Contents

Contributions Table

Team member name	Detailed contributions per chapter	Percentage contributions overall	Signature
Michael Parker UP2263259	Chapter 1: Problem specification Chapter 2: Design Chapter 3: Implementation Chapter 4: Testing Chapter 5: Critical analysis	33.33%	
Lewis Ryan UP2277906	Chapter 1: Problem specification Chapter 2: Design Chapter 3: Implementation Chapter 4: Testing Chapter 5: Critical analysis	33.33%	
Nickyyl Sharma UP2269939	Chapter 1: Problem specification Chapter 2: Design Chapter 3: Implementation Chapter 4: Testing Chapter 5: Critical analysis	33.33%	

Requirements Gathering Methods

To develop requirements, we utilised a semi-structured interview, with an additional set of questions designed for society committee members – collecting qualitative primary data. This allowed us to gain a comprehensive understanding of our participants' experience as both members of a society, and members of their committees.

We made use of an opportunity sampling method – and for the majority of our interviews we attempted to keep our sample representative by limiting our sample frame to people sitting down in the following buildings: Eldon, Park, University Library

We interviewed 9 members who were members of a wide range of societies, on average for 10 minutes each, recording each one as it happened. The interviews were then transcribed for us to analyse, allowing us to see patterns to inform our requirements.

Question Sets

1. How many societies are you a part of?
2. How many different apps do you have for society news?
- Would you be interested in something to put all of these in one place?
3. When you are missing information, how do you get in touch with your societies?
- How long does it take for them to get back to you?
4. Do you find you miss events due to messages getting lost?
5. What type of posts do the societies make on social media?
6. What is your favourite social media app that you use for societies and why?
7. How many social media apps do you use, and of these which do you use the most and why?
8. How many societies where you interested in, and how many have you actually gone to?
- Why did you miss their events?
9. What's your best way to interact with a society's event?
10. Do you prefer seeing the details first, chatting to someone before deciding or being able to reserve a spot through an app?
11. If you were to share an event with friends, what would be the easiest way for you to do that?
12. If you're deciding whether to attend an event, does viewing a guest list help?

1. How do distribute news to your members?
- Why do you use this method
2. How would you say members engage with your announcements?
3. What is the biggest hurdle to engage with your society?
4. What forms of media (E.g. text, images, video, polls) do you distribute and why?
5. Have you set up a shared calendar? If not, do you believe this could be useful?
6. Do you send out communication over more than one channel already?
7. Do you manage your society on multiple social media accounts, if so, how difficult is this
8. When managing queries related to the society, how long do you typically take to reply
9. When posting about events, what information do you typically include?
10. What is your typical event turnout, and how do you think it might be improved?

Reflections on Methodology

Largely we believe our methodology to have been successful, through the nature of our semi structured interview format, we avoided many of the pitfalls of structured and unstructured interviews. The use of an opportunity sample may have been problematic, as it would have given rise to investigator effects and so our data may not be as representative as we hope, potentially lacking external validity.

On the other hand, our research may also lack internal validity, with our 3rd question perhaps being too leading – giving way to demand characteristics; a majority of our participants replied positively.

Whilst most of the individual questions were suitable for prompting participants, question 11 proved to be less useful, as finding out the details was a must for all participants – and so we would change this if we did this again.

Our 9th question was also too vague, requiring us to add clarification which should have been there to begin with.

Reflections on Results

As mentioned above, most participants were positive about the potential application.

The insights regarding the numbers of societies people were a part of (average x), and how many they intended to join from the start of the year (average y) indicated the troubles societies have with member retention – with this being partially demonstrated by interviews with committee members.

There were however some concerns about notifications being overwhelming. This is consistent with the notification heavy nature of WhatsApp and Instagram – which were noted as being the apps people use most for societies.

The information we received about people's favourite apps (TikTok and Instagram) ultimately reinforced our understanding of the prevalence of doomscrolling[insert reference], which we will be taking on board as we progress to the design of our user interface. Equally, the generally speedy response time for members contacting societies will need to be something we enable with our UI.

The most common way people interact with posts is likes (or affirming reactions), across Instagram Posts, Stories and WhatsApp messages – with the posts mainly being text/image based.

Calendar posters also seem to be a common way societies outreach to members – supporting our core calendar feature. These also require images to be posted – which alongside text were the most common post type. Polls and videos were also common, and so a seamless integration for these will be important.

Few expressed support for a guest list, and so we'll ensure our UI does not prioritise this feature.

User Requirements

Society Member requirements

- users should be able to log in / register
- users should be able to join multiple societies
- users should be able to contact a societies' committee
- users should be able toggle on/off notifications for a societies' feed
- users should be able to interact with posts
  - vote in polls
  - like
  - tag (invite) friends to an event
- users should be able to add a single societies' event to their device calendar
- users should be able to add all a societies' events to their device calendar
- users should be able to view a guest list
- users should be able to RSVP to an event
- users should be able to add society members as their friends
    - users should be able to see a list of society members

Committee Member requirements

- users should be able to upload a post with a combination of text / images
  - users should be able to 'link' their post to an event
- users should be able to manage events
  - create an event
  - modify an event's details
  - switch an event's status (to public / private / scheduled)
  - link a post
- users should be able to respond to a society member

System Requirements

Concepts

SM – Society Members
CM – Committee Members
MIDS – Multi-item Data Structure
ID – A String
File Path – A String

SM (Society Members) Users have:

An Email [must be a valid port.ac.uk email address | not-null, unique] (String)
A password [minimum 8 characters | not-null] (String)
List of societies [nullable] (MIDS, composed of IDs, with settings)
List of friends [nullable] (MIDS, composed of IDs)

CM (Committee Members) Users have:

An Email [must be a valid upsu.net email address | not-null, unique] (String)
A password [min 8 chars | not-null] (String)
Society ID [not-null] (string)

Societies:

List of posts [nullable] (MIDS, composed of IDs)
List of events [nullable] (MIDS, composed of IDs)
List of members [nullable] (MIDS, composed of IDs)
List of settings [not-null] (MIDS)

Posts:

List of Likes [default 0 | not-null] (int)
Post Data [not-null] (MIDS, see below)
Linked Event [nullable] (String)

Events:

List of Likes [default 0 | not-null] (int)
Post Data [not-null] (MIDS, see below)
Linked Event [nullable] (String)

Post Data:

Description / Text [max 2,200 chars | not-null] (String)
List of Images [nullable] (MIDS, composed of file paths)
Poll [nullable] (MIDS, see below)

Poll:

List of Options [min 2 items | not-null] (MIDS, composed of Strings, with votes (int))
List of Voters [nullable] (MIDS, composed of IDs, with index of voted option (int))

Functional requirements

users can register for an account:

needs a unique Portsmouth uni email address

user enters a password for the account

user is also logged in after account creation

user can log in:

user provides an email address

user provides a password

user only gets logged in if both password and email address match an account's email and password

user adds a society to their list of societies:

user searches for a society

user then adds the society to their list of societies

only societies that are not already in the list can be added

if the society is already in the list the system message the user of this fact and not add it to the list

this message should be acknowledged by the user

users can contact committee members of a society:

user is shown the email address of the committee members of the society

society must be in the users list of societies

users can toggle society notifications:

user can toggle the notifications attribute of the societies in their list of societies

users can interact with posts:

vote in polls

like posts

vote in polls:

users can vote in polls in a society's post

user selects the option they want to vote for

user can only vote once

society that made the poll must be in the users list of societies

like posts:

user can increment or decrement a post's likes

user can change wither they have incremented, decremented or done nether

user can only like posts whose society is in the users list of societies

The system should generate an ICS file for either a single event or a society's upcoming events, through data fetched from the database

The system should direct the user to their calendar app of choice to save the event, or save the ICS file should no appropriate app be installed

The system should update to demonstrate to the user that the event was added

The system should query the database to return a list of attendees for an event

The system should update the database with a user's RSVP (Boolean, attending / not)

The system should update to demonstrate to the user their RSVP status

The system should query the database to return a list of members within a society for users to view

The system should verify the target user is accepting friend requests

The system should update both users' outgoing/incoming requests, notifying target users of new requests

The system should notify a target user that they have been invited to an event

The system should verify that all post details are filled with valid data.

The system should send an API request to the backend with the verified post data.

The system should then add this post to the database via the database connector.

The system should fetch an event ID from the database using the restAPI.

The system should fetch a Post object from the database and then send a new request to the database manager inserting the eventURL to the posts row in the database.

The system should verify that all event details are filled with valid data.

The system should send an API request to the backend with the verified event data.

The system should then add the new event record to the database via the database manager.

The system should fetch the existing event object from the database using the restAPI to populate the fields.

The system should verify that the modified text boxes contain valid data.

The system should send an API request to the backend with the updated data and the event ID.

The system should update the specific event row in the database via the database manager.

The system should send an API request to the backend containing the event ID and the selected status tag.

The system should query the database manager to locate the specific event.

The system should update the status column for that event in the database to reflect the user's selection.

The system should fetch the ID of the raised concern from the database using the restAPI.

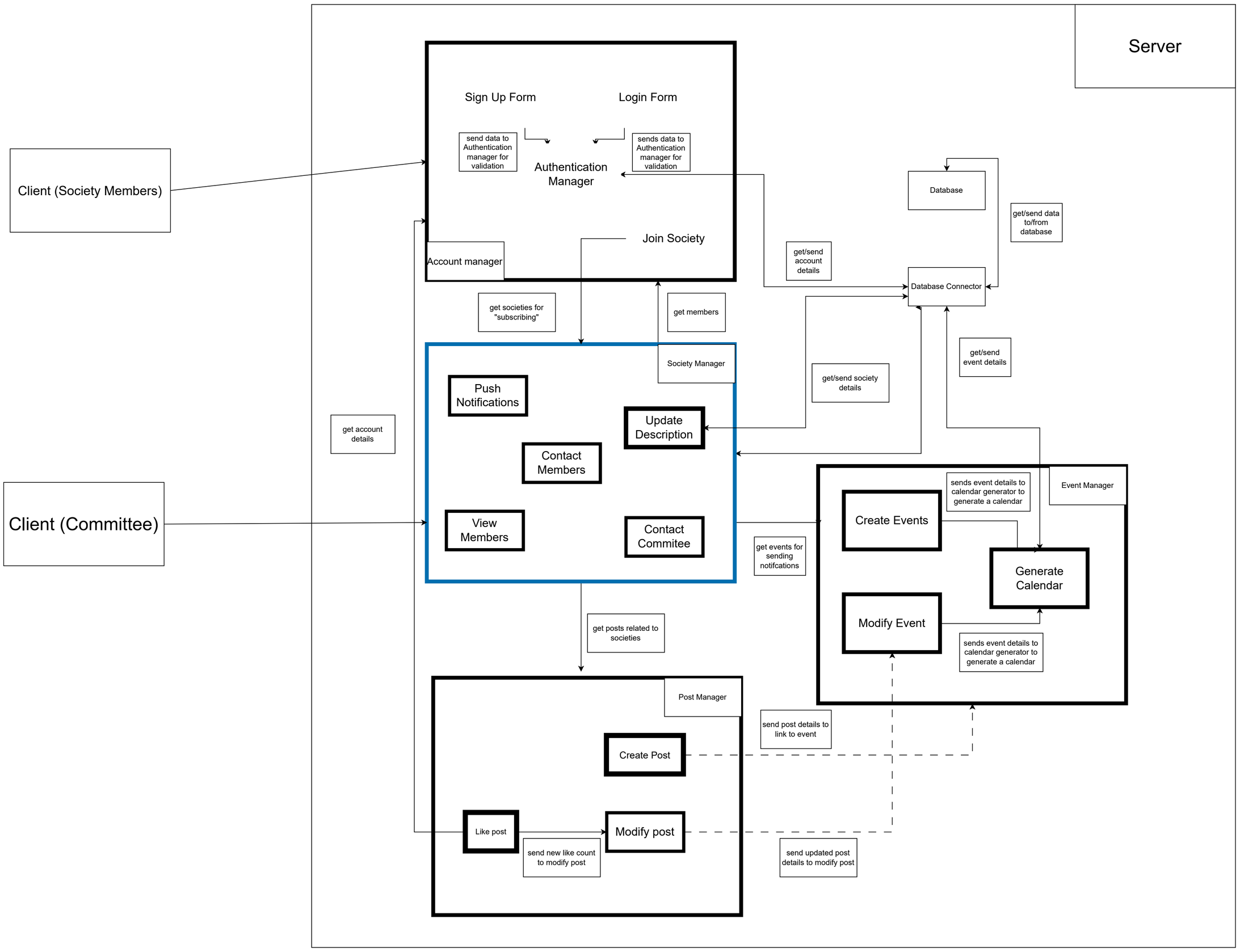
The system should verify that the response text box is filled with valid data.

The system should send an API request to the backend linking the new response to the concern's ID.

The system should insert the response text into the database via the database manager.

Non-Functional requirements

1. The system should not take more than 3 seconds to load the event dashboard.
2. The system should not take more than 3 seconds to complete the upload of a standard text post.
3. The system should be able to fetch and display a specific event's details within 3 seconds of the user's request.
4. The system will have login information saved, allowing logins to be reused (session persistence).
5. API requests should complete within 2-3 seconds under normal load
6. Maximum file size of 10MB per image
7. Users must be authenticated before creating, modifying, or linking posts/events
8. All input data must be sanitized to prevent SQL injection and XSS attacks



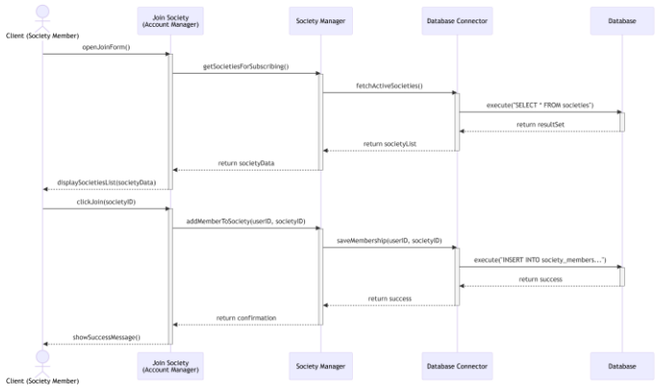
The system diagram outlines a client-server architecture designed to manage a university society application. The structure is divided into two main areas: the client side, which includes the user interfaces for "Committee" members and "Society Members," and the server side, which handles all the processing, logic, and data storage. The server is organized into four main sections that handle specific tasks: authentication, society management, event management, and post management.

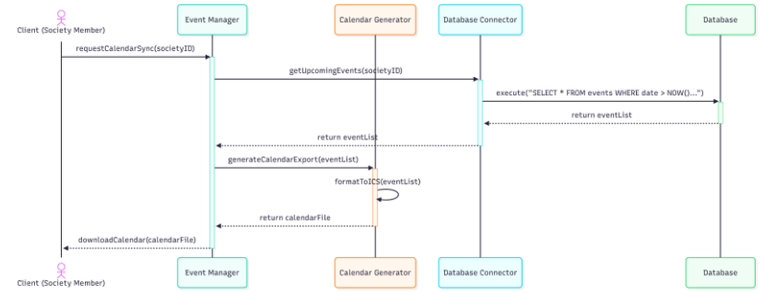
The Account Manager, which contains the "Sign Up" and "Login" forms. When a user enters their details, the data is sent to an "Authentication Manager" to check if the information is correct. There is also a "Join Society" feature here, allowing users to become part of a specific group once they are logged in.

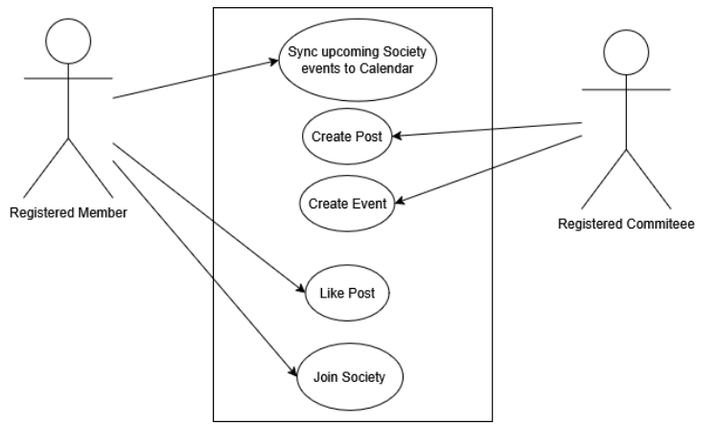
The society management. This part of the system handles the information for the society. It includes features for viewing the member list, contacting other members, and contacting the committee. It also allows administrative users to update the society's description and send push notifications to users. A "Society Manager" controls this section, handling requests such as getting the list of members or finding societies for a user to subscribe to.

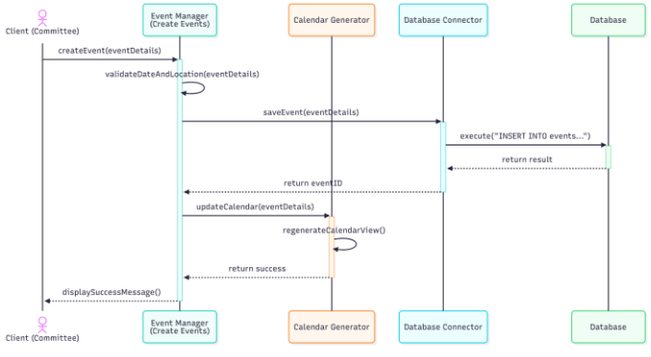
The event management section. This module is responsible for scheduling. Users can "Create Events" or "Modify Events." Once an event is created or changed, the details are sent to a "Generate Calendar" component, which generates an ICS file. This entire process is overseen by the "Event Manager." Connected to this is the post management section at the bottom in red. This allows users to create and modify posts on a feed. It also includes a "Like post" feature, which sends updated like counts back to the modify component. Arrows connecting the event and post sections suggest that posts can be linked directly to specific events.

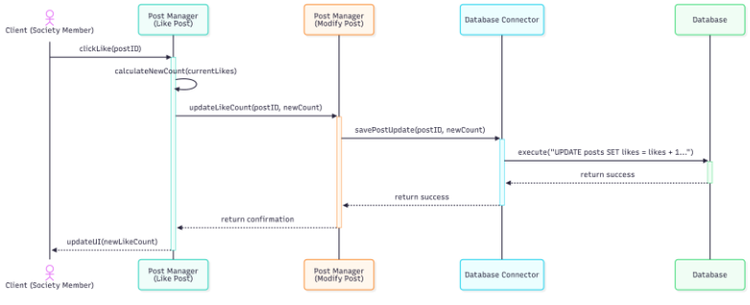
Finally, all the data from these different sections is stored in a central database on the far right. The system does not connect every single component directly to the database. Instead, it uses a "Database Connector." This acts as a central hub. The Account, Society, Event, and Post Managers all send their data to this connector, which then handles the actual reading and writing to the "Database."

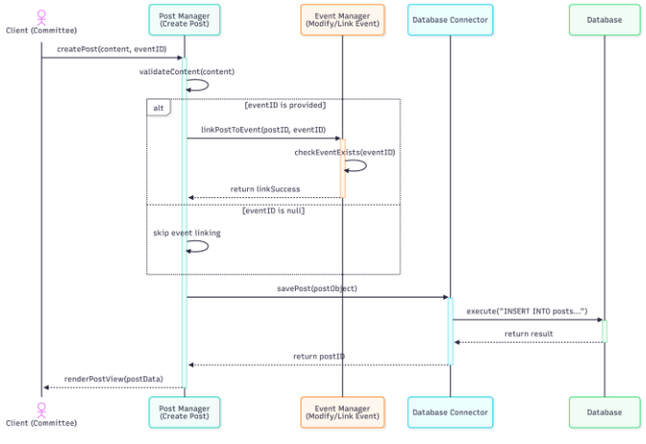

Client (Society Member)
Join Society (Account Manager)
Society Manager
Database Connector
Database
openJoinForm()
getSocietiesForSubscribing()
fetchActiveSocieties()
execute("SELECT * FROM societies")
return resultSet
return societyList
return societyData
displaySocietiesList(societyData)
clickJoin(societyID)
addMemberToSociety(userID, societyID)
saveMembership(userID, societyID)
execute("INSERT INTO society_members...")
return success
return success
return confirmation
showSuccessMessage()


Client (Society Member)
Event Manager
Calendar Generator
Database Connector
Database
requestCalendarSync(societyID)
getUpcomingEvents(societyID)
execute("SELECT * FROM events WHERE date > NOW()...")
return eventList
return eventList
generateCalendarExport(eventList)
formatToICS(eventList)
return calendarFile
downloadCalendar(calendarFile)


Registered Member
Registered Commiteee
Sync upcoming Society events to Calendar
Create Post
Create Event
Like Post
Join Society


Client (Committee)
Event Manager (Create Events)
Calendar Generator
Database Connector
Database
createEvent(eventDetails)
validateDateAndLocation(eventDetails)
saveEvent(eventDetails)
execute("INSERT INTO events...")
return result
return eventID
updateCalendar(eventDetails)
regenerateCalendarView()
return success
displaySuccessMessage()


Client (Society Member)
Post Manager (Like Post)
Post Manager (Modify Post)
Database Connector
Database
clickLike(postID)
calculateNewCount(currentLikes)
updateLikeCount(postID, newCount)
savePostUpdate(postID, newCount)
execute("UPDATE posts SET likes = likes + 1...")
return success
return success
return confirmation
updateUI(newLikeCount)


Client (Committee)
Post Manager (Create Post)
Event Manager (Modify/Link Event)
Database Connector
Database
createPost(content, eventID)
validateContent(content)
alt
[eventID is provided]
linkPostToEvent(postID, eventID)
checkEventExists(eventID)
return linkSuccess
[eventID is null]
skip event linking
savePost(postObject)
execute("INSERT INTO posts...")
return result
return postID
renderPostView(postData)

Implementation Discussion

We will be running the backend using Python with the Flask framework alongside JWT and Flask-SQLAlchemy. Flask allows us to quickly and easily set up multiple web endpoints. JWT will provide access control for the Flutter mobile app; specifically, the authorization tokens will be created on the Flutter app and sent in the URL request to Flask.

After the endpoint has verified that the JWT is valid, it can make a request to the SQL database using SQLAlchemy, extract the relevant data, and return it via a REST API. The entire server will be run within a Docker container so it can be easily deployed onto a VM or VPS. The frontend will be developed entirely in Flutter, utilizing persistent storage to keep the user logged in using the shared storage system. When the app opens, a JWT will be created for the session, allowing the app to download the list of users' societies and check for new notifications.

We also explored alternatives for this stack. We considered using Firebase, which would remove the need for a dedicated server and SQL database by utilizing Google's serverless backend and built-in authentication. We also looked at React Native as a frontend alternative due to its ease of use and extensive set of libraries to help optimize development; however, the major advantage of Flutter is that all our members already know Dart.

Testing Discussion

The none-functional requirements will be tests by several methods, including timing, and testing inputs.

Non-functional requirements 1, 2, 3 and 5 will be tested by timing the speed of the system, passing if they do not exceed the time limit

requirement 4 will be tested by simulating logging in and ending the session then begining another, if the user is still logged in the test passes

requirement 6 will be tested by trying to upload an image over the max file size, if it fails the test passes

requirement 7 will be tested by trying to mutate a post while unauthenticated, if the system does not allow the mutation the test passes

requirement 8 will be tested by trying to execute XSS/SQL injection attacks if the attacks fail the test passes

the tests will be automatically run via flutter unit tests; other testing methods would be harder to integrate into the system as we are already using flutter, and as it’s the front end of the system, the other components can be tested through it.

Critical Analysis

This semester we have identified a key gap in the university experience for students - one which we believe our application can fill.

Through defining our methodology for gathering requirements, developing user requirements from these and then translating these into system requirements we were able to acutely identify the problem we aimed to solve.

We then developed an architectural model, to support our eventual implementation.

We accompanied these with 5 key use cases.

To prepare for implementation we then investigated the tools we could make use of, and the ways we would test our design.

Completing this was not a straightforward task as we encountered complications which challenged our teamwork skills - offering us opportunity to learn and achieve.

We initially we began with 7 team members, and using a simple table in a word document to track attendance.

As the first few weeks went by as we brainstormed draft ideas for our app, and explored how we designed our questions, we realised there was an

At the start we began with 7 team members but found that 4 of them wouldn't show up to pre agreed coursework meeting due to them oversleeping alarms and found they missed labs regularly due to sport societies. To deal with this we started agreeing to tasks over messages that they could do to contribute without having to come to group meeting. We ended up deciding the best course of action was splitting into two group. This allows the larger group to work around their sports while out new smaller group can follow a regular set timetable.