	PlanNUS will enforce an inclusion of numbers and symbols when users are setting a new password for their accounts.
	NFR2.1.3	Should	PlanNUS allows users to reset their password through their registered email if users forget their password
	NFR2.1.4	Wish	PlanNUS prevents subsequent login attempts if an account's password is incorrectly entered 5 times in a row
	NFR2.1.5	Wish	PlanNUS sends a warning email to the registered email of an account if the incorrect password is entered 5 times in a row
Privacy	NFR2.2.1	Must	PlanNUS will encrypt passwords and sensitive information such IP addresses
	NFR2.2.2	Should	PlanNUS allows users to view only the display names of other users

Performance

Sub Category	Code	Priority	Requirement
Latency	NFR3.1.1	Should	PlanNUS will respond within 1 second to any user input if the user's internet connection is stable
	NFR3.1.2	Should	PlanNUS will fetch all available modules from NUSMods within 10 seconds if the user's internet

			connection is stable
	NFR3.1.3	Should	PlanNUS will load saved timetable(s) within 10 seconds if the user's internet connection is stable
Algorithm efficiency	NFR3.2.1	Should	PlanNUS will complete timetable generation and scoring within 10 seconds if the user's internet connection is stable

Scalability

Code	Priority	Requirement
NFR4.1	Could	PlanNUS will be able to accommodate 10,000 concurrent users at any time

Installability

Code	Priority	Requirement
NFR5.1	Must	PlanNUS does not require local installation to be used

Interoperability

Code	Priority	Requirement
NFR6.1	Wish	Timetables saved on PlanNUS can be imported into NUSMods
NFR6.2	Should	PlanNUS allows timetables to be exported in .PNG format

Quality attributes analysis

	Interoperability	Installability	Performance	Scalability	Security	Usability
Interoperability		<	^	^	^	^
Installability			^	^	^	^
Performance				<	^	^
Scalability					^	^
Security						<
Usability						

Prioritising security can result in a decrease in usability. For example, the implementation of NFR2.1.1 forces users to set passwords of sufficient complexity (6 characters). This may negatively affect usability for some users who prefer to use a consistent password across their accounts on different platforms, and that password does not meet the complexity requirements of PlanNUS.

External interface requirements

User interfaces

Code	Priority	Requirement
EIR1.1	Must	PlanNUS utilises Material-UI for styling
EIR1.2	Should	PlanNUS allows users to navigate to different pages at any time through persistent navigation bar
EIR1.3	Should	PlanNUS will return to the home page when the user clicks the PlanNUS logo
EIR1.4	Wish	PlanNUS will support keyboard shortcuts
EIR1.5	Should	PlanNUS has a consistent design throughout all the pages it features

Software interfaces

Code	Priority	Requirement
EIR2.1	Must	PlanNUS uses Firestore as its storage mechanism
EIR2.2	Must	PlanNUS will run on all modern web browsers
EIR2.3	Must	PlanNUS will run on Windows, MacOS, and Linux operating systems
EIR2.4	Should	PlanNUS will run on all modern mobile browsers

Communication interfaces

Code	Priority	Requirement
EIR3.1	Must	PlanNUS uses HTTP as its communication protocol
EIR3.2	Must	PlanNUS uses REST APIs to perform requests to NUSMods API

Assumptions and dependencies

NUSMods API

Code	Assumption/Dependency
AD1.1	Module information fetched from NUSMods API is assumed to be complete
AD1.2	Module information fetched from NUSMods API is assumed to be correct
AD1.3	Module information fetched from NUSMods API is assumed to be up to date

Software design

Technology stack

React.js and Material UI

The frontend application is written in React and Material UI is used for component styling.

Node.js and Firebase Cloud Function

The backend logic is implemented as Firebase Cloud Functions and deployed onto Google servers, running in a Node.js (v12) environment.

Cloud Firestore

Cloud Firestore provides the backend storage.

Firebase Hosting

The web application is deployed with Firebase Hosting.

Software architecture

The implementation of PlanNUS follows a 2-tier layered architecture, comprising the presentation layer and a combined “server” (which consists of the application layer as well as the database layer).

This architecture is made possible by employing the Firebase serverless framework, which provides a means to integrate the application and database layers together. Backend processes in the application layer are enabled via Firebase Cloud Functions, which automatically run backend code in response to events triggered by Firebase features and HTTPS requests. Database functionality is provided by Cloud Firestore, which can be manipulated via API calls in the Firebase Cloud Functions.

The following diagram provides a high-level overview of the 2-tier layered architecture:

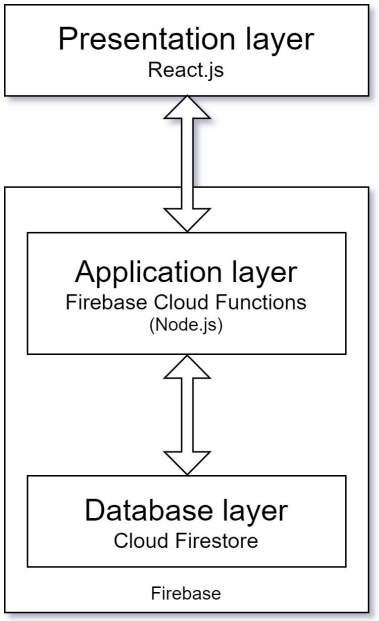

Figure 5: 2-tier layered architecture

The following diagram provides a more detailed architecture, highlighting the specific components of each layer. Discussion about the specific implementation of each sub-component follows in the next section.

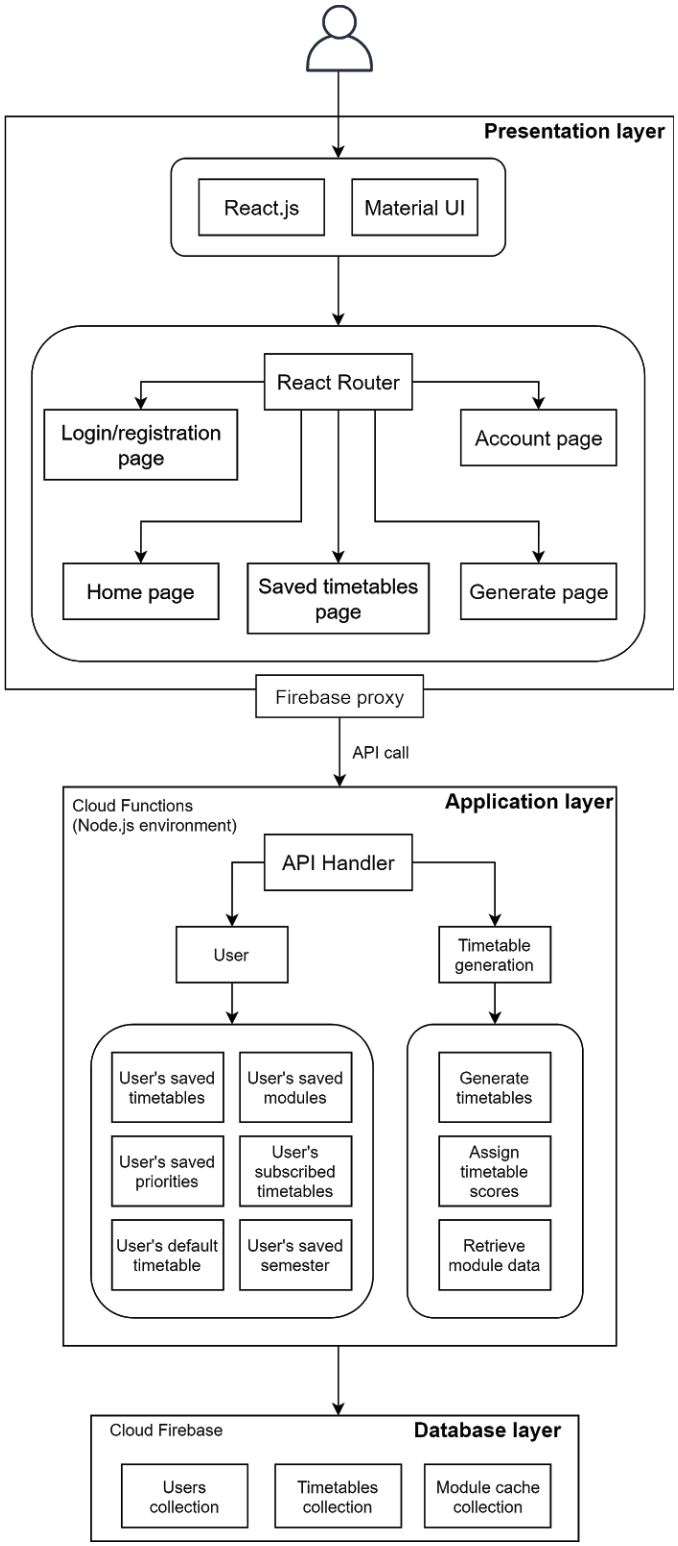

Figure 6: Overall architecture

The following sequence diagram gives a generic understanding of the interaction between the above-mentioned components:

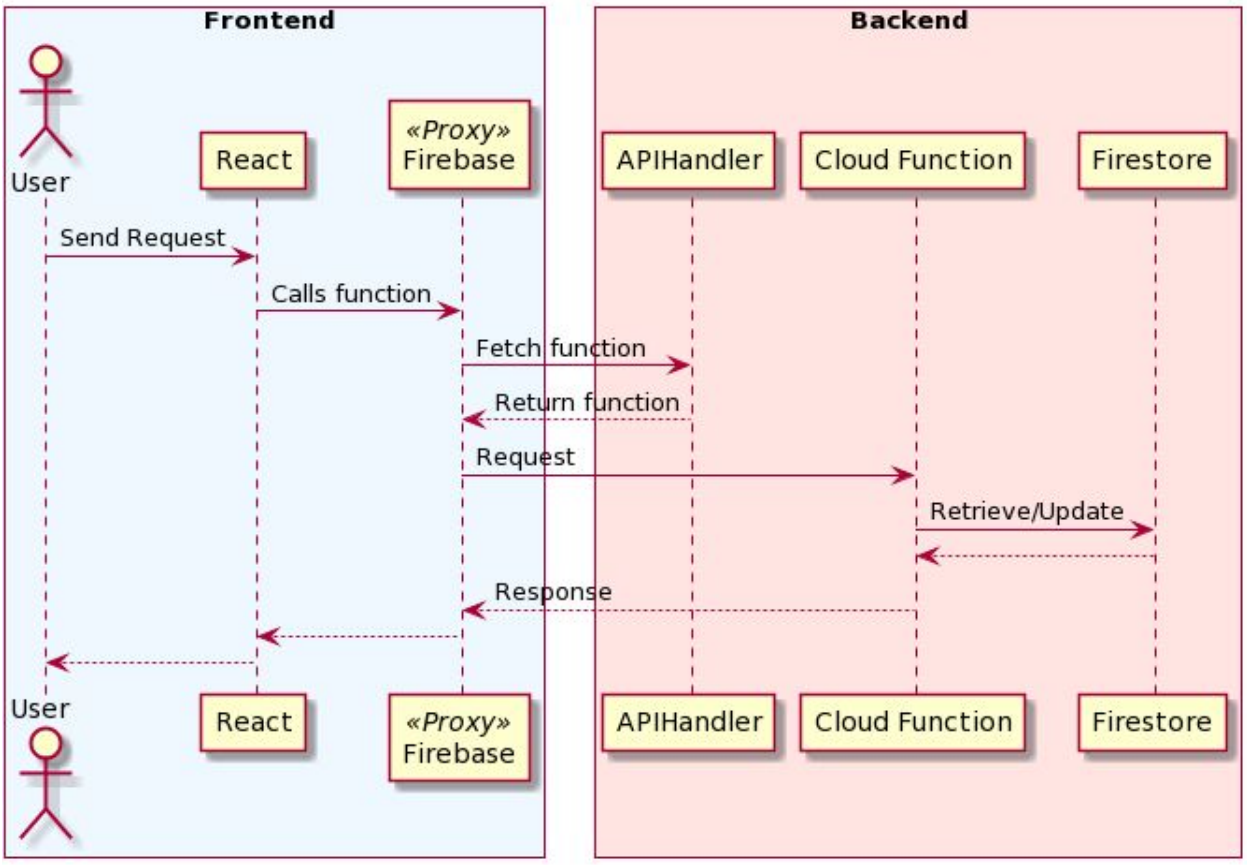

Figure 7: High-level sequence diagram

More specific examples will be given in the “Component interaction” section below.

JavaScript object interface

The presentation layer, application layer and database layer utilize JavaScript. Thus, we define an interface imposed on objects that the layers use to communicate. This lets all the layers use or program directly to this interface, which reduces the need to convert from one data format to another, and eliminates the confusion that would be encountered if different layers use different object formats. This methodology of data transfer between the layers adheres to the **Data Transfer Object pattern**.

Timetable

Where a timetable is expected, it should follow the format described below:

```
{
    score: number,
    year: number;
    semester: number;
    modules: string[];
    events: TimetableEvent[];
}
```

The properties of a Timetable object are:

1. `score`: The score (out of 100 percent) assigned to the timetable by the scoring algorithm
2. `year`: The academic year associated with this timetable
3. `semester`: The academic semester associated with this timetable
    - `1` represents Semester 1
    - `2` represents Semester 2
    - `3` represents Special Term (Part 1)
    - `4` represents Special Term (Part 2)
4. `modules`: The modules for which this timetable includes
5. events: An array containing Event objects, representing the lessons(tutorials, lectures, labs, sectionals, etc.) in this timetable

TimetableEvent

Each `TimetableEvent` is a class that has to be attended if the timetable is followed. It is structured as follows:

```
{
    moduleCode: string;
    lessonType: string;
    classNo: string;
    location: string;
    day: number;
    startTime: number;
    endTime: number;
    weeks: number[]
}
```

A `TimetableEvent` object has the following fields:

1. `moduleCode`: The module associated with this lessons
2. `lessonType`: The type of lesson
3. `classNo`: The class number of the lesson
4. `location`: The location of the lesson
5. `day`: The day of the week on which the lesson happens
    - `0` represents Monday, `1` represents Tuesday, and so on
6. `startTime`: The starting time of the lessons, represented as minutes from midnight
    - For example, `840` represents 2:00 P.M. (840 ÷ 60 = 14 hours after 12:00 A.M.)
7. `endTime`: The ending time of the lessons, represented as minutes from midnight
8. `weeks`: An array of weeks on which the lesson falls

In particular, the `day`, `startTime` and `endTime` fields are designed to be as such for two reasons. Firstly, it makes it easier for the timetable score to be calculated. Additionally, it makes it more convenient for the frontend to use the integer values to populate the timetable view.

The Timetable object was designed with the intention of being easily “deconstructed” into a view element in the frontend. The following code fragment is an example of a Timetable JavaScript object containing two `TimetableEvent` objects:

```
{
    "score": 50.0,
    "year": 2020,
    "semester": 1,
    "modules": [
        "CS3219"
    ],
    "events": [
        {
            "moduleCode": "CS3219",
            "lessonType": "Lecture",
            "location": "COM1-01-01",
            "classNo": "02",
            "day": 3,
            "startTime": 840,
            "endTime": 960,
            "evenWeek": true,
            "oddWeek": true,
            "weeks": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
        },
        {
            "moduleCode": "CS3219",
            "lessonType": "Tutorial",
            "location": "COM1-03-02",
            "classNo": "04",
            "day": 4,
            "startTime": 600,
            "endTime": 660,
            "evenWeek": true,
            "oddWeek": true,
            "weeks": [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
        }
    ]
}
```

Priority

Each priority follows the following format:

```
{
    id: string;
    rank: string;
    fields: object;
    mustHave: boolean;
    name: string;
    type: string
}
```

Presentation layer

Package Overview

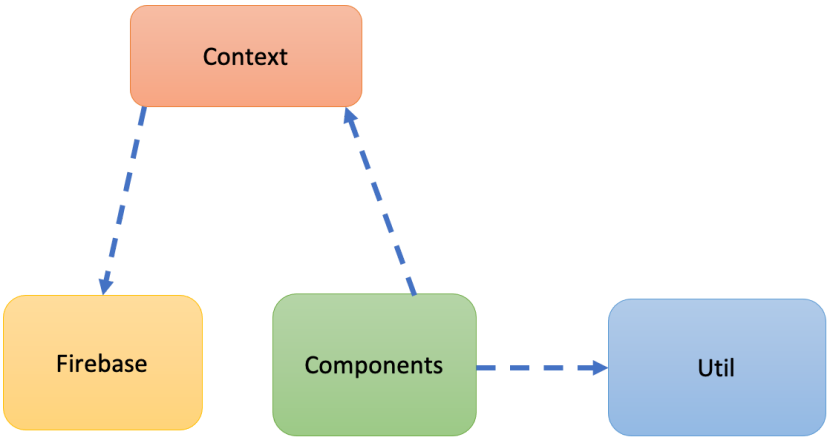

Figure 8: Package overview

Diagram above illustrates the frontend package overview of this application.

Context

The context contains the Firebase object. Components which require the Firebase API consume this context. This will be further elaborated on under **Provider pattern**.

Components

Components are reusable React classes including the UI and display logic. Components can be nested to allow for **Separation of Concerns** as it divides the page into different portions that serve different functions.

Firebase

Firebase object which contains all functionalities of Firebase required by the application.

Util

Util contains all common utility functions and constants used in the application

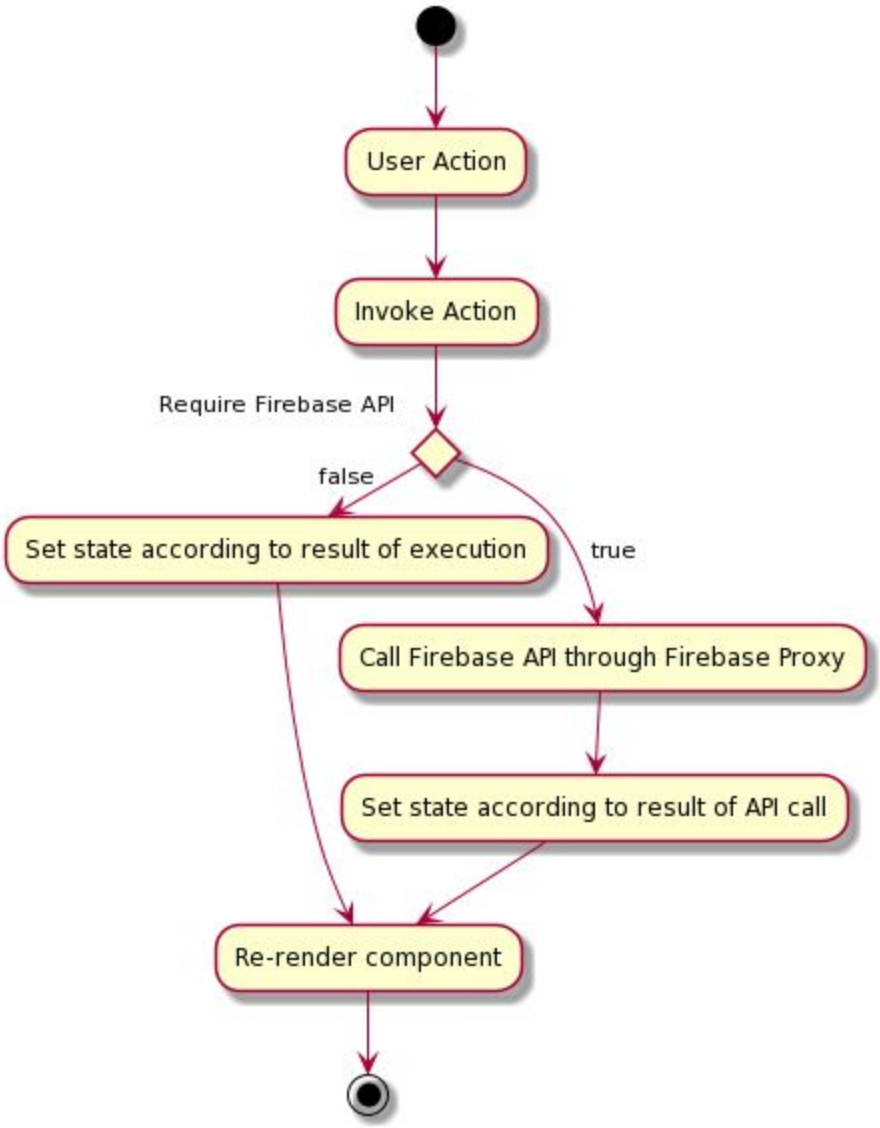

Figure 9: General user action activity diagram

The activity diagram above illustrates the process whenever there is an user action in the GUI.

Frontend design patterns and principles

Firebase proxy

The **proxy design pattern** was used by having a custom `Firebase` class that handles communication with Firebase API. Some functions of the class include:

- Defining the configuration for initializing a Firebase application
- Maintains the current user logged in with the application
- Providing methods that utilizes the Firebase Cloud Functions, in which the methods format input for the Cloud Functions and response for React components

This abstracts out the logic of importing the Firebase library and formatting HTTP requests and responses from the React components, which allows for Separation of Concerns where the React components can instead focus on the presentation logic. The diagram below shows an example.

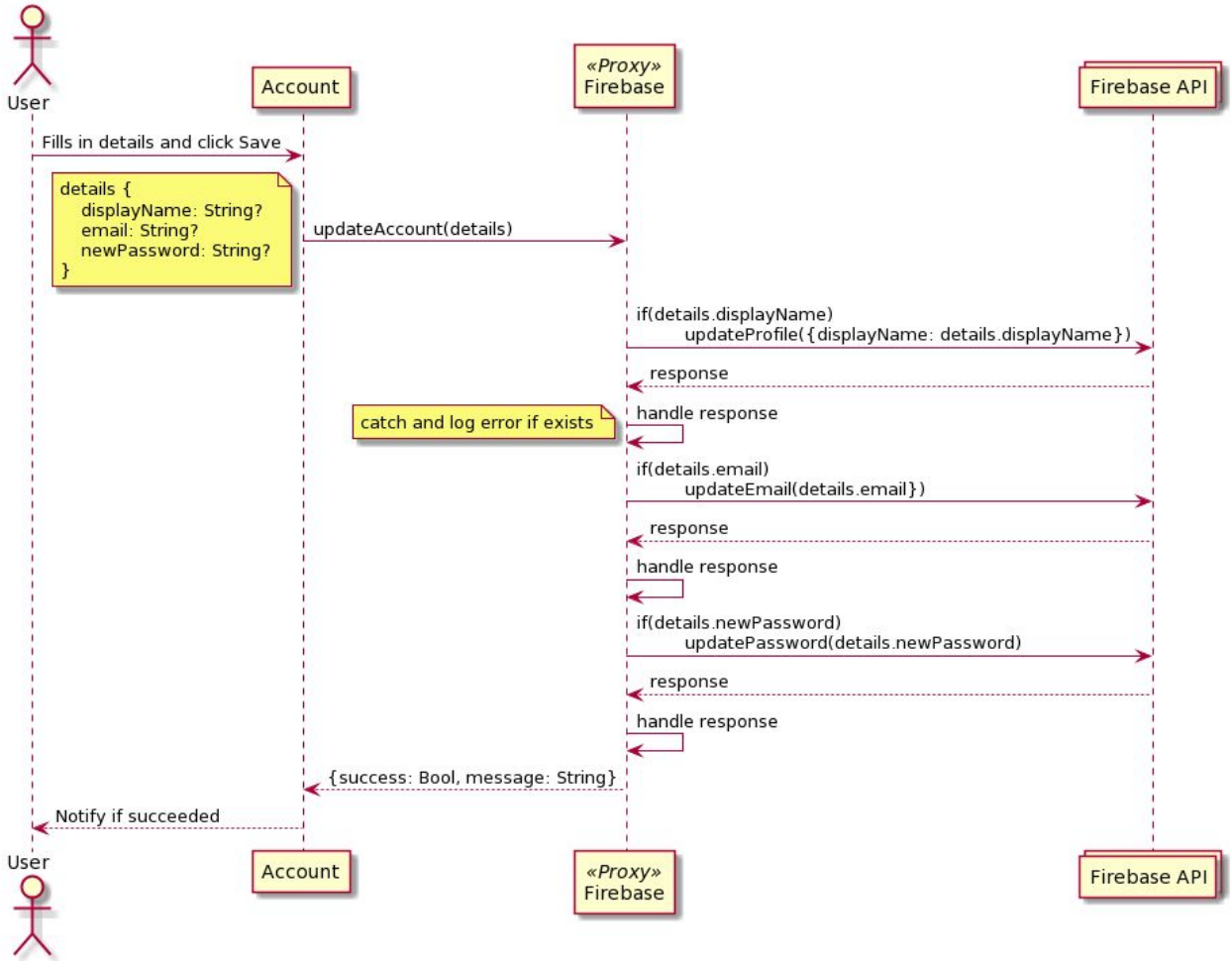

Figure 10: Use of proxy design pattern

The `Account` component provides text fields for the user to fill in individually if the user wants to change the account details. The text fields include display name, email and password. When the user clicks “Save Changes”, the component will retrieve the input from the text fields and wrap them into an object to pass into the `updateAccount` method provided by the `Firebase` proxy.

The `updateAccount` method takes in a `details` object and returns a `response` object.
`details` object contains:

- `displayName`: an optional string
- `email`: an optional string
- `newPassword`: an optional string

`response` object contains:

- `success`: a boolean
- `message`: a string

The Firebase proxy checks if the input contains a displayName, and if so, formats it into an object of `{displayName: details.displayName}` before passing the formatted input into the Firebase function. If an error response is received, it will log the error and set `response.success = false` and `response.message = “Unable to update display name.”` before returning the response. Similarly for email and password. If all updates perform successfully, it will set `response.success = true` and return the response.

Upon receiving the response, the Account component checks if the update succeeded. If the update succeeded, a success alert message is shown and the text fields reset to display the updated account details. Else, the error message in `response.message` is displayed.

Other than Separation of Concerns, the Firebase proxy also allows the methods to be reused. For example, the method `fetchTimetable(id)` is used when fetching the default timetable, saved timetables or a shared timetable.

Provider pattern

React provides a Contexts API that is used to solve the problem of prop drilling. Prop drilling is the process we have to go through to get data to parts of the React component tree. If the component that requires some data is deeply nested, we need to pass the data down as props to every level of the component tree, even if some intermediate components do not require it.

Contexts are created to pass data through components without having to pass it as props manually in every component. In other words, any component that requires some data can consume the context no matter how deeply nested it is. Parent components that do not require that particular data do not have to consume it. This ensures that we do not have to pass the data down the component tree as prop manually in every level even if some intermediate component does not require it.

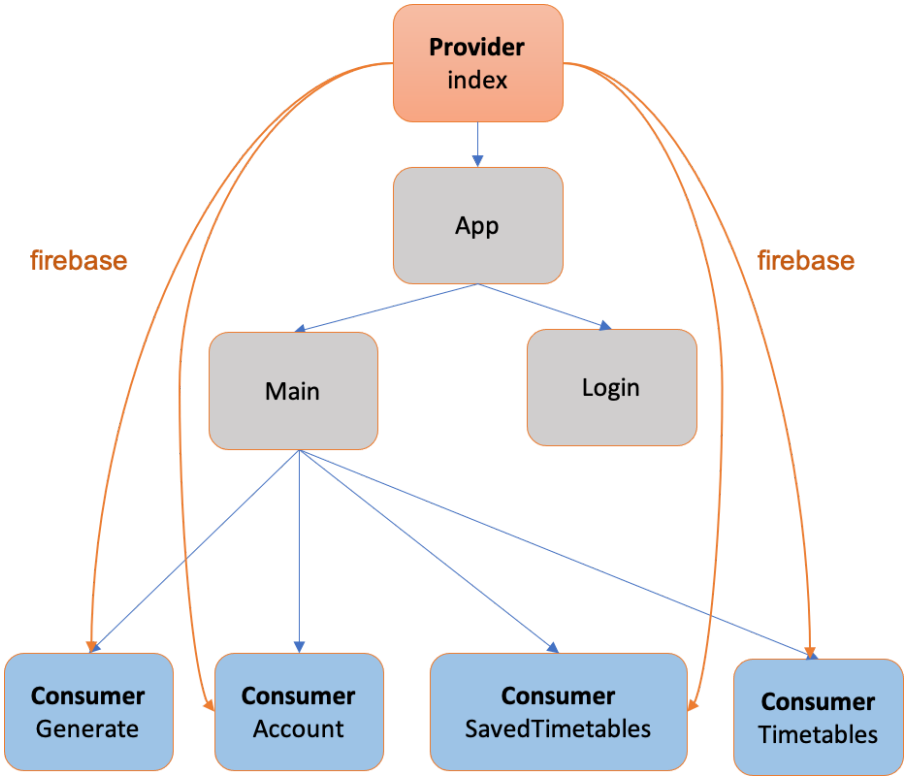

Figure 11: Use of provider pattern

Since our project utilises Firebase Cloud Functions, most of our components required access to the Firebase object. We created the context, `withFirebase`, where we passed the Firebase object as value:

```
const FirebaseContext = React.createContext(null);
export const withFirebase = Component => props => (
   <FirebaseContext.Consumer>
       {firebase => <Component {...props} firebase={firebase} />}
   </FirebaseContext.Consumer>
);
```

The context is provided in `index.js` and any component which requires the Firebase object imports the `withFirebase` function and consumes it. Thereafter, the component is able to access the Firebase object by calling `props.firebase`.

Example usage by the `Account` component:

```
componentDidMount() {
       this.observer = this.props.firebase.observeCurrentUser((user) => {
           if (this.signal?.aborted) return
           this.updateState(user)
       })
}
```

The function in the Firebase class `observeCurrentUser` accepts a callback function and the Account component calls it in componentDidMount via `props.firebase.observeCurrentUser` to get the updated account details for display.

```
export default withAuthUserConsumer(withFirebase(Account));
```

`Account` component is exported with `withFirebase(Account).`

Container and presentation components

Container and presentation components are commonly used in React applications. Container components handle the interaction of the presentational components. They provide data and callback actions for the presentational components and keep states about the logic of the component. Presentational components handle the display and take in data and callback actions via props. They keep states about the UI presentation.

This pattern provides separation of concerns where container components perform imports of dependencies for fetching and loading data while presentational components perform styling and layout of the data. This also allows presentational components to be reusable in different scenarios with different data sources. Below is an example of how the pattern is used for timetable displaying.

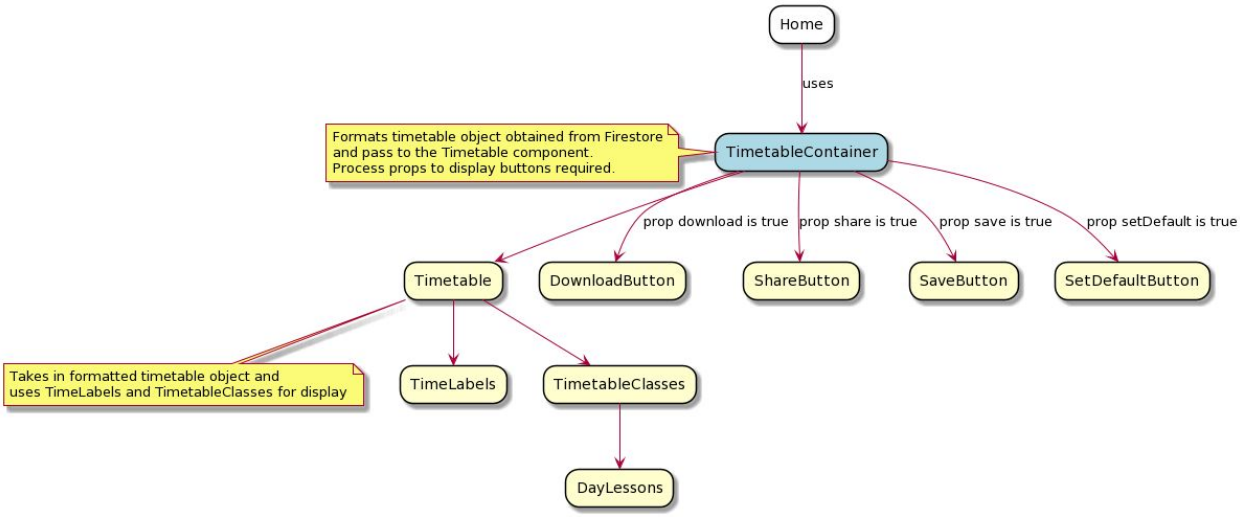

Figure 12: Container and presentation components

Container component:

- TimetableContainer

Presentational components:

- Timetable
- DownloadButton
- ShareButton
- SaveButton
- SetDefaultButton

`TimetableContainer` receives a timetable object from its parent component. The timetable object is in the format used in the backend with a list of classes and their timings. TimetableContainer formats the timetable object by:

- Generating the time labels
- Providing color for each module
- Calculating the duration of each class (in minutes)
- Checking for overlapping time slots (for odd and even week classes)
- Group classes by day

`Timetable` receives the timetable object from `TimetableContainer` and performs layouts and styling by

- Using duration of each class to calculate width of cells
- Using overlapping time slots to determine height of each row and height of class cells

`TimetableContainer` also receives props `download, share, save, setDefault` to determine which buttons to render. For example, a generated timetable will require a save button but not a share button.

`TimetableContainer` also imports the `Firebase` class and external library `'react-component-export-image'` as a dependency and utilizes the functions in the class for callbacks to pass into the buttons.

- `onSave()` and `onUnsave()` callbacks are passed to the `SaveButton`
- `onSetDefault()` callback is passed to the `SetDefaultButton`
- `onDownload()` callback passed to `DownloadButton`

Design Decisions

Debounce

Design decision: How to make module search more efficient?	
Problem	The search function was implemented by fetching the list of modules from the backend and performing a filter whenever user input changes. However, as the module list was large, every time filter was performed, there would be a significant lag in rendering.
Decision	After some research, we decided to make use of the debounce function from the lodash library. Debounce limits the rate in which filter is performed, preventing significant lags due to the filter being called too frequently.

Lazy render

Design decision: How to list timetables and modules more efficiently?	
Problem	Trying to list all timetables at once significant lag in rendering as the browser is trying to render too many components at once. This is especially so as the timetable is a huge object and undergoes formatting every time it is rendered. Similarly with the listing of modules as the list is long.
Decision	We decided to make use of the libraries “react-infinite-scroll-component” and “react-virtualized-auto-sizer” to implement lazy rendering. With lazy rendering, the component will only be rendered when it is needed. For example, when the user scrolls to the relevant part of the list. A tradeoff of lazy rendering is an increase in loading time.

UI components breakdown

The following diagrams give a graphical overview of the respective UI components:

Home.jsx

Figure 13: Breakdown of Home.jsx

TimetableContainer.jsx

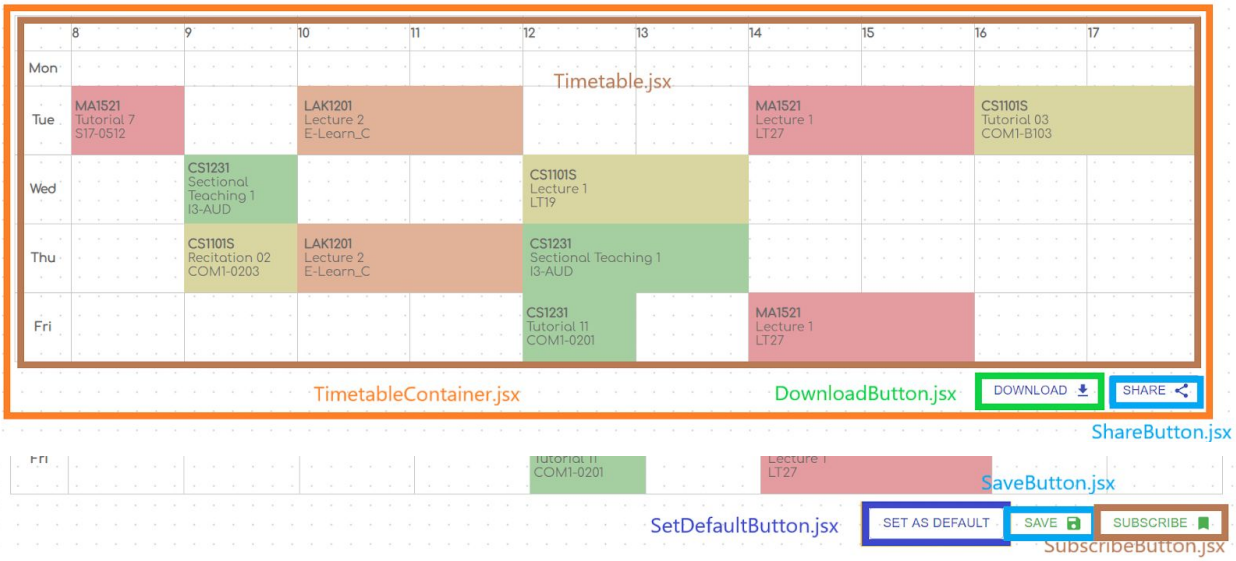

Figure 14: Breakdown of TimetableContainer.jsx

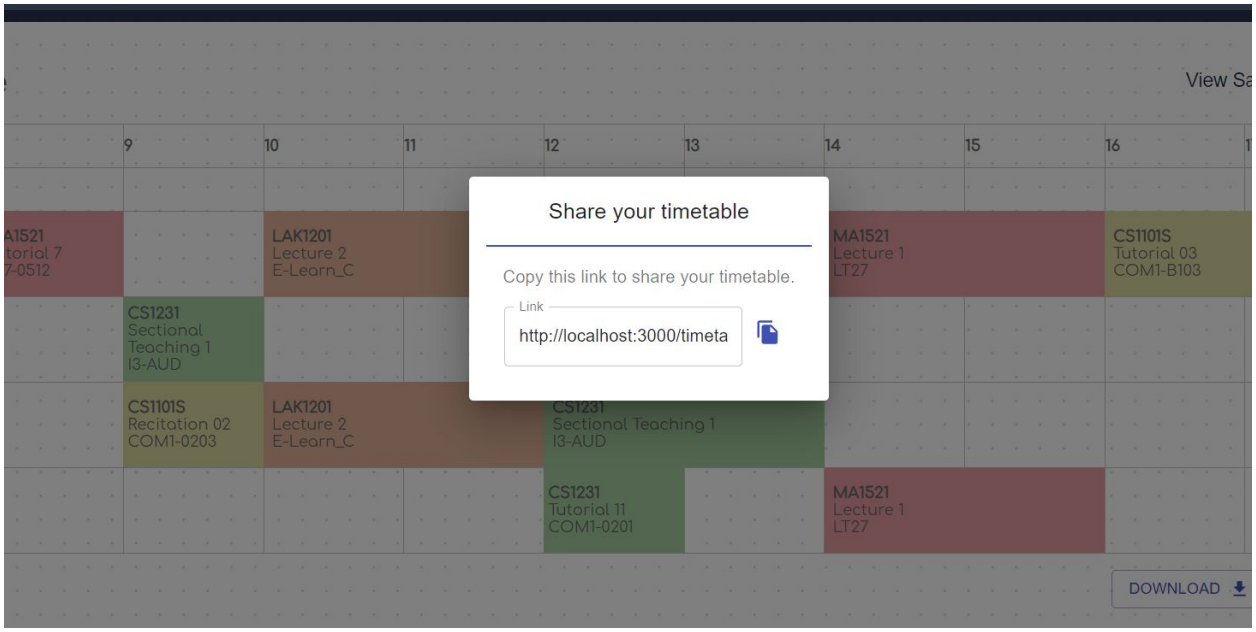

Figure 15: On clicking on Share Button

Generate.jsx

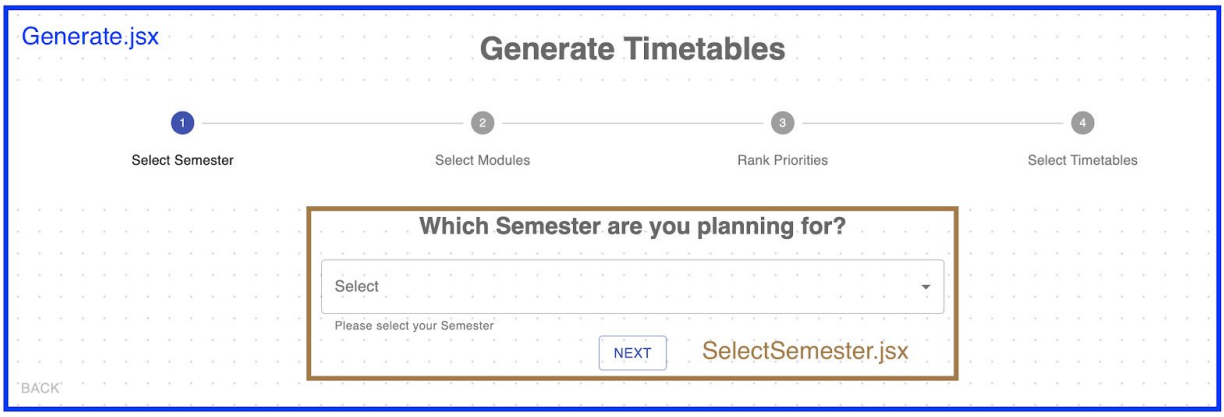

Figure 16: Breakdown of Generate.jsx - Select Semester

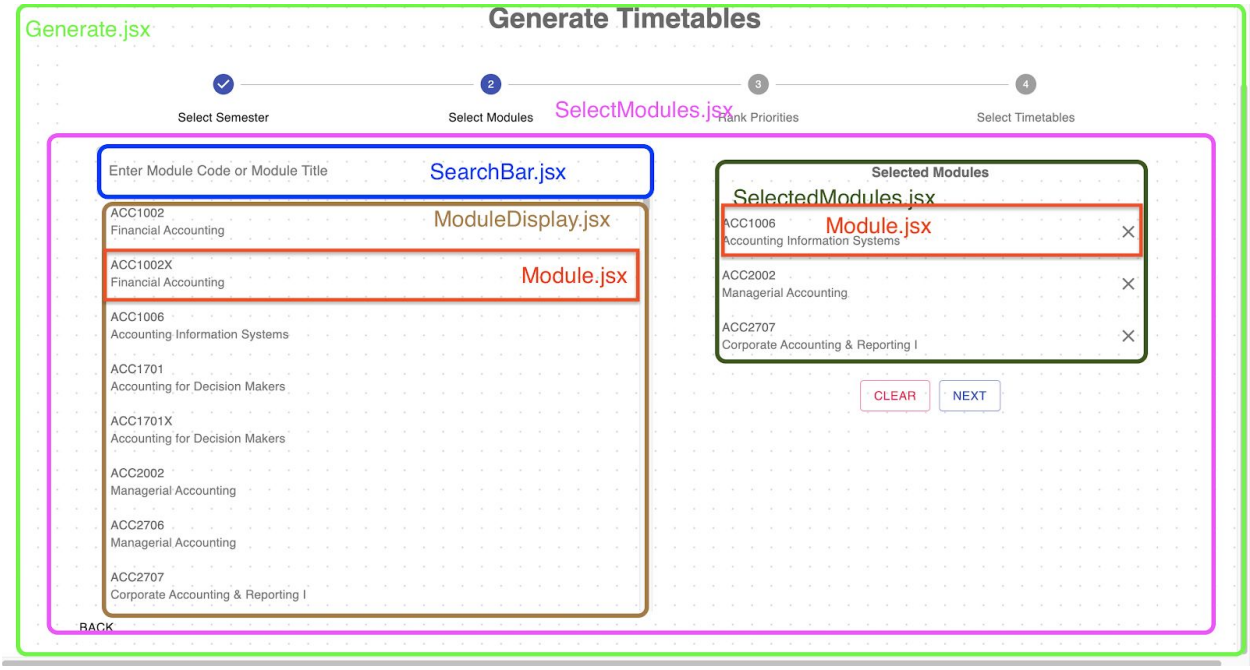

Figure 17: Breakdown of Generate.jsx - Select Modules

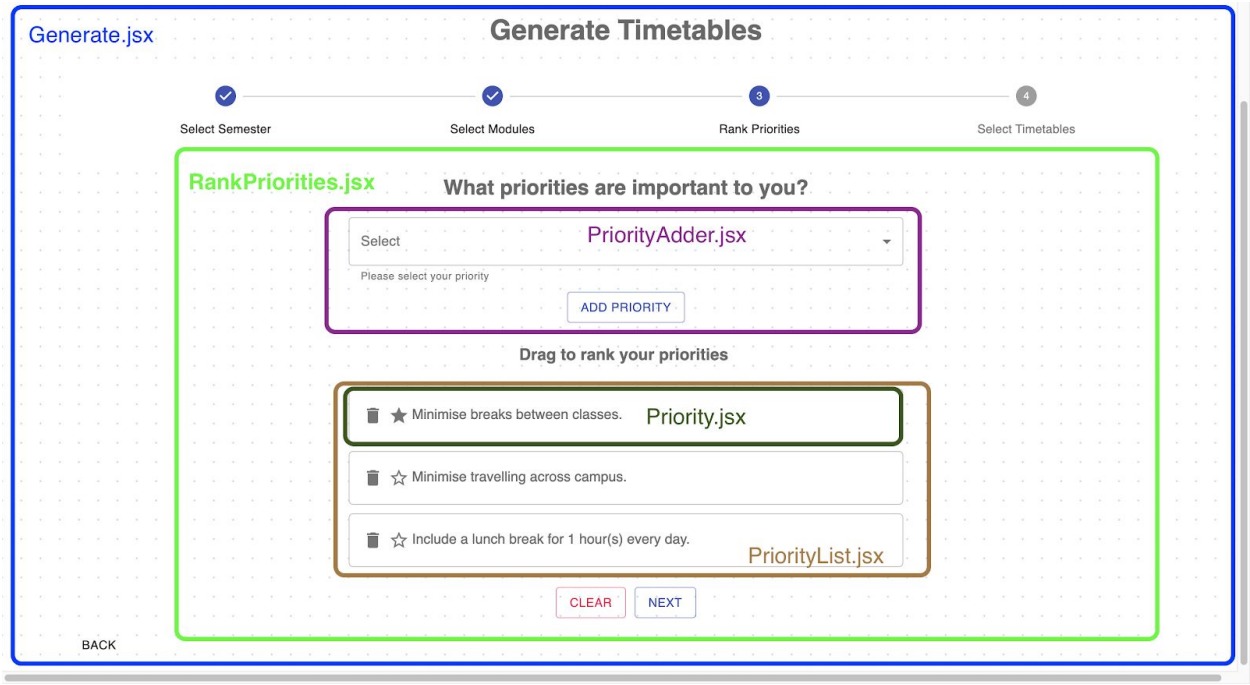

Figure 18: Breakdown of Generate.jsx - Rank Priorities (1)

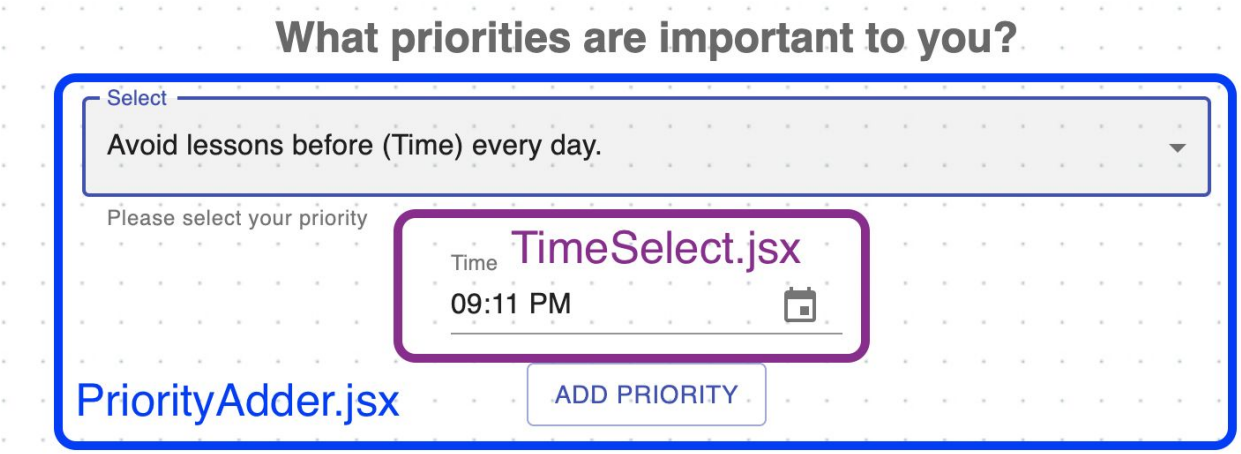

Figure 19: Breakdown of Generate.jsx - Rank Priorities (2)

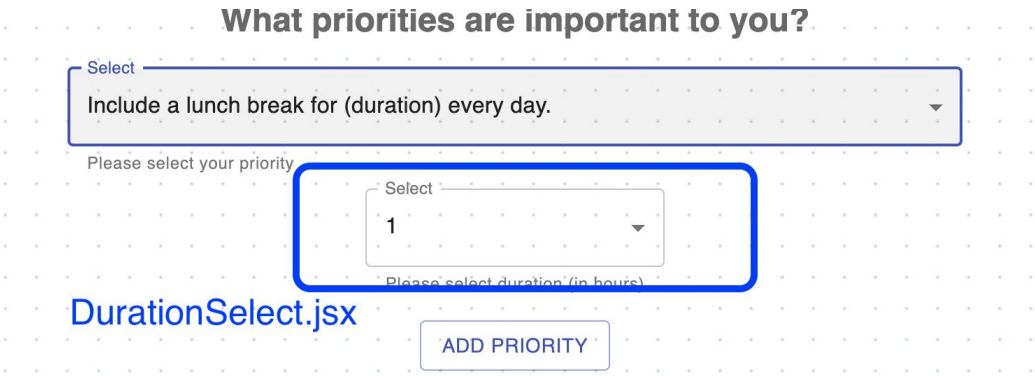

Figure 20: Breakdown of Generate.jsx -Rank Priorities (3)

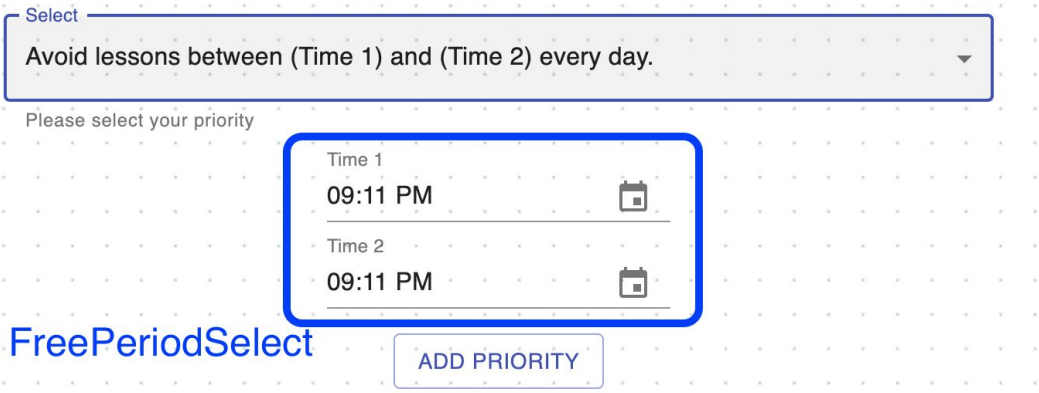

Figure 21: Breakdown of Generate.jsx - Rank Priorities (4)

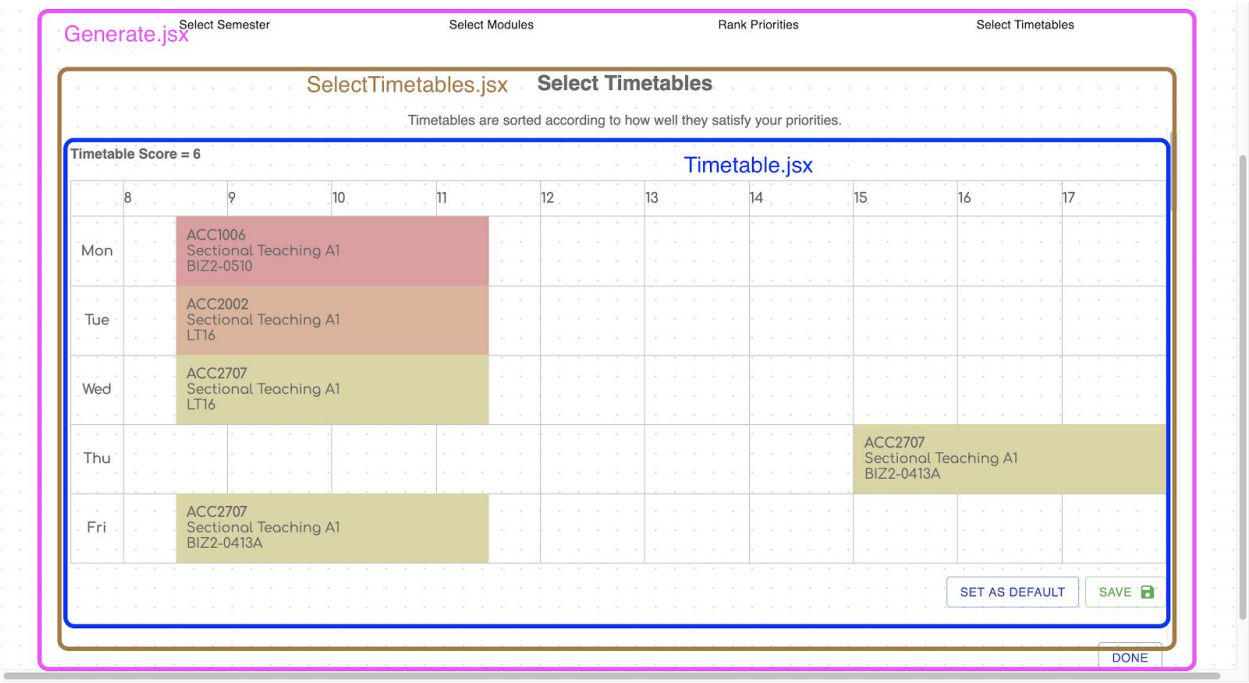

Figure 22: Breakdown of Generate.jsx - Select Timetables

Saved.jsx

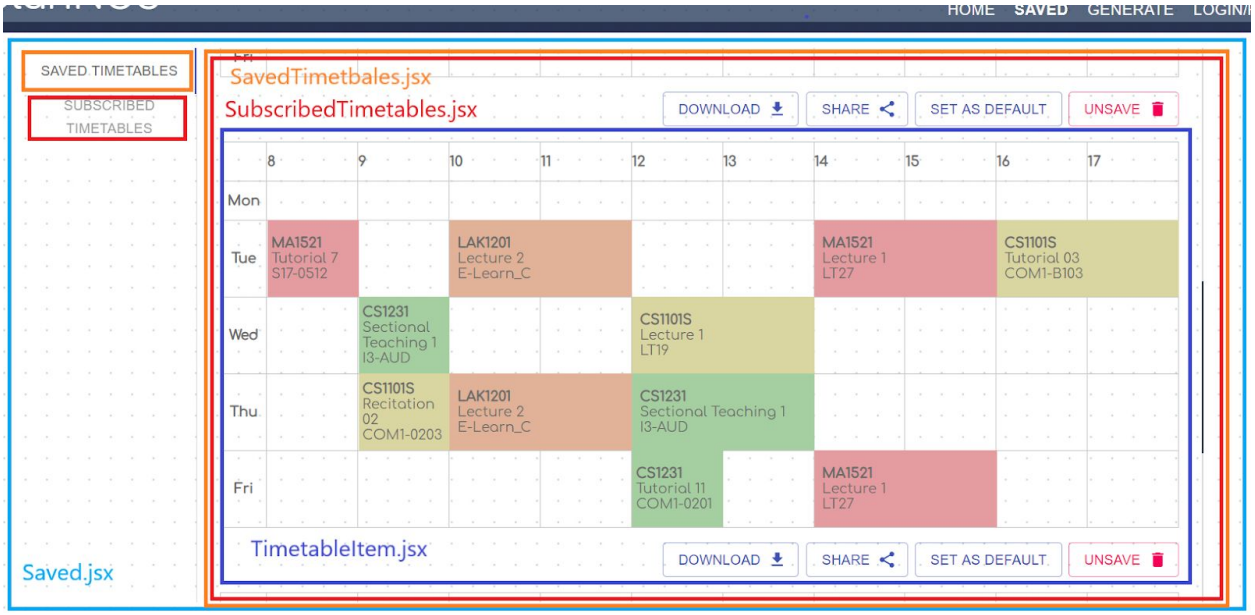

Figure 23: Breakdown of Saved.jsx

Login.jsx

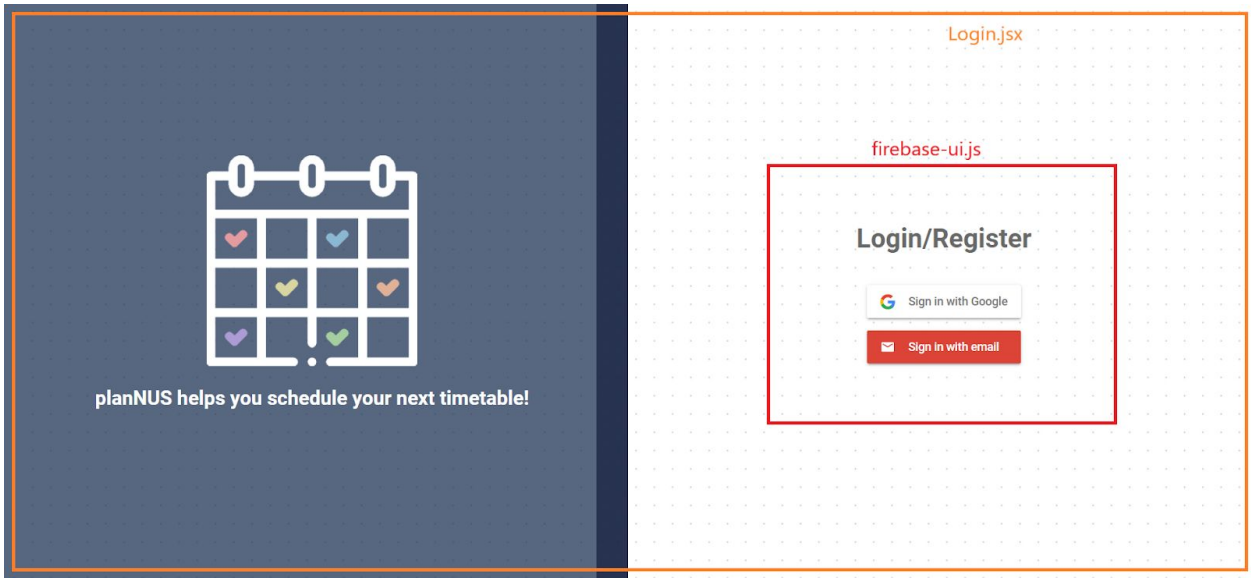

Figure 24: Breakdown of Login.jsx

Account.jsx

Figure 25: Account.jsx

Application layer

Component architecture

The application layer consists of components that support the logic of the PlanNUS application. Functionality of these components is provided by Firebase Cloud Functions, a serverless framework provided by Google. It deploys JavaScript code on Google servers in a Node.js environment and allows the backend algorithms to automatically respond to HTTPS requests (via API calls) or events triggered in other Firebase features.

The Cloud Functions below are all callable Cloud Functions, meaning that they are called directly from within the app, and can return a JavaScript object directly. Also, when the Cloud Function is called, the user's identity is already verified and the user's ID is available to each Cloud Function implicitly.

When a Cloud Function is deployed, it is independently uploaded onto Google servers. Every function runs in isolation, with its own environment and its own configuration. As such, we shall consider each Cloud Function as a separate component.

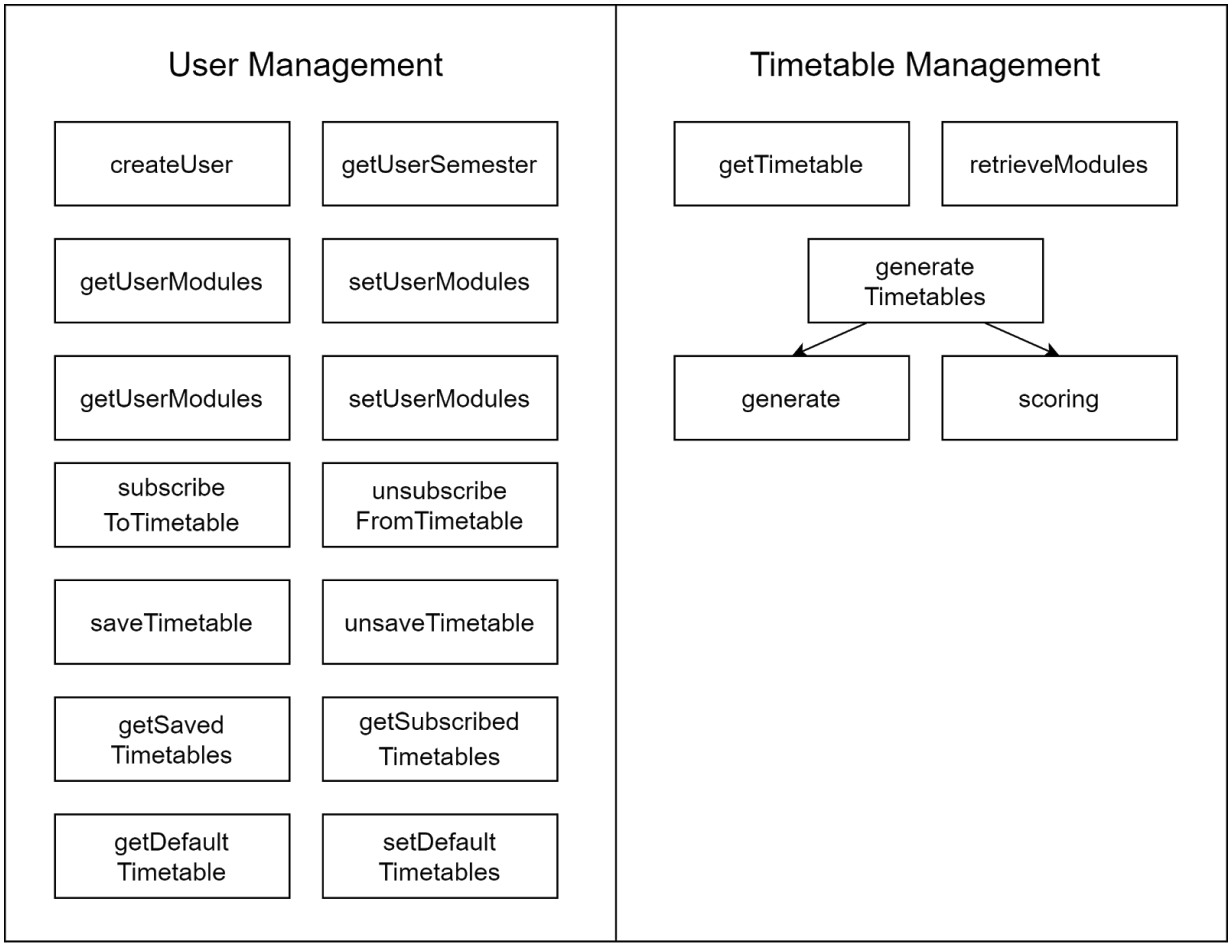

Figure 26: Application layer architecture

As seen in the architecture diagram above, we have organised all components in the application layer into two packages, each providing a different set of functionality. By keeping these components in separate packages, we intend to fulfil the software design principle of **Separation of Concerns**:

1. **User Management package**: Comprises components that handle user-related queries, including reading and updating of user-related data.
2. **Timetable Management package**: Comprises components that perform timetable-related queries that do not apply to a specific user, including the timetable generation and timetable scoring algorithms.

User Management package

This package manages the data related to an individual user, and provides getter and setter APIs to the front end for this data. The following is a summary of the entities that this package manages, and the corresponding APIs it provides.

Entity	Getter APIs	Setter APIs
User record	*The APIs below are used instead*	`createUser` `deleteUser`
Year, semester	`getUserSemester`	*This is set via the* `retrieveModules` *Cloud Function*
Modules	`getUserModules`	`setUserModules`
Priorities	`getUserPriorities`	`setUserPriorities`
Saved timetables	`getSavedTimetables`	`saveTimetable` `unsaveTimetable`
Subscribed timetables	`getSubscribedTimetables`	`subscribeToTimetable` `unsubscribeFromTimetable`
Default timetable	`getDefaultTimetable`	`setDefaultTimetable`

Each API performs a specific function and hence fulfills the **Single Responsibility Principle**. The entities and corresponding getter and setter functions are described below:

User record

- This contains all the other entities in the table, and should be created before the other entities are set.
- `createUser`
    - Creates a record to contain a user's data.
    - Parameters: none
    - Return: `{ success: boolean }`
    - success is whether the user creation was successful.
- `deleteUser`
    - Deletes the record associated with the user.
    - Parameters: none
    - Return: `{ success: boolean }`

Year, semester

- These entities are used as default values in the timetable generation flow, and they correspond to the values which were last set or used to generate timetables.
- `getUserSemester`
    - Retrieves the year and semester last set or used to generate timetables.
    - Parameters: none
    - Return: `{ year: number; semester: number }`

Modules

- This entity contains modules used as default values. It is an array which contains the last set of modules that the user generated the timetables with.
- `getUserModules`
    - Parameters: none
    - Return: `{ modules: string[] }`
- `setUserModules`
    - Parameters: `{ modules: string[] }`
    - Return: `{ success: boolean }`

Priorities

- Similarly, the priorities are the last set of priorities that the user generated the timetables with.
- `getUserPriorities`
    - Parameters: none
    - Return: `{ priorities: Priority[] }`
- `setUserPriorities`
    - Parameters: `{ priorities: Priority[] }`
    - Return: `{ success: boolean }`
- Priority is a plain object as described in the "JavaScript object interface" section.

While implementing the presentation and application layer, we came across this design decision regarding which Cloud Function the default values for the year, semester, modules and priorities should be saved.

<table>
<tr><td colspan="4">Design decision: How should the default values for the user be saved?</td></tr>
<tr><td>Problem</td><td colspan="3">The year, semester, modules, and priorities can be saved to the database in different Cloud Functions.</td></tr>
<tr><td>Options</td><td colspan="3">Option 1: <code>setUserModules</code>/<code>Priorities</code> Cloud Functions
Option 2: <code>retrieveModules</code>/<code>generateTimetables</code> Cloud Functions</td></tr>
<tr><td rowspan="3">Comparing options</td><td>Criteria</td><td>Option 1</td><td>Option 2</td></tr>
<tr><td>API design</td><td>APIs are easier to understand</td><td>Having a get API but no set might be confusing</td></tr>
<tr><td>Response time</td><td>Requires two Cloud Function calls e.g. <code>setUserSemester</code> then <code>retrieveModules</code> which is much slower.</td><td>Requires only one Cloud Function call e.g. <code>retrieveModules</code>.</td></tr>
<tr><td>Decision</td><td colspan="3">We chose Option 2, as it is significantly faster and improves the user experience. Also, the APIs are not too hard to understand with documentation.</td></tr>
</table>

Additionally, the year and semester are stored and handled together because they change together.

Timetable Management package

The Timetable Management package consists of components (Cloud Functions) that are responsible for handling timetable-related queries that are not related to any specific user. The following table provides a summary of these APIs:

Functions	Parameters	Returns
`getTimetable`	`timetableId`	The corresponding timetable object from the Timetables database
`retrieveModules`	`year` `semester`	The listing of the available modules for the given semester
`generateTimetables`	`priorities` `modules`	A list of timetables with the given modules scheduled and scored with the given priorities

`scoring`	`priorities`
	`timetable`
	`minMaxValues`
		The score of the given `timetable`, based on the given set of `priorities` and `minMaxValues` (out of 100 percent)

These APIs perform the work using the other components in this package. In particular, the `generateTimetables` API employs the **Facade structural pattern**. Instead of interacting with the `generate` and `scoring` sub-components, the frontend calls the `generateTimetables` facade, making these subsystems easier to use. The following diagram showcases this Facade pattern:

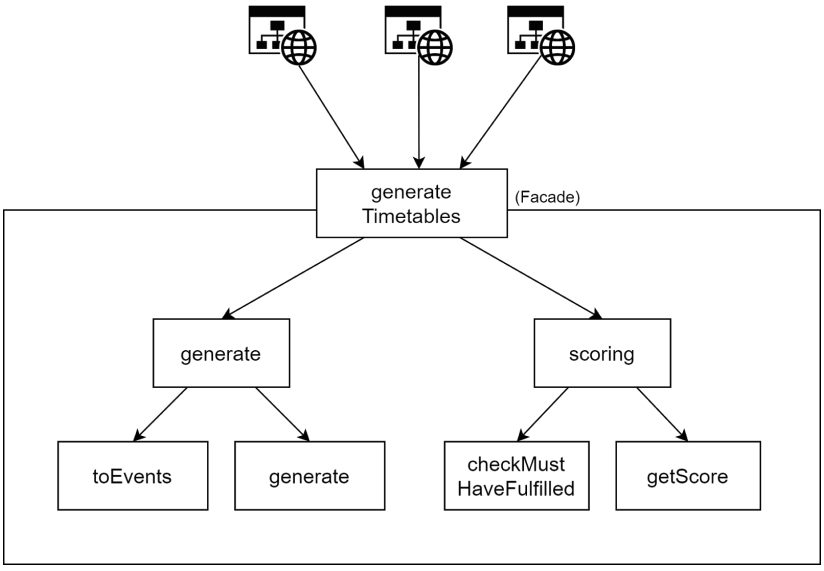

Figure 27: Facade pattern

The components in the Timetable Management package handle the three functionalities listed below. Take note that timetable generation and timetable scoring have been abstracted into separate functions, given that they have linked but different responsibilities. This fulfills the **Single Responsibility Principle.**

1. Timetable retrieval

Every saved timetable is automatically assigned a unique `timetableID` using a UUID v4 generator. Given so, every timetable that has been saved can be retrieved using its unique `timetableID`. The backend provides the frontend with the `getTimetable(timetableId)` API in order to send a read query to the database.

The API returns the corresponding Timetable object, following a **Data Transfer Object pattern**, as mentioned in the "JavaScript object interface" section.

2. Timetable generation

This module is concerned with the creation of non-clashing timetables. It exports one function, `generate`, which takes in the module data, and the year and semester to schedule for. It calls on the other helper functions in the module and returns the timetables created by permuting the module data.

```
exports.generate = (moduleData, year, semester) => {
    const moduleEvents = toEvents(moduleData, semester);
    const eventList = product(moduleEvents);
    return eventList.map(t => ({
        year: year,
        semester: semester,
        modules: moduleData.map(m => m.moduleCode),
        events: t
    }));
}
```

(Input: Array of module data from NUSMods API[1])

Firstly, it calls the helper function `toEvents(moduleData, semester)` to convert the input from nusmods into a format closer to our internal timetable format which is more convenient for permuting the events later. This reduces **rigidity**, which is important since the data comes from NUSMods, and changes could be necessary in the future to adapt to other data sources. If the source of the `moduleData` changes, only this function needs to be modified.
This function returns:

`ScheduleEvent[]`

[1] NUSMods API. (n.d.). Retrieved from https://api.nusmods.com/v2/

`ScheduleEvent` is an object containing events to be scheduled. Each event to be scheduled must have one of the choices below appearing in the final timetable. The structure is as follows:

`{ moduleCode: string; lessonType: string; choices: ScheduleChoice[] }`
Example: `moduleCode` could be “CS3219”, `lessonType` could be “Tutorial”
`ScheduleChoice` is one of many choices that could be made for the `ScheduleEvent`. It contains the following:

`{ classNo: string; lessons: ScheduleLesson[] }`
Example: classNo could be “4” corresponding to tutorial 4
The `ScheduleLesson` array denotes all the classes that have to be attended for this choice. Most tutorials only have one `ScheduleLesson`, but some lectures have more than one class that has to be attended. For example, some modules could have lectures on either Tuesday and Friday or Monday and Thursday, for instance. The structure of a `ScheduleLesson` is as follows:

```
{
    location: string;
    day: number;
    startTime: number;
    endTime: number;
    weeks: number[]
}
```
Weeks is an array containing the weeks that this lesson is held. For example, lectures could have the numbers [1, 2, …, 13] for all weeks, whereas a tutorial could be held on [3, 4, …, 13].

Up till this point, the `generate` function has called `toEvents` and obtained an array. Next, it takes this array and passes it to the `product(moduleEvents)` function, which takes all the events that need to be scheduled and creates possible timetables which the events do not clash with each other. This function returns another array of the possibilities. This is structured as follows:

`TimetableEventList[]`
This `TimetableEventList` corresponds to a scheduled list of events, ready to be inserted into a timetable. It is also an array as seen below:

`TimetableEvent[]`
Each `TimetableEvent` is a class that has to be attended if the timetable is followed. This is described in the JavaScript object interface section, but is reproduced here for easy reference. It is structured as follows:

```
{
    moduleCode: string;
    lessonType: string;
    classNo: string;
    location: string;
    day: number;
    startTime: number;
    endTime: number;
    weeks: number[]
}
```

Note that this is a combination of fields from the above objects from `toEvents`.

The result is obtained and a header is added to each element to form a timetable, as can be seen from the code fragment at the start of this section. This list of timetables is then returned.

3. Timetable scoring

The timetable scoring is performed for each timetable via the `scoring(priorities, timetable, minMaxValues)` function:

1. `priorities`: the array of priorities received from the frontend
2. `timetable`: the timetable object received from the timetable generation component
3. `minMaxValues`: an object containing 5 specific values that are important in ensuring the accuracy of the scoring algorithm

The timetable scoring algorithm maintains a weighted point system, based on how the user has ranked his/her selected priorities. For every priority that has been selected by the user, the score is given by the following formula:

$$timetableScore = \sum_{priority} (scoreMultiplier_{priority} \times maxScore_{priority})$$

The equation has two parameters:

1. $maxScore_{priority}$ is the weight that is given to each priority based on the ranking that the user has assigned to it. It is an integer and its value corresponds to the maximum possible number of points that can be given to a given priority. It is calculated as such: $maxScore_{priority} = numberOfPriorities - rank_{priority} + 1$

2. $scoreMultiplier_{priority}$ is a fraction value between 0 and 1. It represents the extent to which the given priority has been fulfilled in the given timetable. A value of 1 signifies that the priority has been completely fulfilled, while a value of 0 signifies that the priority has been completely unfulfilled.

While the calculation of $maxScore_{priority}$ is simple, the calculation of $scoreMultiplier_{priority}$ is more complex. We recognised that different priority types require a different approach in order to evaluate the extent to which it has been fulfilled. For example, the priority *"Have a maximum number of free days"* would require a different way of evaluation from the priority *"Avoid lessons before (time) every day."*

Given that, we decided to abstract out the calculation of $scoreMultiplier_{priority}$ into individual functions, and maintain a single `getScore(count, priority, timetable, minMaxValues)` function that routes to the individual functions according to the priority type. The following code fragment is the code implementation of the routing function. Take note of the use of switch-case and the abstraction out to the respective multiplier-calculation functions, emphasised in bold.

```
function getScore(count, priority, timetable, minMaxValues) {
   let multiplier = 0;
   const rank = priority.rank;
   switch(priority.type) {
       case "AvoidAfterPriority":
           const afterTime = priority.fields.time;
           multiplier = avoidLessonsAfterPriorityMultiplier(afterTime,
timetable);
           break;
       case "AvoidBeforePriority":
           const beforeTime = priority.fields.time;
           multiplier = avoidLessonsBeforePriorityMultiplier(beforeTime,
timetable);
           break;
       case "FreePeriodPriority":
           const fromTime = priority.fields.fromTime;
           const toTime = priority.fields.toTime;
           multiplier = freePeriodPriorityMultiplier(fromTime, toTime,
timetable);
           break;
       case "LunchBreakPriority":
           const hours = priority.fields.hours;
           multiplier = lunchBreakPriorityMultiplier(hours, timetable);
           break;
       case "MaxFreeDaysPriority":
           const maxPossibleFreeDays = minMaxValues.maxPossibleFreeDays;
           multiplier = maxFreeDaysPriorityMultiplier(timetable,
maxPossibleFreeDays);
           break;
       case "MinBreaksPriority":
           const minPossibleHours = minMaxValues.minPossibleHours;
           const maxPossibleHours = minMaxValues.maxPossibleHours;
           multiplier = minBreaksPriorityMultiplier(timetable,
minPossibleHours, maxPossibleHours);
           break;
       case "MinTravellingPriority":
           const minPossibleDist = minMaxValues.minPossibleDist;
           const maxPossibleDist = minMaxValues.minPossibleDist;
           multiplier = minTravellingPriorityMultiplier(timetable,
minPossibleDist, maxPossibleDist);
   }
   return calculateScore(count, rank, multiplier);
}
```

Such an implementation follows a **Strategy pattern**. Each abstracted multiplier-calculation can be considered a concrete implementation of the multiplier calculation that is isolated from the other “strategies”.

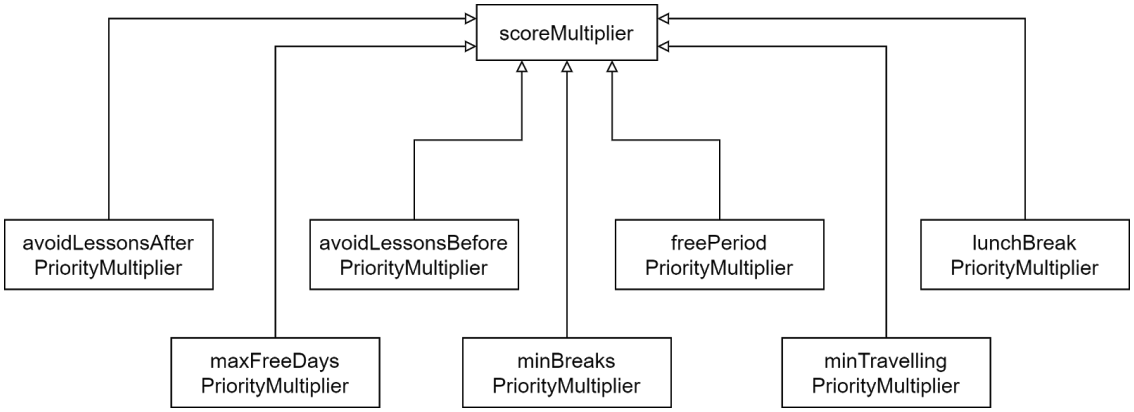

Figure 28: Strategy pattern

In addition, it adheres to a number of software design principles:

1. **Single Responsibility Principle:** Each abstracted function is responsible for the calculation of $scoreMultiplier_{priority}$ for one type of priority. This encourages decoupling within the Timetable Scoring component. In addition, each function contains more abstracted logic. For example, under `minTravellingPriorityMultiplier`, there exists an abstracted-out function `GetDistanceBetween(start, dest)` that serves to only retrieve the geographical distance between two locations, and nothing else.

2. **Open-Closed Principle:** By using the Strategy pattern, we keep the current multiplier-calculation implementations closed to modification by abstracting them out to other functions. In addition, the Strategy pattern encourages extension of functionality. New priorities can be added by simply introducing a new case along with a new multiplier-calculation function.

4. Module data

The Timetable Management package utilizes the NUSMods API for the data about the modules. All these requests are handled by this module. While writing this module, we faced the following design decision:

<table>
<tr><td colspan="4">Design decision: Should the data obtained from the NUSMods API be cached?</td></tr>
<tr><td>Problem</td><td colspan="3">Module data can be retrieved from NUSMods directly or cached for a period of time to service requests from the other functions in the Timetable Management package.</td></tr>
<tr><td>Options</td><td colspan="3">Option 1: Retrieve module data whenever it is needed.
Option 2: Cache module data for a period of time and serve requests from this cache within this period of time.</td></tr>
<tr><td rowspan="4">Comparing options</td><td>Criteria</td><td>Option 1</td><td>Option 2</td></tr>
<tr><td>Data staleness</td><td>The newest data is always used to generate timetables.</td><td>Sometimes, NUSMods may have updated the module data but the old data is still being used.</td></tr>
<tr><td>Request speed</td><td>Every time timetables are generated, all the modules involved have to be fetched from the NUSMods servers, even if they were only just used a couple of seconds ago.</td><td>A copy of the module data is fetched from a local database, which has a much lower latency. This is especially useful for the listing of modules, which all users need when generating timetables.</td></tr>
<tr><td>Scalability</td><td>This makes our application dependent on the speed of an external API. With a large amount of users, this does not scale, and is also not polite towards NUSMods.</td><td>This option reduces the dependency and is more scalable.</td></tr>
<tr><td>Decision</td><td colspan="3">We chose Option 2, as this makes the response time significantly faster, making for a better user experience. The timetable data also does not change that frequently, which does not justify querying the NUSMods API all the time, and a tradeoff can be made via the time period that the data is cached.</td></tr>
</table>

Database layer

Data in Cloud Firestore is organised as collections and documents. Collections contain documents, and a document is not unlike a JavaScript object. In addition, a document can contain other collections, but we do not use this feature.

Component architecture

Our data is stored with the following structure:

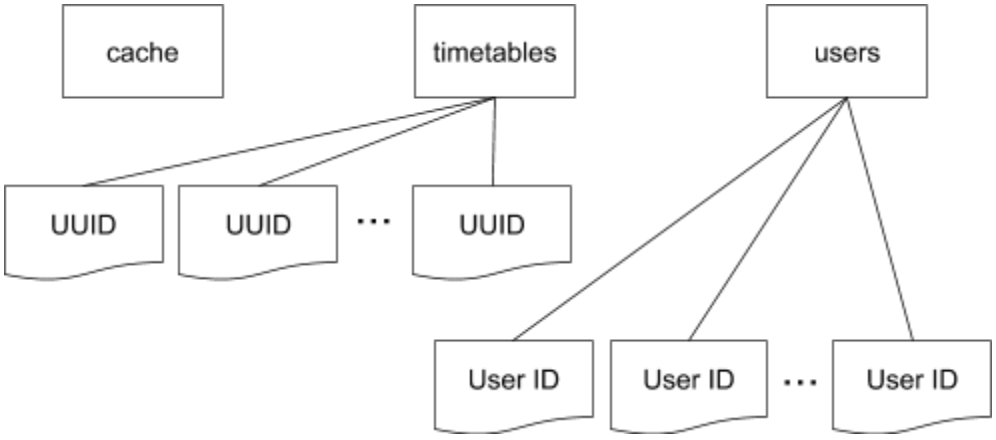

Figure 29: Database layer architecture

1. **User collection:** This collection contains data related to specific users.
2. **Timetables collection:** This collection contains timetable data that is not related to any specific user.
3. **Module cache collection:** This collection contains a cache of module data retrieved via the NUSMods API.

While thinking of how to design the database to store user and timetable data, we came up with two options. Our design decision process is summarised in the following table:

<table>
<tr><td colspan="4">Design decision: How to organise users and timetables in the database?</td></tr>
<tr><td>Problem</td><td colspan="3">User information and timetable information are two important entities to store in the database. However, the manner in which they are stored may affect query performance or the functionality of the PlanNUS application.</td></tr>
<tr><td>Options</td><td colspan="3">Option 1: Store timetables in the Users collection.
Option 2: Store timetables in a separate Timetables collection.</td></tr>
<tr><td rowspan="3">Comparing options</td><td>Criteria</td><td>Option 1</td><td>Option 2</td></tr>
<tr><td>Ease of implementation</td><td>There will only be a need to query into a single Users collection.</td><td>Clear boundaries must be drawn between what is considered “user data” and “timetable data”.</td></tr>
<tr><td>Coupling</td><td>Having timetables stored in the Users collection would mean that all timetables should “belong” to a specific user. This may limit certain functionality in the application.</td><td>By storing the timetables in a separate Timetables collection, timetables need not be directly linked to a specific user. However, there must be a means to link a specific user to a specific timetable.</td></tr>
<tr><td>Decision</td><td colspan="3">We chose Option 2, because we wanted to maintain a low level of coupling between the two entities. In order to draw links between specific users and specific timetables, we made use of the unique <code>timetableId</code> that is generated for every timetable in the database as a reference code to store in the Users collection. This allowed us to maintain low coupling while maintaining the functionality of associating users with timetables.

As mentioned in the Firebase documentation,[2] as long as the data to be stored is not hierarchical in nature, storing data in root-level collections (i.e. storing Users and Timetables as two separate root-level collections) is good for many-to-many relationships.</td></tr>
</table>

The chosen database design adheres to the **Single Responsibility Principle**. As described above, there exists two separate database collections for storage of Users as well as storage of Timetables. By keeping them separate, we keep them loosely coupled and reduce the dependencies between user-related operations and timetable-related operations. Given that, irregular behaviour in one collection should be less likely to affect behaviour in the other collection.

[2] Choose a data structure | Firebase. (n.d.). Retrieved from https://firebase.google.com/docs/firestore/manage-data/structure-data

Users collection

The following screen capture from the Firebase console gives an overview of the database design for the Users collection:

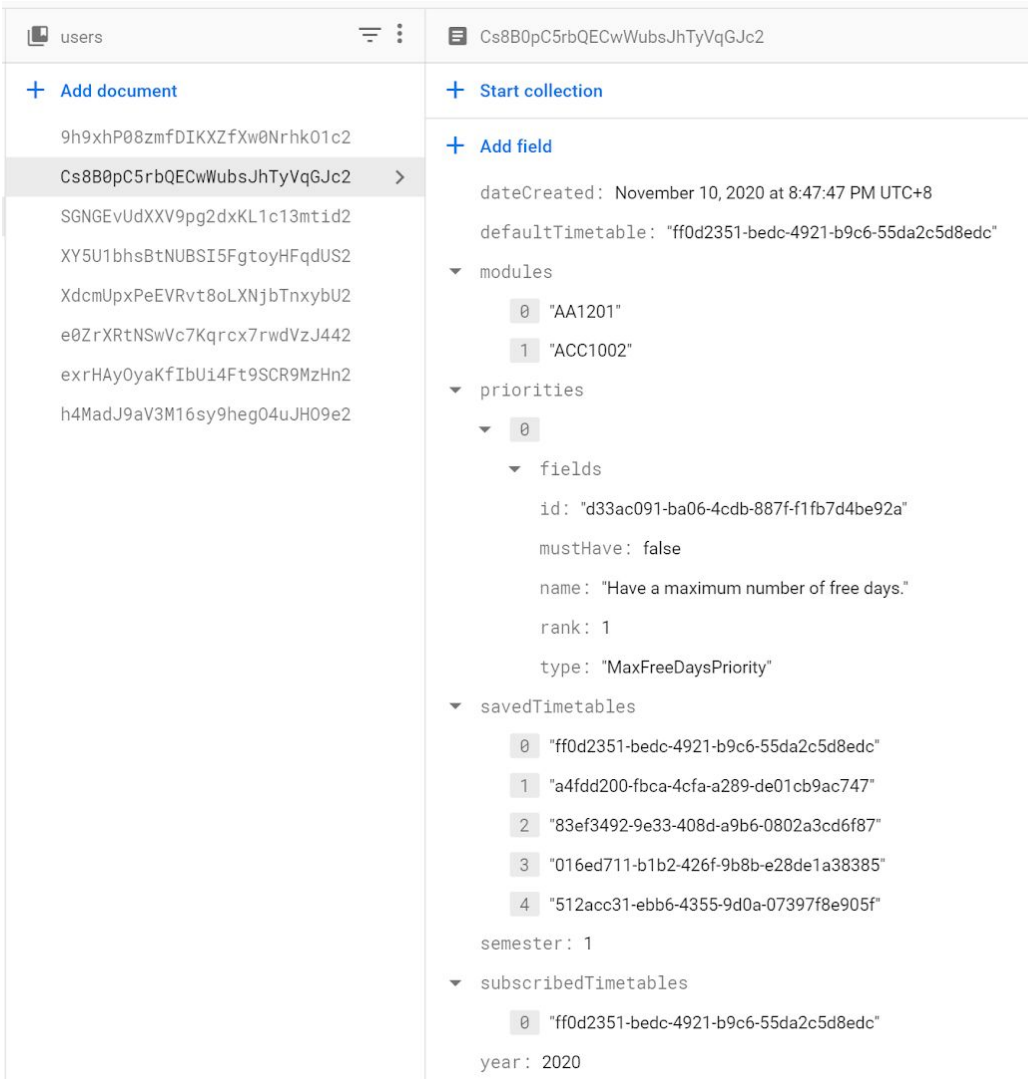

Figure 30: Users collection

The left panel is a list of user authentication IDs. Every user (registered or not) is assigned a unique user authentication ID by Firebase. The database retrieves this user authentication ID for a current user and creates a new document in the Users collection for the user. This operation is performed when the frontend calls the `createUser` API, instructing the Application layer to send a create request in the Database layer.

The following table describes the fields in the document in the Users collection:

Field	Data type	Details
`dateCreated`	Timestamp	The date and time on which the user account was created
`year`	Integer	The latest academic year and semester selected by the user in the Generate tool
`semester`	Integer	
`modules`	Array of string	The latest set of modules selected by the user in the Generate tool
`priorities`	Array of Priority objects	The latest set of priorities selected by the user in the Generate tool
`defaultTimetable`	String	The `timetableId` of the user's default timetable
`savedTimetables`	Array of string	The list of `timetableId`s of every timetable that the user has saved
`subscribedTimetables`	Array of string	The list of `timetableId`s of every timetable that the user has subscribed to

As previously mentioned, the `timetableId`s are used to associate users in the Users collection and timetables in the Timetables collection.

Timetables collection

The following screen capture from the Firebase console gives an overview of the database design for the Users collection:

Figure 31: Timetables collection

The left panel is a list of timetable IDs. Each timetable ID is uniquely assigned in the Application layer whenever a timetable is saved (i.e. assigned in the `saveTimetable` Cloud Function).

The following table describes the fields in the document in the Timetables collection:

Field	Data type	Details
`dateModified`	Timestamp	The date and time on which the timetable was last modified
`owner`	String	The ID of the user who has saved this timetable
`timetable`	Timetable object	Represents all data related to this specific timetable
`timetableId`	String	The unique identifier generated for the timetable

Module cache collection

The following screen capture from the Firebase console gives an overview of the database design for the Module Cache collection:

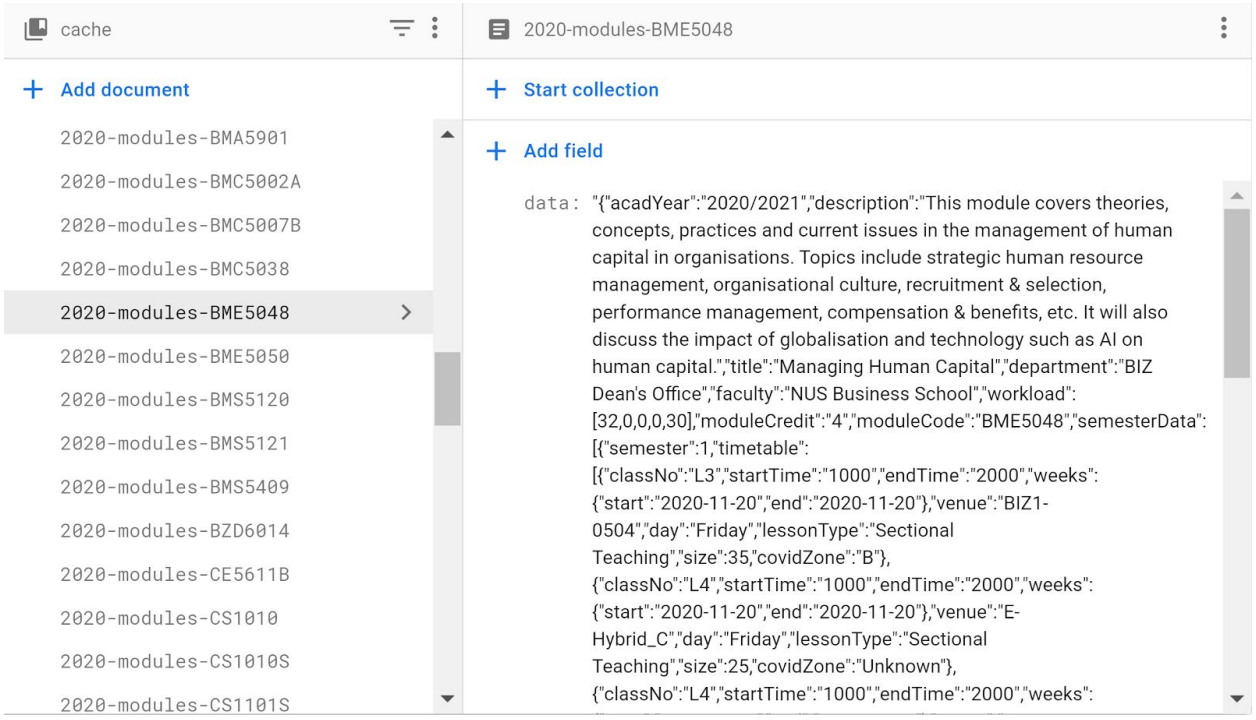

Figure 32: Module cache collection

The left panel is a list of IDs signifying the academic year and module. The following table describes the fields in the document in the Timetables collection:

Field	Data type	Details
`data`	String	A copy of the module data retrieved from NUSMods API
`updated`	Timestamp	The date and time on which the module data was last updated

This collection is manipulated directly by the module data component in the Timetable Management package. It contains a cache of the module list and class timings for modules as a string which was requested from the NUSMods API.

Program flow

The following activity diagrams describe the flow that users experience for each use case:

Registering and signing in

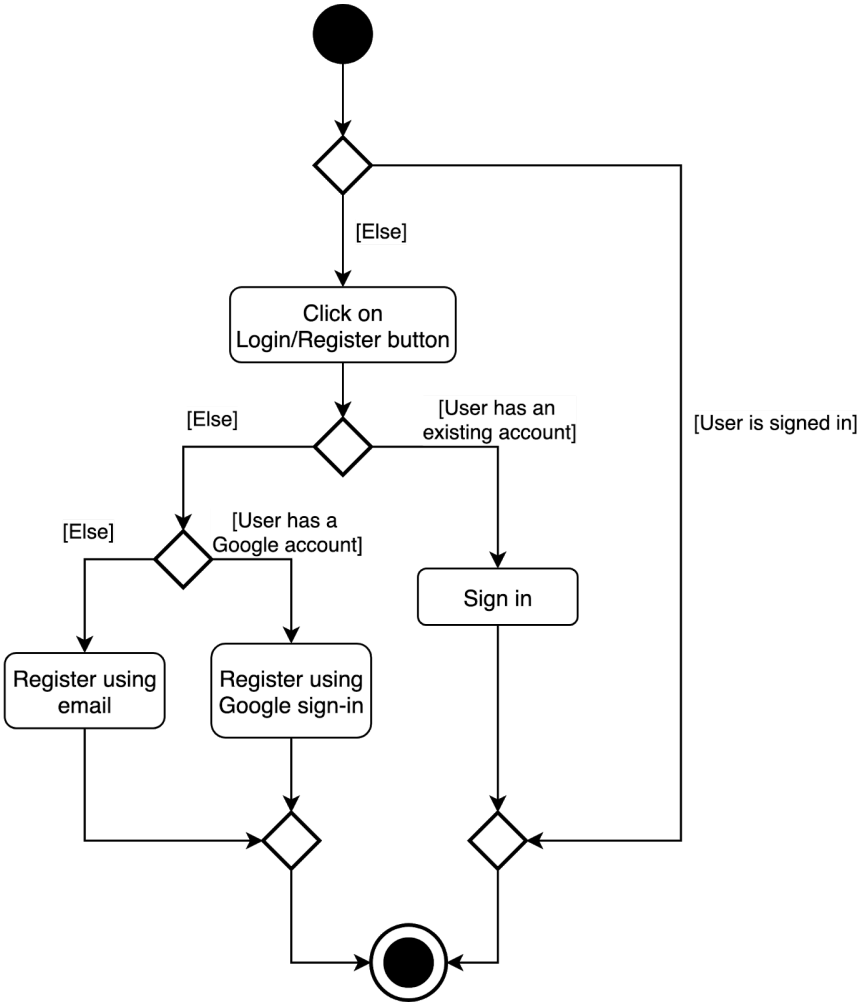

Figure 33: Activity diagram for registration and signing in

Generating a new timetable

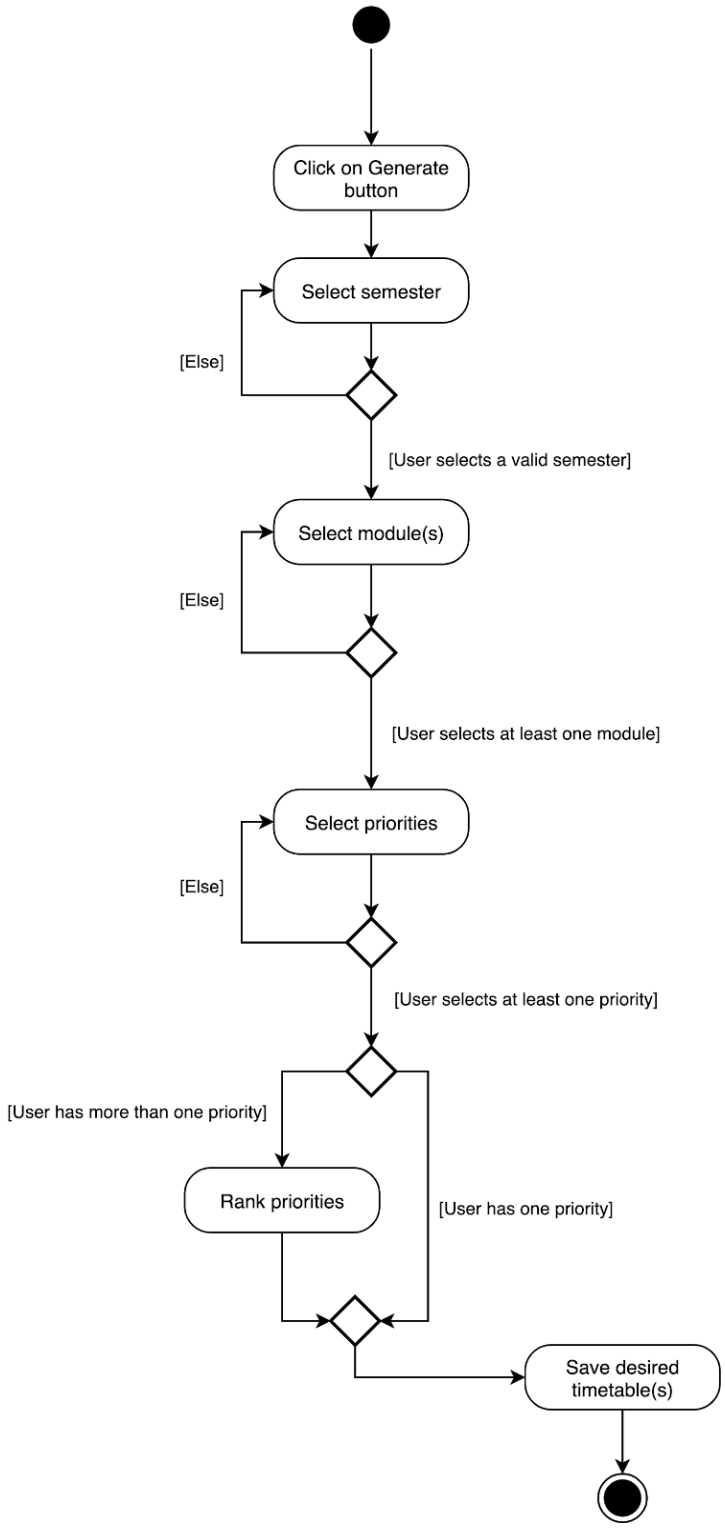

Figure 34: Activity diagram for generating timetable

Sharing a timetable

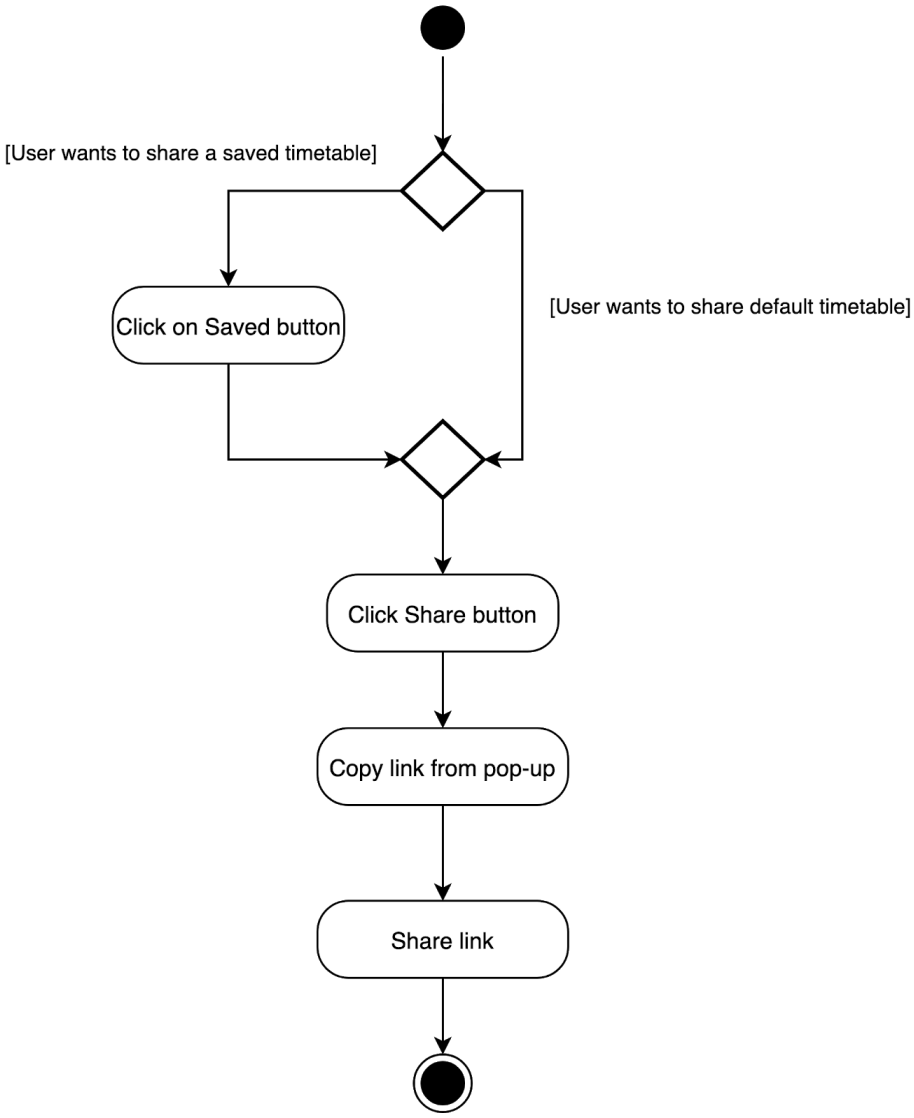

Figure 35: Activity diagram for sharing timetable

Component interaction

The following sequence diagram gives an overview of the interaction between the components of the frontend and backend:

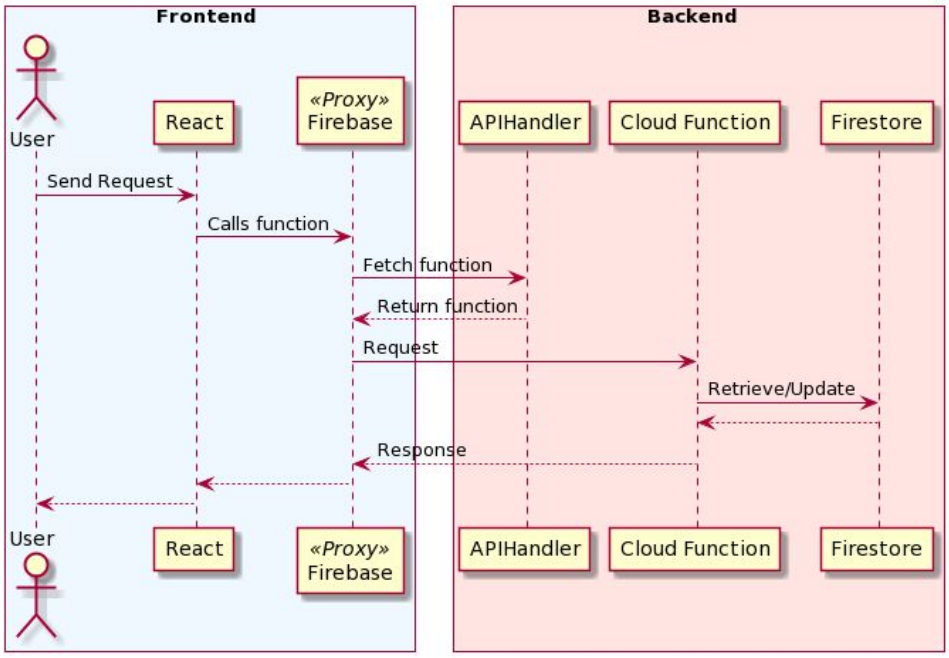

Figure 36: Sequence diagram for general component interaction

The following sequence diagrams describe the specific interactions for each use case:

Creating a new user

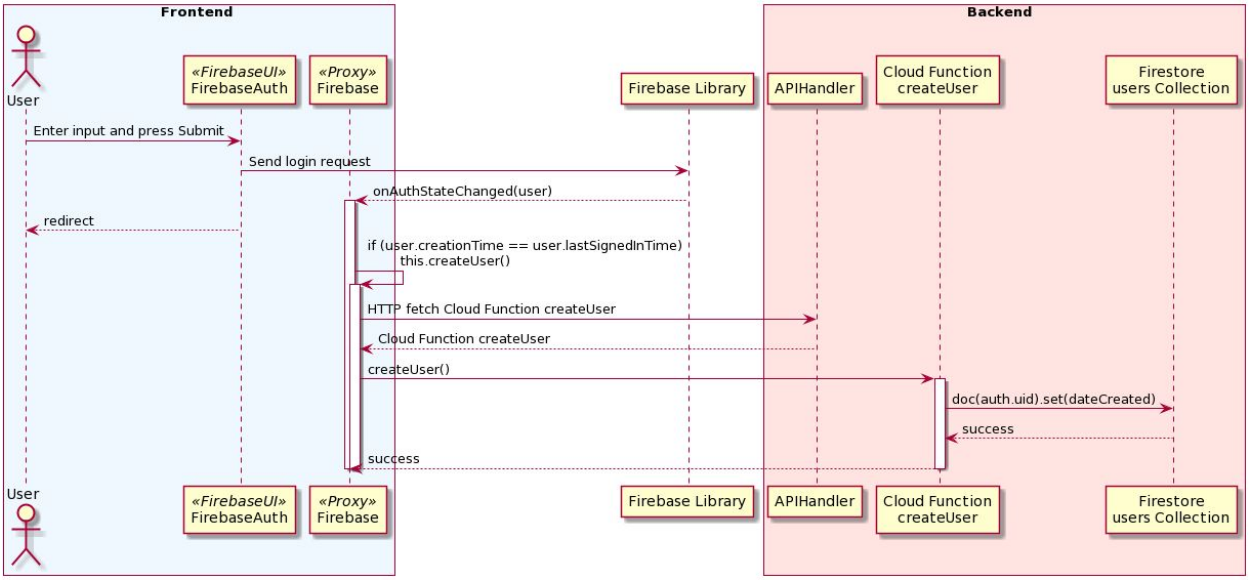

Figure 37: Sequence diagram for creating new user

Saving a timetable

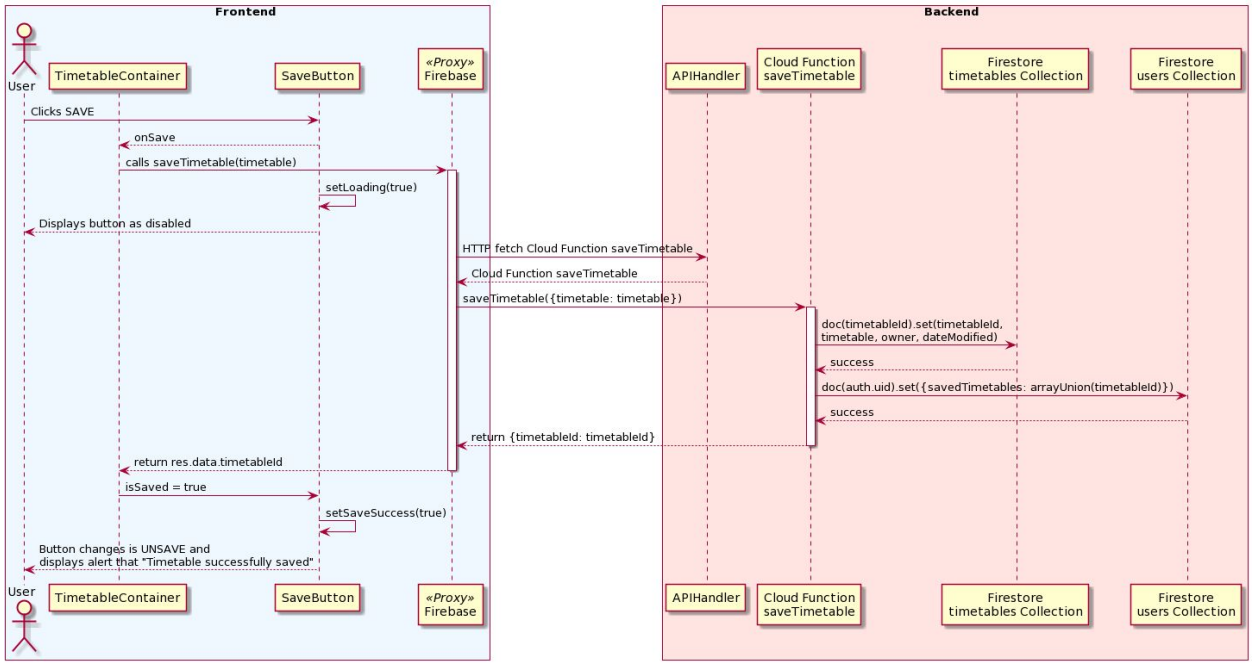

Figure 38: Sequence diagram for saving timetable

Subscribing to a timetable

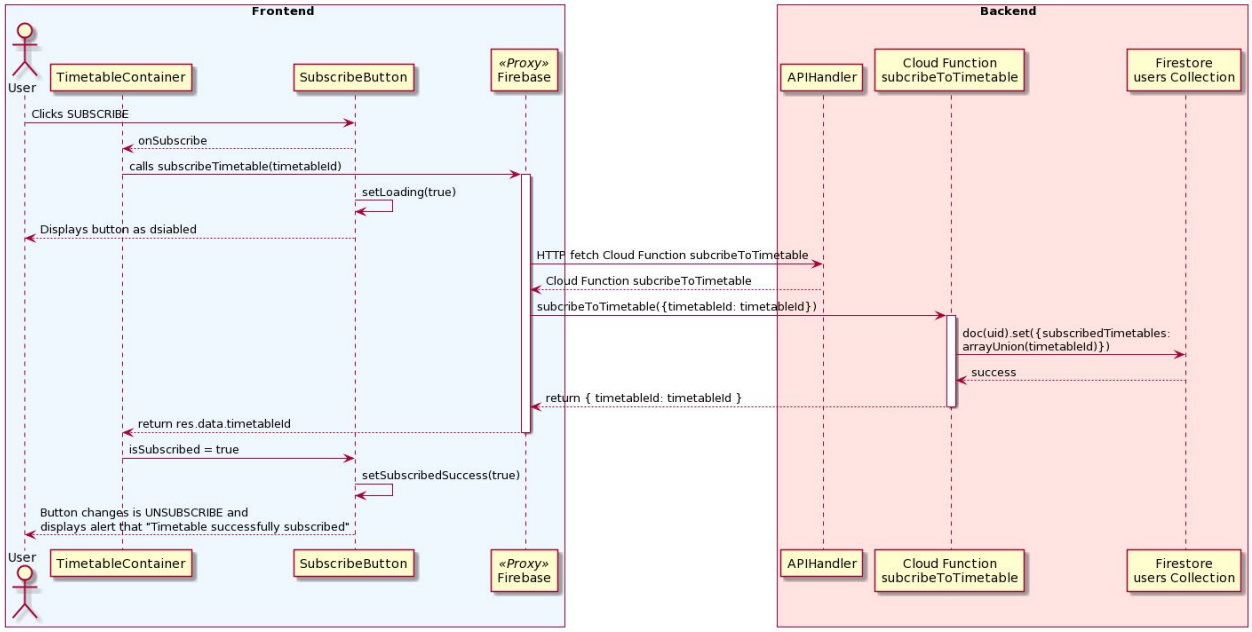

Figure 39: Sequence diagram for subscribing to timetable

Generating new timetables

1. Selecting year and semester

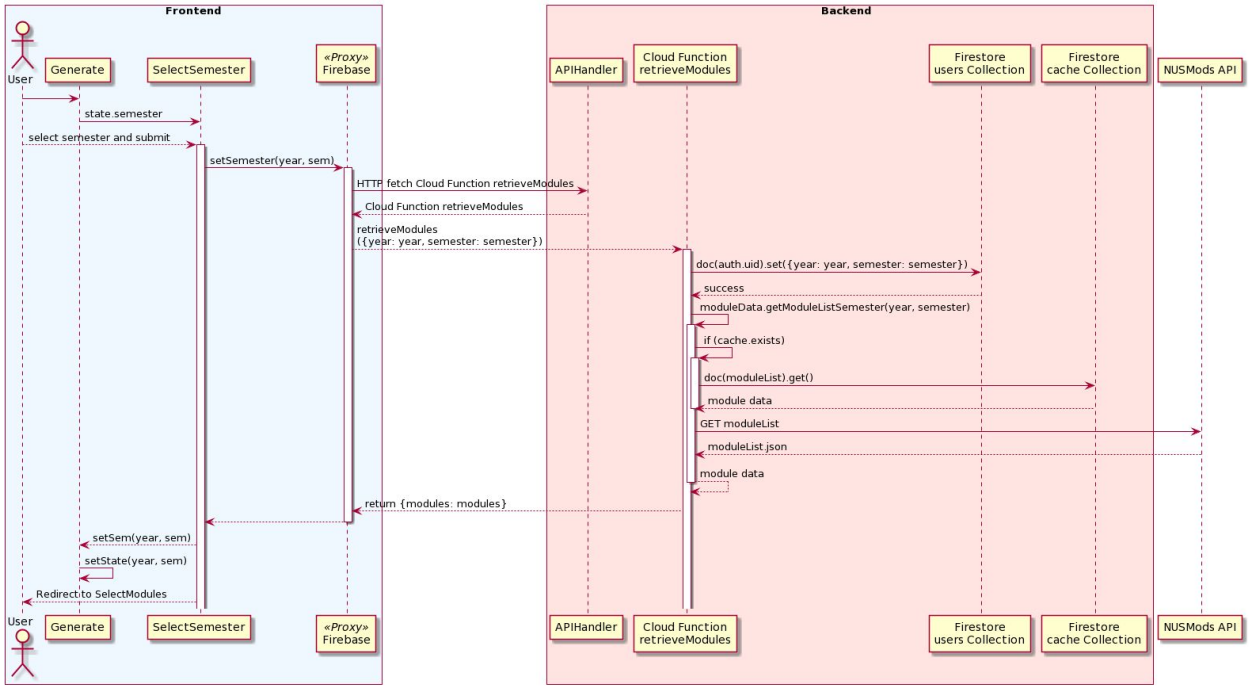

Figure 40: Sequence diagram for semester selection

2. Selecting modules

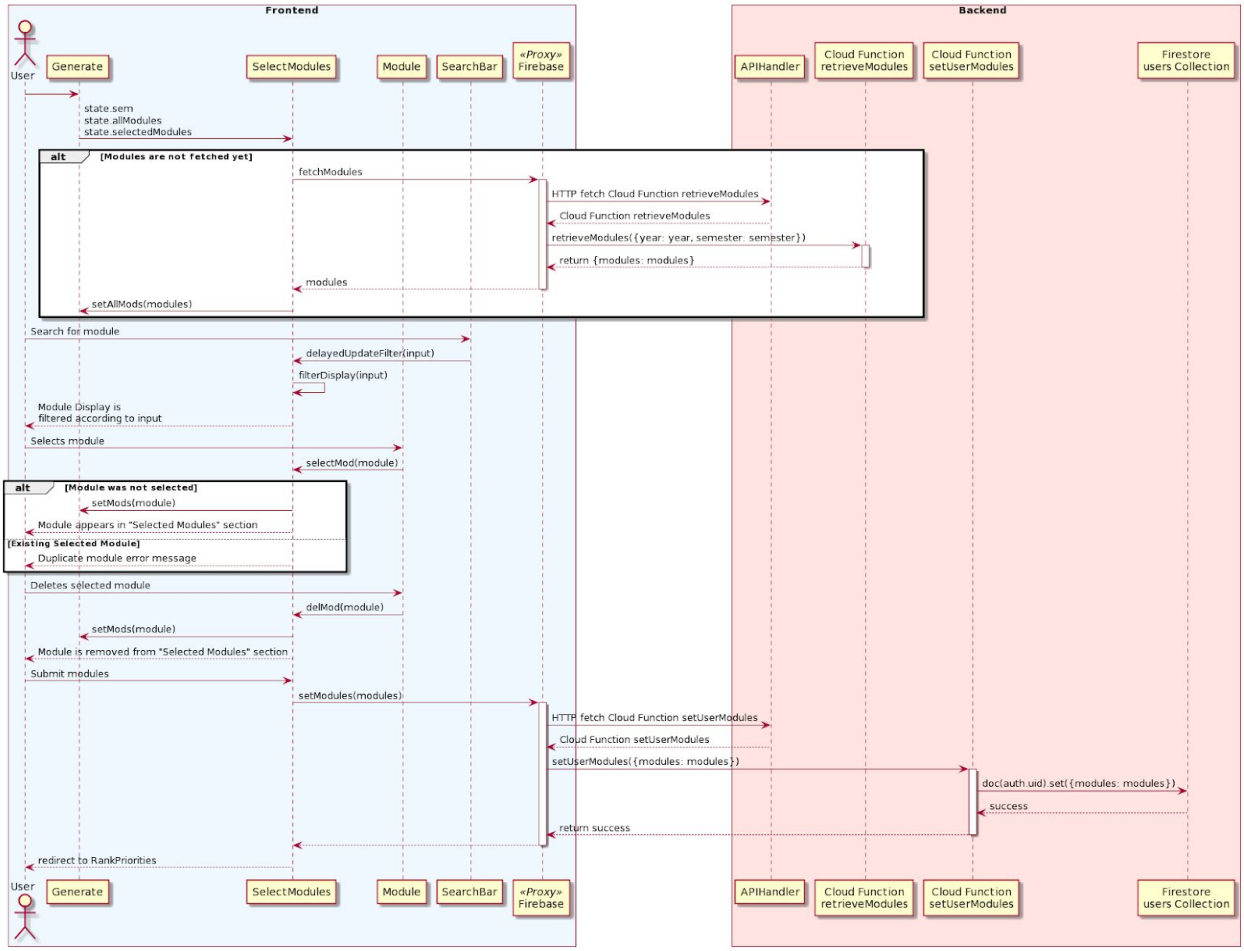

Figure 41: Sequence diagram for module selection

3. Selecting priorities

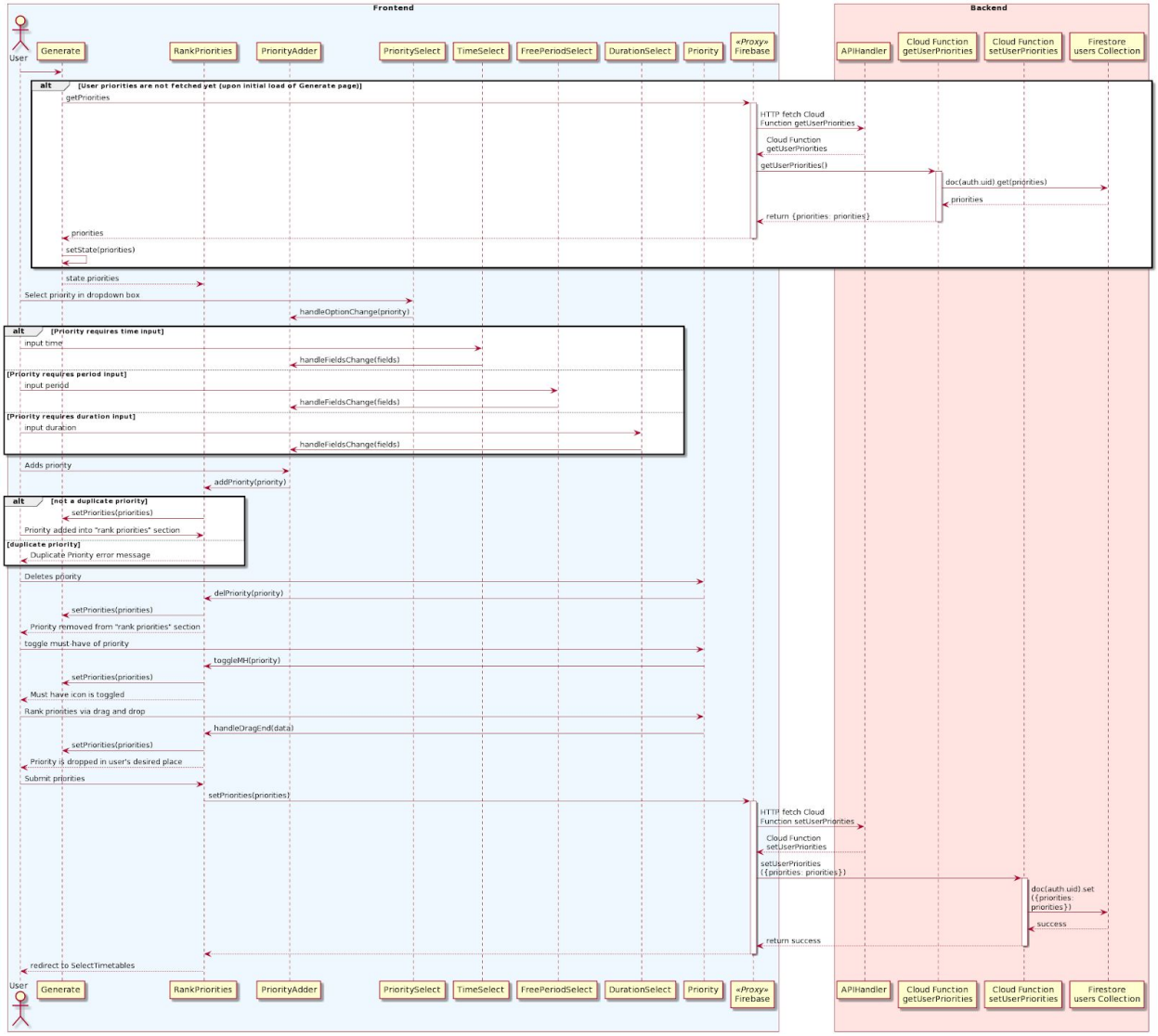

Figure 42: Sequence diagram for priority selection and ranking

4. Selecting timetables

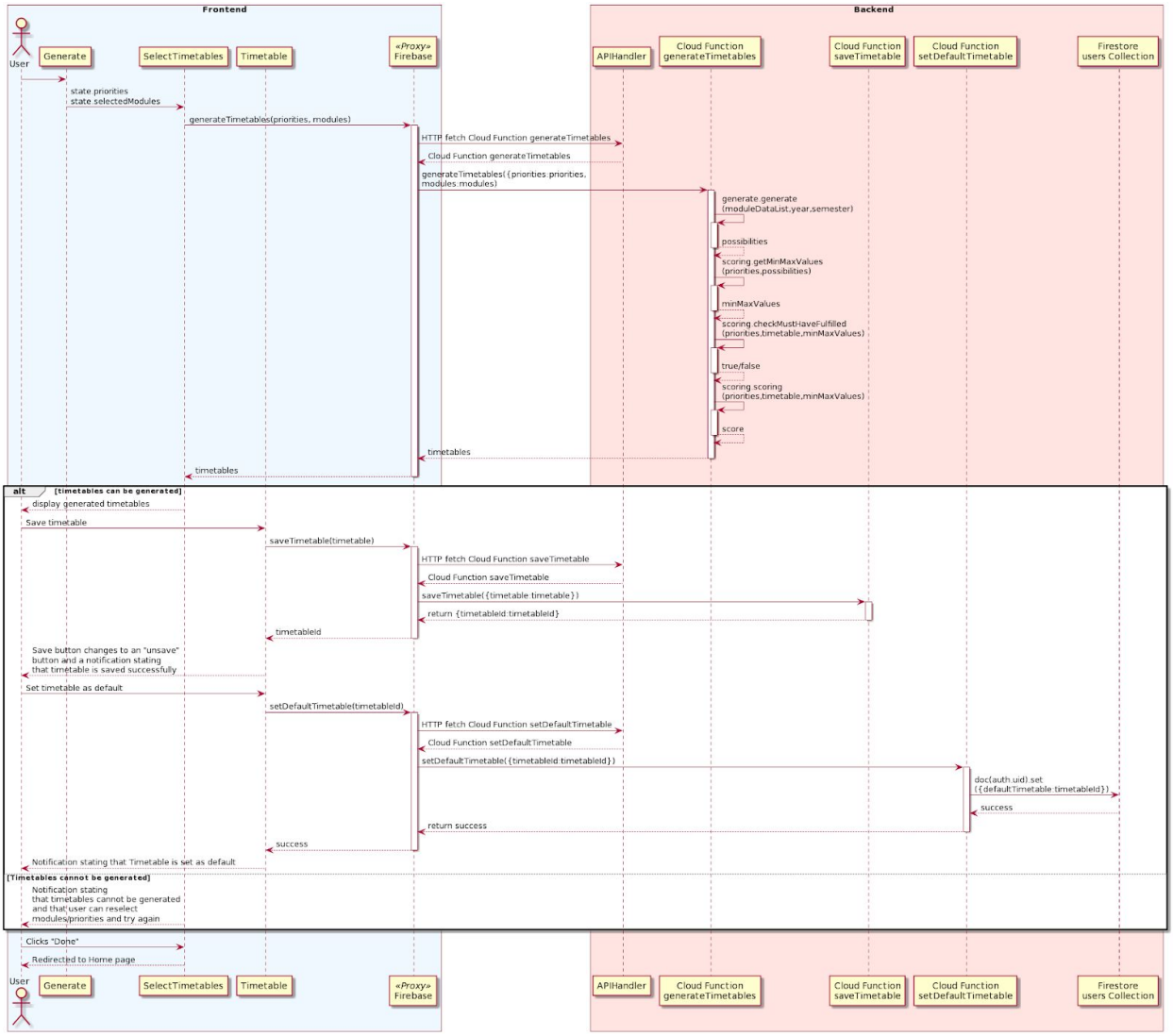

Figure 43: Sequence diagram for selection of generated timetables

Testing

Integration tests

Integration tests were done to ensure that the Application layer was well-connected to the Database layer. In particular, we strategically selected a few Firebase Cloud Functions and wrapped them with the `firebase-functions-test` package:

```
const test = require('firebase-functions-test')({
   databaseURL: 'https://plannus-cfd18.firebaseio.com',
   projectId: 'plannus-cfd18',
   storageBucket: 'plannus-cfd18.appspot.com',
}, '../../serviceAccountKey.json');
```

Using the Mocha test framework, we simulated the data and context that was to be passed to the wrapped Cloud Function:

```
const testUid = "testUser";
const docRef = admin.firestore().collection("users").doc(testUid);
const context = { auth: { uid: testUid } };
```

The Cloud Function then writes to the database (Cloud Firestore). The Node.js `assert` module is used to test for the database response. In addition, we check that the database contains the written data. After ensuring so, we clean up and delete the test data that was written to the database.

The following code fragment describes the test case for the `createUser` API:

```
describe("createUser()", () => {
   const wrapped = test.wrap(myFunctions.createUser);
   const delUser = () => docRef.delete();
   before(delUser);
   after(delUser);
   it("creates a user document", () => {
       return wrapped(undefined, context).then(result => {
           assert.deepEqual(result, { success: true })
           return docRef.get()
       }).then(doc => {
           assert.ok(doc.exists, "document does not exist");
           assert.ok(doc.data().hasOwnProperty("dateCreated"), "document
does not have property dateCreated");
           return true;
       });
   });
});
```

The following code fragment describes the test case for the `generateTimetables` API:

```
describe("generateTimetables()", () => {
   const wrapped = test.wrap(myFunctions.generateTimetables);
   const data = { modules: ["CS3203", "CS3219"], priorities: [] };
   const userDoc = { year: 2020, semester: 1 };
   beforeEach(() => docRef.set(userDoc));
   after(() => docRef.delete());
   it("checks for priorities", () => {
       return wrapped({ modules: ["CS3203", "CS3219"] }).then(result =>
           assert.deepEqual(result, { success: false })
       );
   });
   it("checks for year", () => {
       return docRef.set({}).then(() =>
           wrapped(data, context)
       ).then(result =>
           assert.deepEqual(result, { success: false })
       );
   });
   it("schedules timetables", () => {
       return wrapped(data, context).then(result => {
           assert.ok(result.success, "result success");
           assert.ok(result.hasOwnProperty("timetables"), "timetables
property does not exist");
           assert.ok(result.timetables.length > 0, "result does not have
timetables");
           console.log(result.timetables.length);
           return true;
       });
   }).timeout(10000);
   it("sets priorities and modules", () => {
       return wrapped(data, context).then(() =>
           docRef.get()
       ).then(doc =>
           assert.deepEqual(doc.data(), { ...userDoc, ...data })
       )
   });
});
```

Suggested enhancements

Interoperability with NUSMods

As a companion tool to be used in conjunction with NUSMods, the interoperability of the two web applications would be greatly enhanced if timetables generated on PlanNUS could be imported onto NUSMods. This would remove the need for users to recreate their PlanNUS timetables manually on NUSMods, allowing the two web applications to complement one another more effectively.

PlanNUS as a standalone timetable planner

In order to elevate PlanNUS from a companion tool to a fully-fledged academic timetable planner, future versions of PlanNUS should implement an edit feature for the generated timetables. An edit feature will benefit students who are unsuccessful in securing all the modules or class slots in their generated timetable and need to make changes to obtain their finalised timetable. Currently, PlanNUS will likely only see heavy user traffic at the beginning of each semester, when students want to create an initial template to base their finalised timetable on. However, with an edit feature, students will no longer need to recreate their generated timetables in NUSMods and can simply utilise PlanNUS on its own for their timetabling needs across the duration of the semester.

Cooperative timetable planning

The inclusion of an edit feature will also greatly enhance the current subscribe feature, further distinguishing it from simply saving a copy of a shared timetable. Once a timetable is shared and subscribed to, any changes made by the owner of the timetable will be automatically reflected for each of that timetable's subscribers.

The share feature itself can be enhanced with the implementation of a timetable comparison feature. Students will be able to make side-by-side comparisons of their own timetables with the ones that their friends have shared. This will be especially useful for friends taking the same module who wish to cooperatively plan their timetables with friends so that they are able to attend the same lecture/tutorial slots, thus enhancing the social aspect of PlanNUS.

Enhancing usability through informative features

While the user interface was kept deliberately simple and minimal, some users may actually find this more confusing compared to a user interface that features abundant and detailed labelling. Therefore, to reach the middle ground between aesthetic and informative, future versions of PlanNUS will prioritise information based features such as a help page and an interactive tutorial for first-time users. These will address the concern of guiding users whilst maintaining the minimalist aesthetic of the application, therefore enhancing usability.

Redux for faster loading and cross component states

In the project, cross component states were handled through the use of context and call from the database whenever data was required in order to ensure a single source of truth. However, calling from the database had an impact on the usability of the application as the loading time for each call was significant.

Redux, as a central store, would help in keeping local states for faster loading of components. For example, in order to generate timetables, user's priorities, modules of the selected academic year, and user's selected modules need to be loaded. As the amount of modules for each academic year is large, fetching the data takes a long time. This is particularly apparent if the user refreshes the page, and all current states and data are reset. It would be quicker if the user's current state and data were saved into the store, and that data were read from the store upon mounting of the component instead of being fetched again. Usability is enhanced as users do not need to wait for the data to be fetched and re-enter whatever they had previously input.

Redux can also update relevant components if a particular state has changed. For instance, the components SavedTimetables and SelectTimetables require the user's default timetable ID. However they are not related to one another. Hence, it was hard to figure out where to pass the data to as props. With Redux, it can be ensured that the components get the updated timetable ID whenever it changes.

In this sense, Redux will be able to provide a more streamlined state management and improve maintainability and predictability of syncing actions with states in the application.

With Redux, state management is improved. As PlanNUS is relatively complex with many states, it may be better to use Redux which provides a strict structure for how the code and state should be managed, thus ensuring greater maintainability of code.