

**NANYANG
TECHNOLOGICAL
UNIVERSITY**
SINGAPORE

SC2006 - Software Engineering

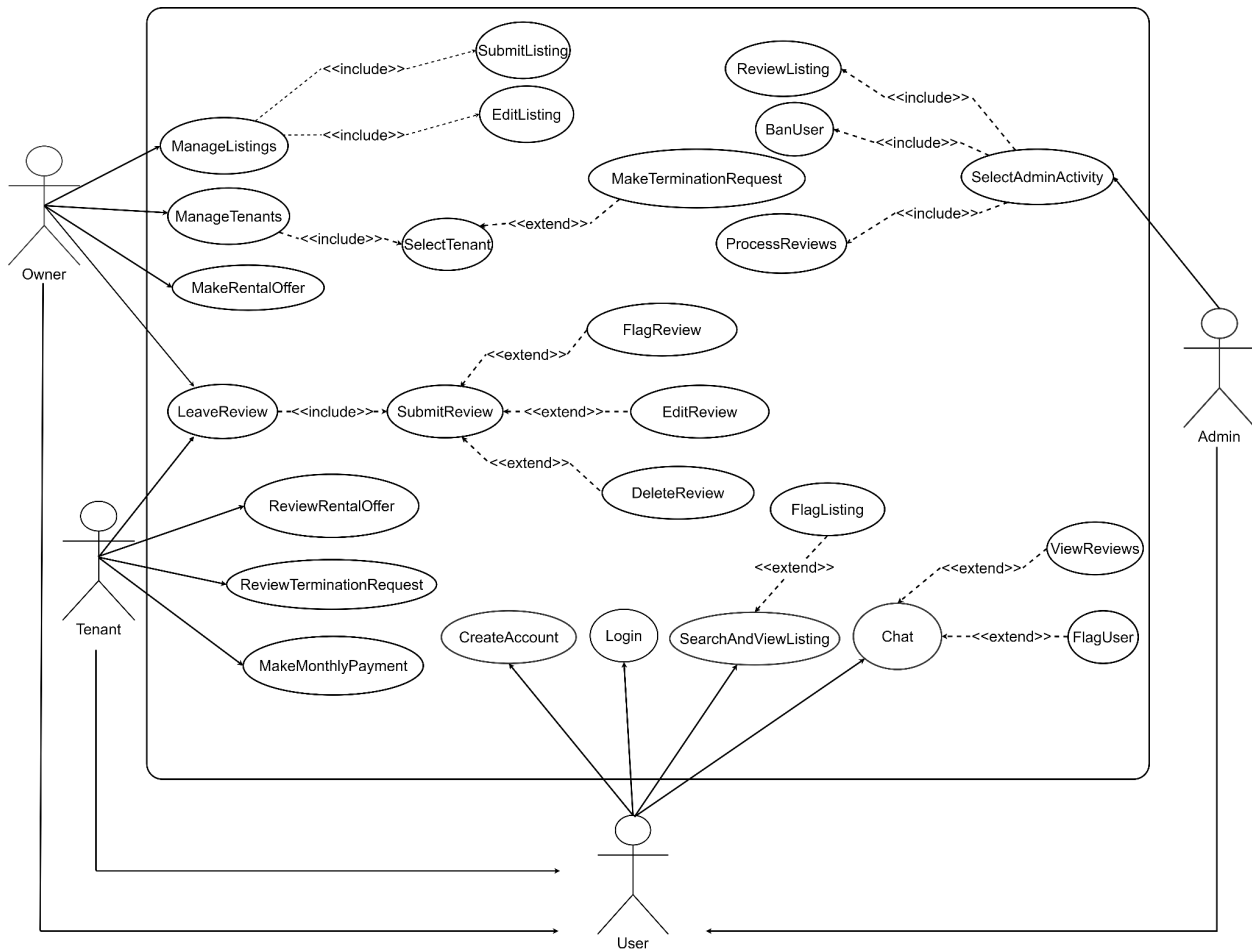
Lab 3 Deliverables

Lab Group	SDAD
Team	HomeGoWhere
Members	Lim Jing Rong (U2323260D)
	Loo Ping Wee (U2321814J)
	Goh Shuen Wei (U2321093D)
	Ng Jing En (U2321510K)
	Babu Sankar Nithin Sankar (U2323953B)
	Lau Zhan You (U2320608L)

1. Use Case Model.....	4
2. Design Model.....	5
A. Key Classes Diagram.....	5
B. Sequence Diagrams of Use Cases.....	7
I. For Use Cases under I.....	7
I.I SelectAdminActivity.....	7
I.II ReviewListing.....	8
I.III BanUser.....	9
I.IV ProcessReviews.....	10
II. For Use Cases under II.....	11
II.I ManageListings.....	11
II.II SubmitListing.....	12
II.III EditListing.....	13
II.IV ManageTenants.....	14
II.V SelectTenant.....	15
II.VI MakeTerminationRequest.....	16
II.VII MakeRentalOffer.....	17
III. For Use Cases under III.....	18
III.I ReviewRentalOffer.....	18
III.II MakeMonthlyPayment.....	19
III.III ReviewTerminationRequest.....	20
III.IV LeaveReview.....	21
III.V SubmitReview.....	22
III.VI FlagReview.....	23
III.VII EditReview.....	24
III.VIII DeleteReview.....	25
IV. For Use Cases under IV.....	26
IV.I Create Account.....	26
IV.II Login.....	27
IV.III SearchAndViewListing.....	28
IV.IV FlagListing.....	29
IV.V GetChat.....	30
IV.VI FlagUser.....	31
IV.VII ViewReviews.....	32
C. Initial Dialog map.....	33
3. System Architecture.....	34
4. Application Skeleton.....	37
A. Frontend.....	37
B. Backend.....	38

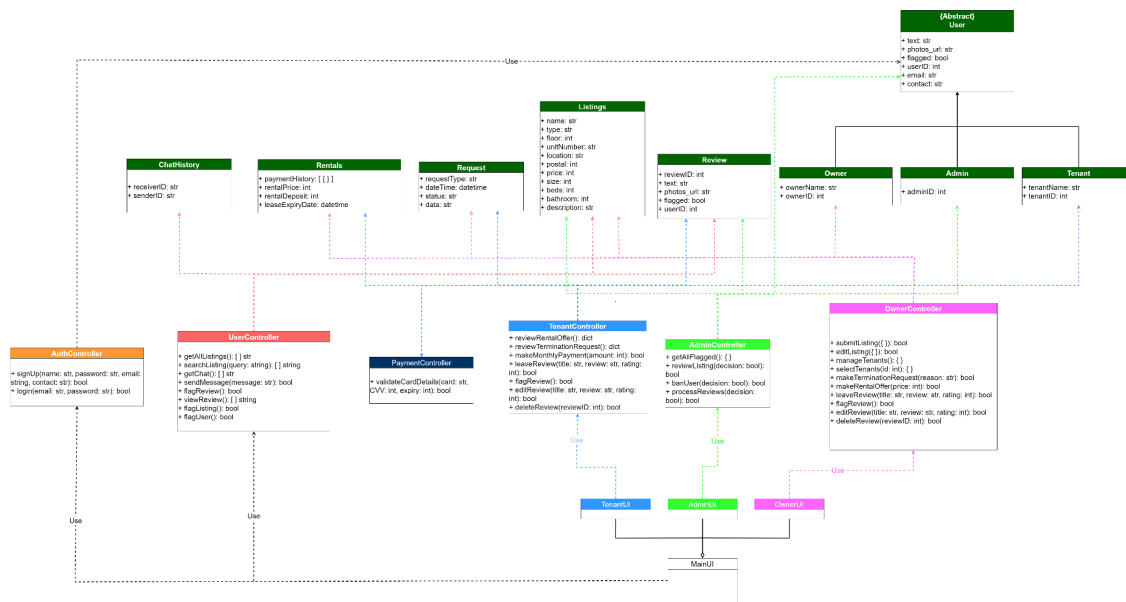
5. Appendix.....	39
Design Patterns Used.....	41
Tech Stack.....	41

1. Use Case Model



2. Design Model

A. Key Classes Diagram



App

Start-up class for initialisation of the frontend and backend.

Operations:

- `()`: renders the frontend User Interfaces (UI)
- `()`: starts the backend application and database for data access and modification.

User Interfaces (UI)

User Interfaces are what the users see and interact with on the application.

- **MainUI**: This is the main screen for Users. Users can perform tasks which can be classified under 2 broad categories: Authentication and HomeGoWhere's common application features that are accessible to Users of different roles.
- **AdminUI**: Screens specific only to Admins. EG: Admin Management Screen with the features to process flagged users, ban user and process reviews.
- **OwnerUI**: Screens specific only to Owners. EG: View listings posted and tenants under them.
- **TenantUI**: Screens specific only to Tenants. EG: Review rental offers and make monthly payments.

Controllers (Facade Pattern)

Application of the Facade Pattern is when the controllers serve as "interfaces" for interacting with the application's business logic, acting as entry points to the application. There are multiple controllers, depending on the specific use case and/or role of the User.

- **AuthController**: This is an Authentication Controller. It lets Users login and sign up. It has access to Users.

- **UserController:** This is the controller for HomeGoWhere's main usage. It lets Users get information on all the listings, search for a particular listing and query listings based on certain properties. It has access to the following entities: ChatHistory, Listings, Review.
- **AdminController:** This is the controller for Admins. It lets Admins review flagged Users, process reviews and ban users. It has access to the following entities: Listings, Review.
- **OwnerController:** This is the controller for Owners. It lets Owners submit listings. It has access to the following entities: Rentals, Request, Listing.
- **TenantController:** This is the controller for Tenants. It lets Tenants review rental offers and make monthly payments, etc. It has access to the following entities: Rentals, Request, Review.
- **PaymentController:** This is the controller for Payments. It allows Tenants to make payments.

Data Access Layer - (Factory Pattern + Strategy Pattern)

Possible improvements to our program can be implementing design patterns to ensure that data access, modifications and storage or your CRUD (Create, Read, Update, Delete) operations are always consistent every time. It allows the application to process huge amounts of data operations and users at scale.

The data access classes and methods would implement two design patterns - **Factory Pattern** and **Strategy Pattern**.

Factory Pattern

- Factory Pattern allows to create object objects where the exact class of the object created is only known during runtime. It encapsulates the instantiation logic and allows for more flexibility and decoupling of code.
- Implementation of this for our data access could be a factory class responsible for creating instances of data we intend to read or store.

Strategy Pattern

- Strategy Pattern allows a program to select its behaviour at runtime. This abstracts away a family of methods and encapsulates them into one single interface that is flexible and extensible, and works after without changing the core code.
- This can be done by creating an interface which will act as the abstraction of a common set of data access methods (CRUD).
- When used in conjunction with the abovementioned factory class, the program will be easily extensible as adding new data types to the storage does not require changes to the core code, and can be easily implemented in the factory and interface classes.

Other Design Patterns

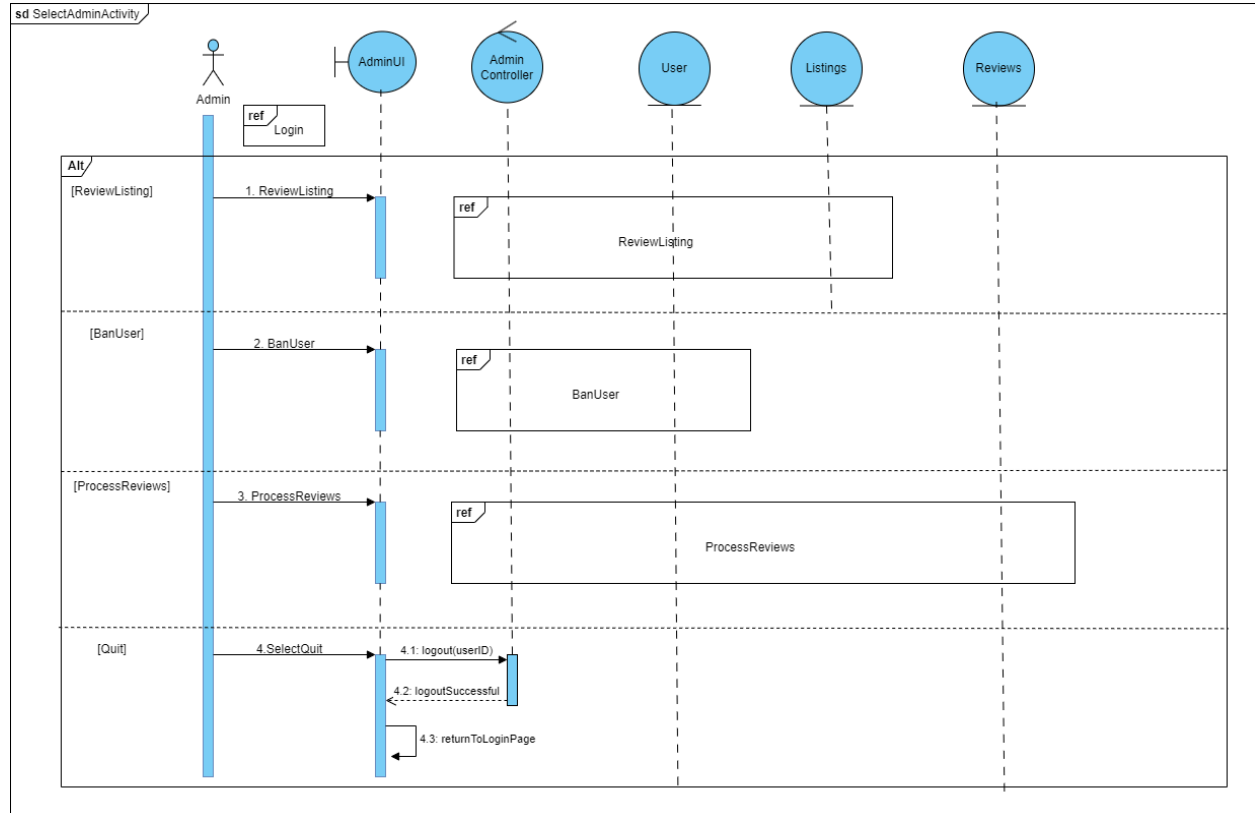
1. Publisher-Subscriber Pattern

Websocket connection between two different Users can be implemented to facilitate real time chat, and will not require a refresh to display every new message.

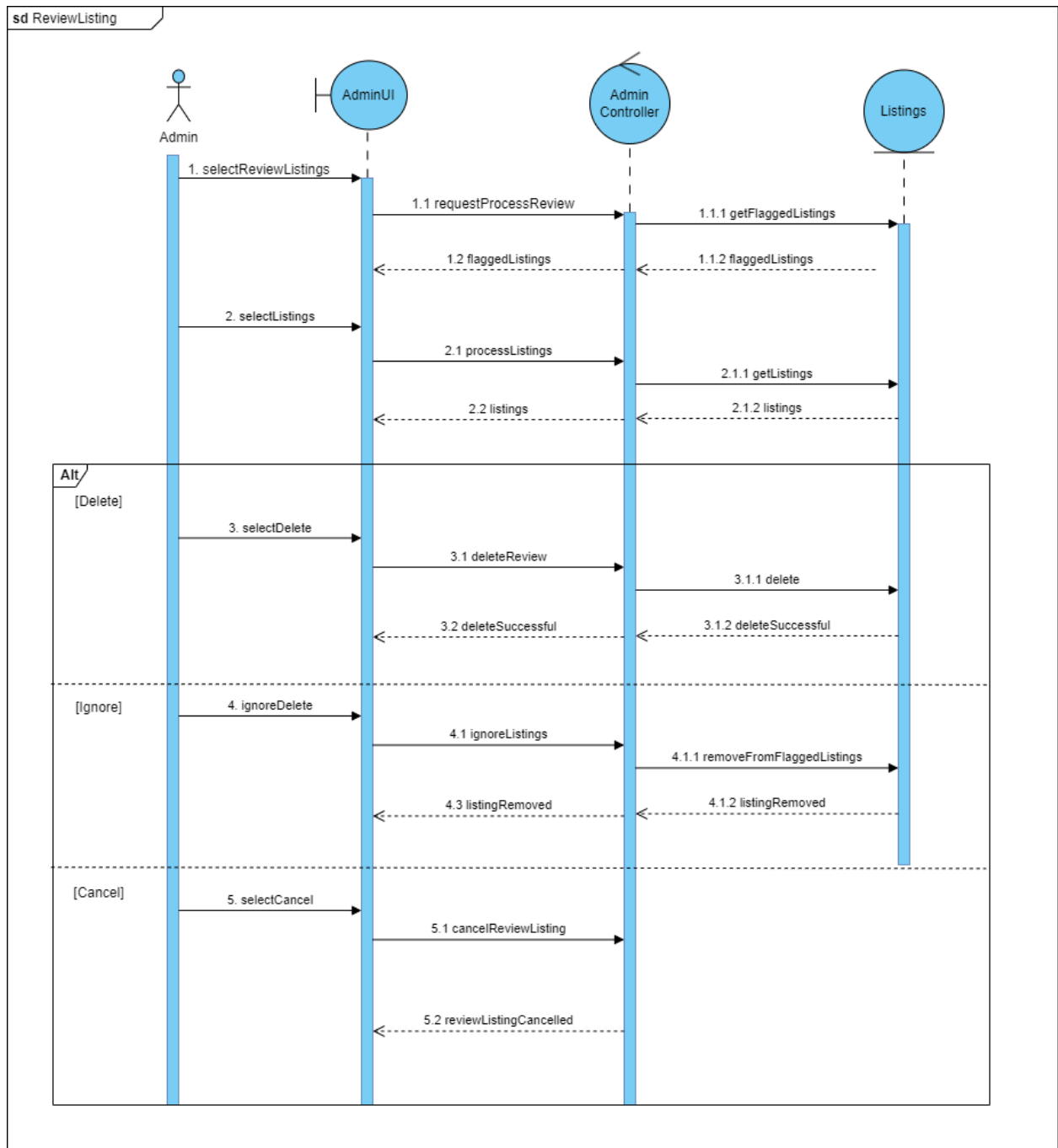
B. Sequence Diagrams of Use Cases

I. For Use Cases under I

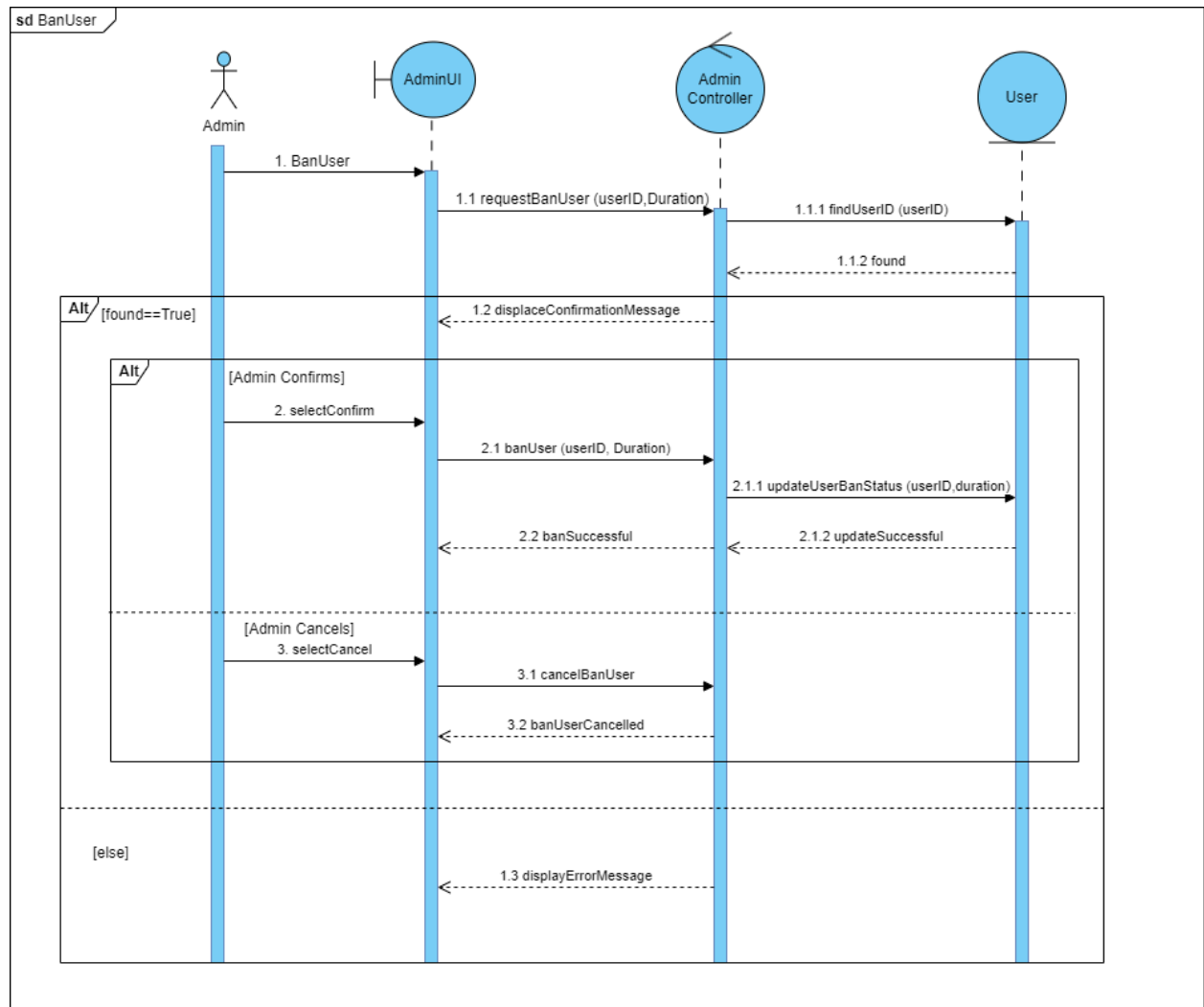
I.I SelectAdminActivity



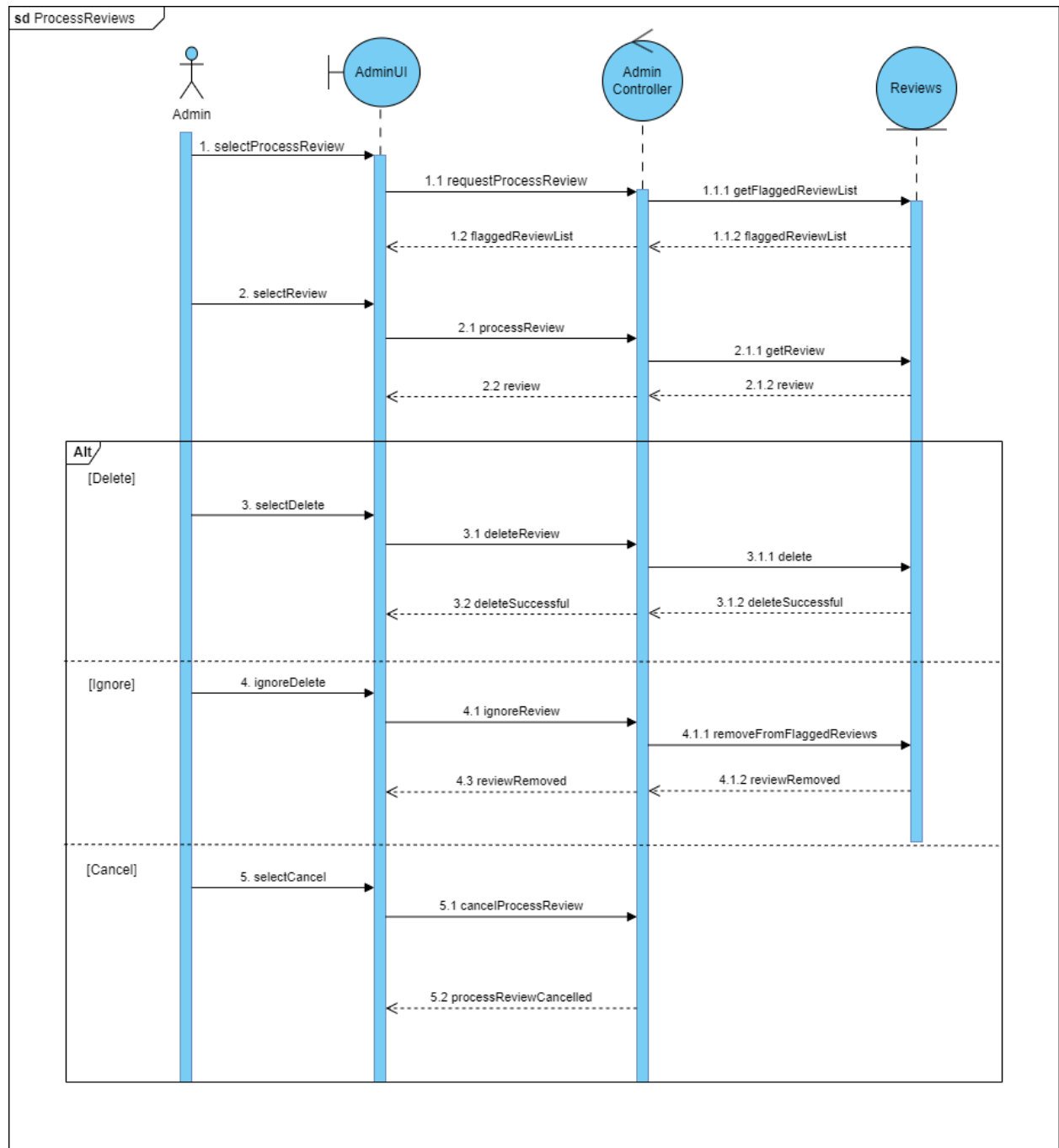
I.II ReviewListing



I.III BanUser

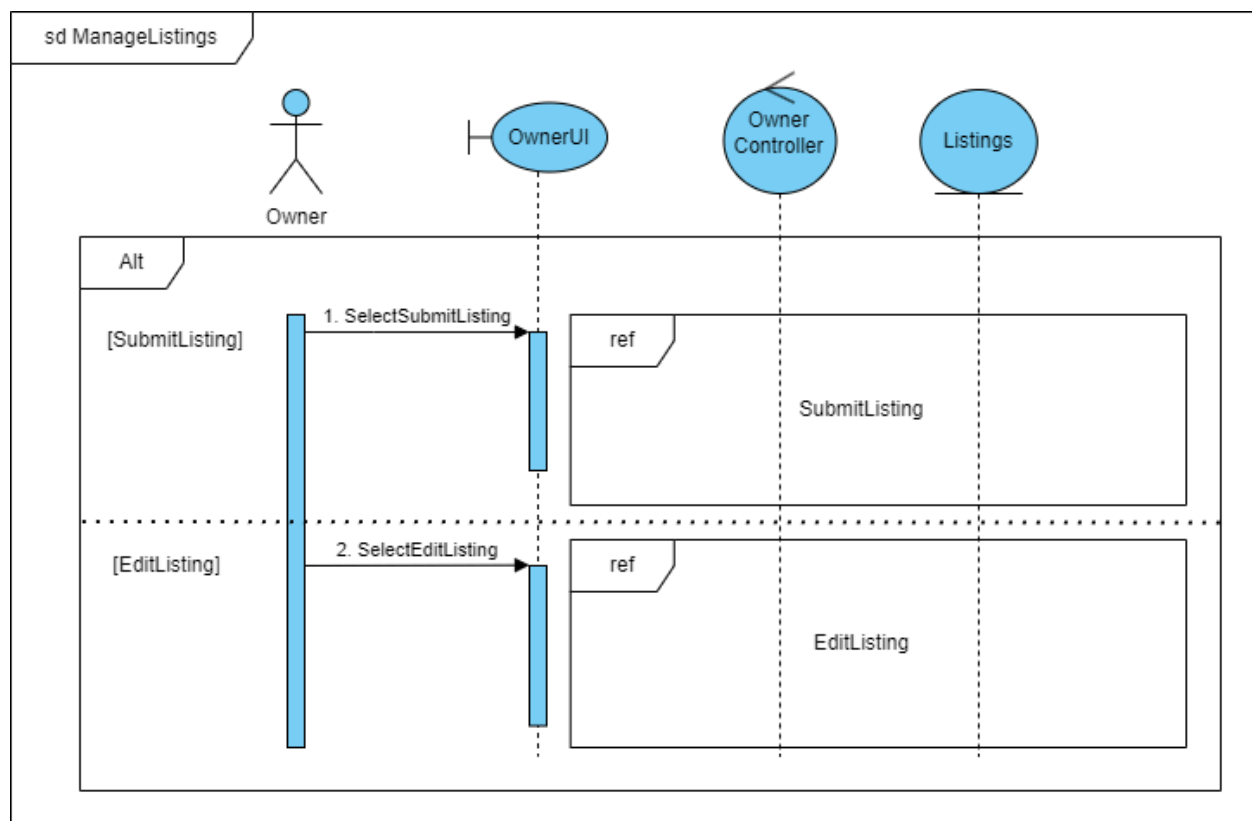


I.IV ProcessReviews

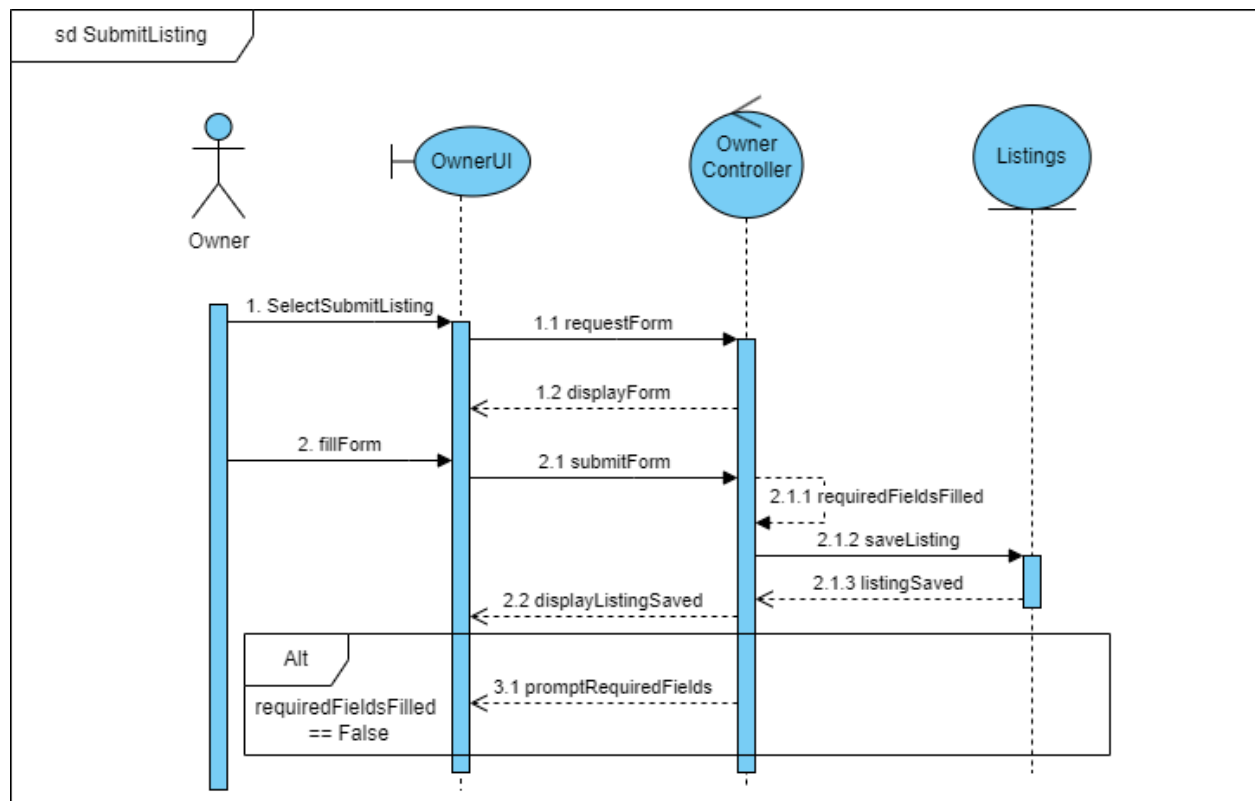


II. For Use Cases under II

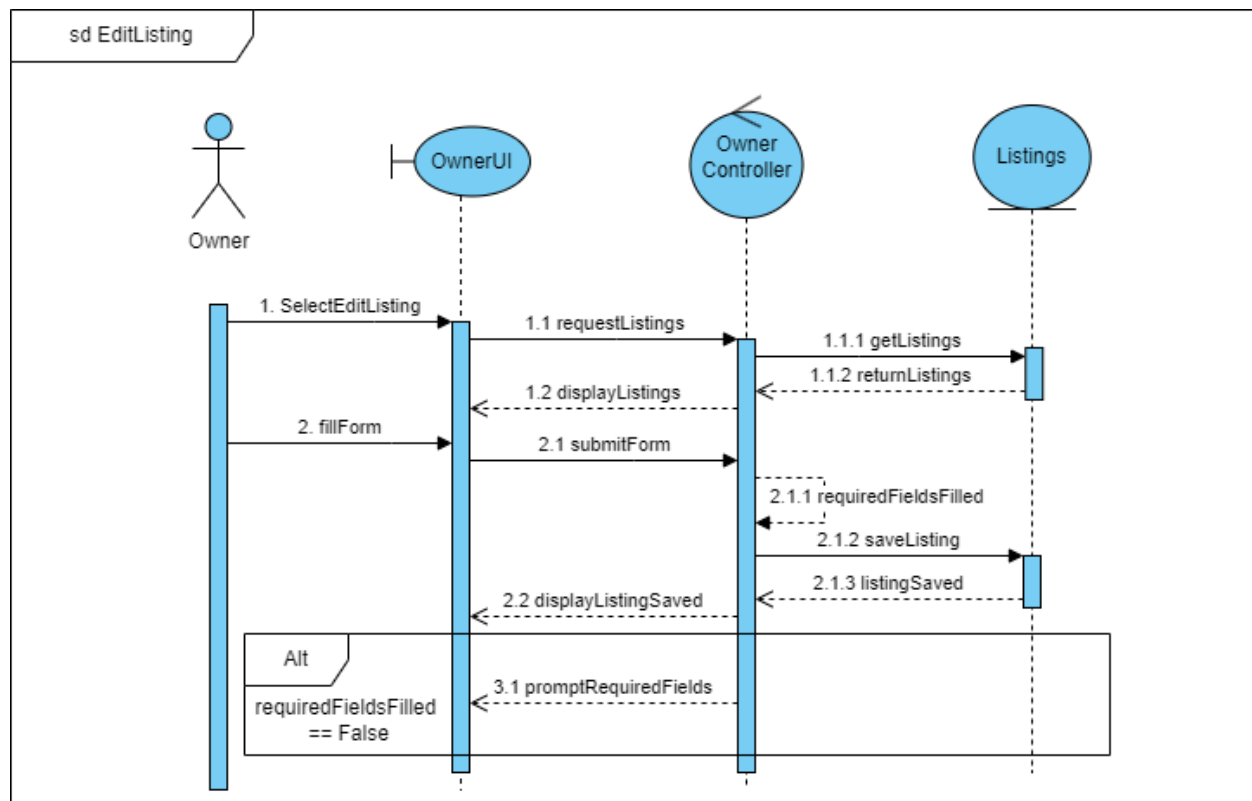
II.I ManageListings



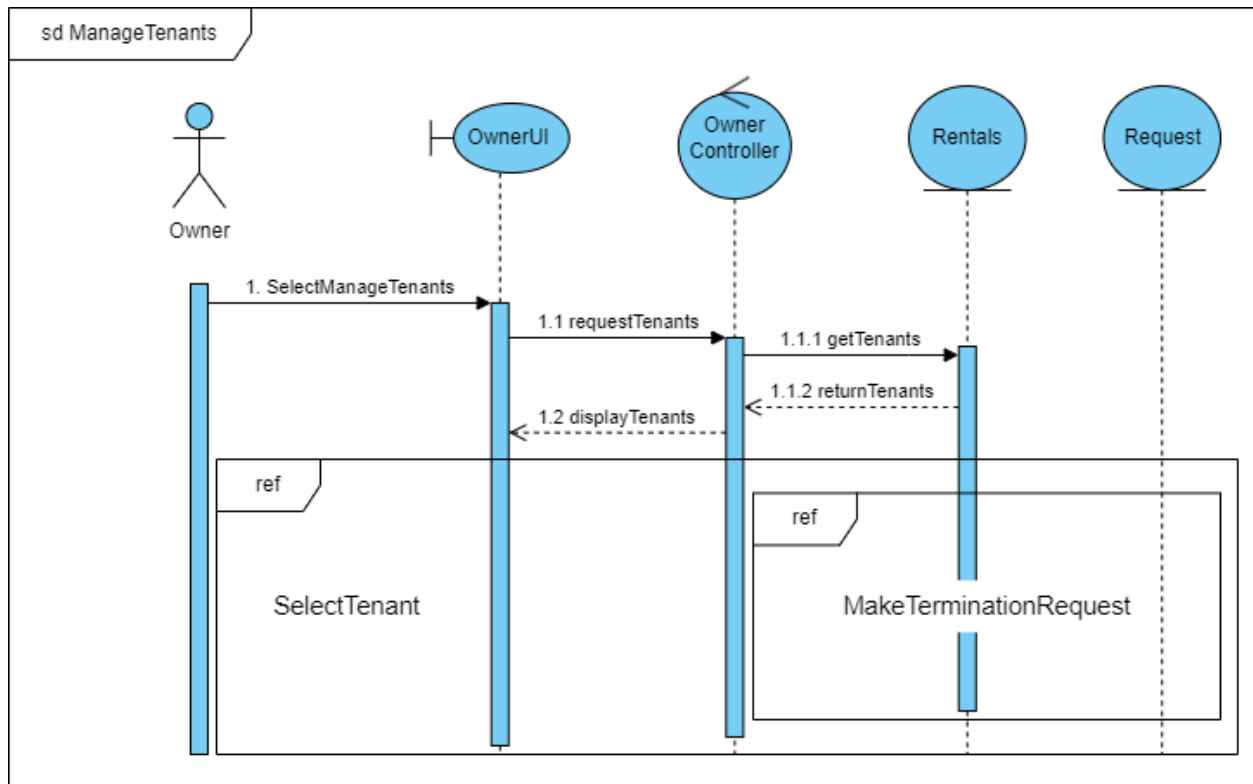
II.II SubmitListing



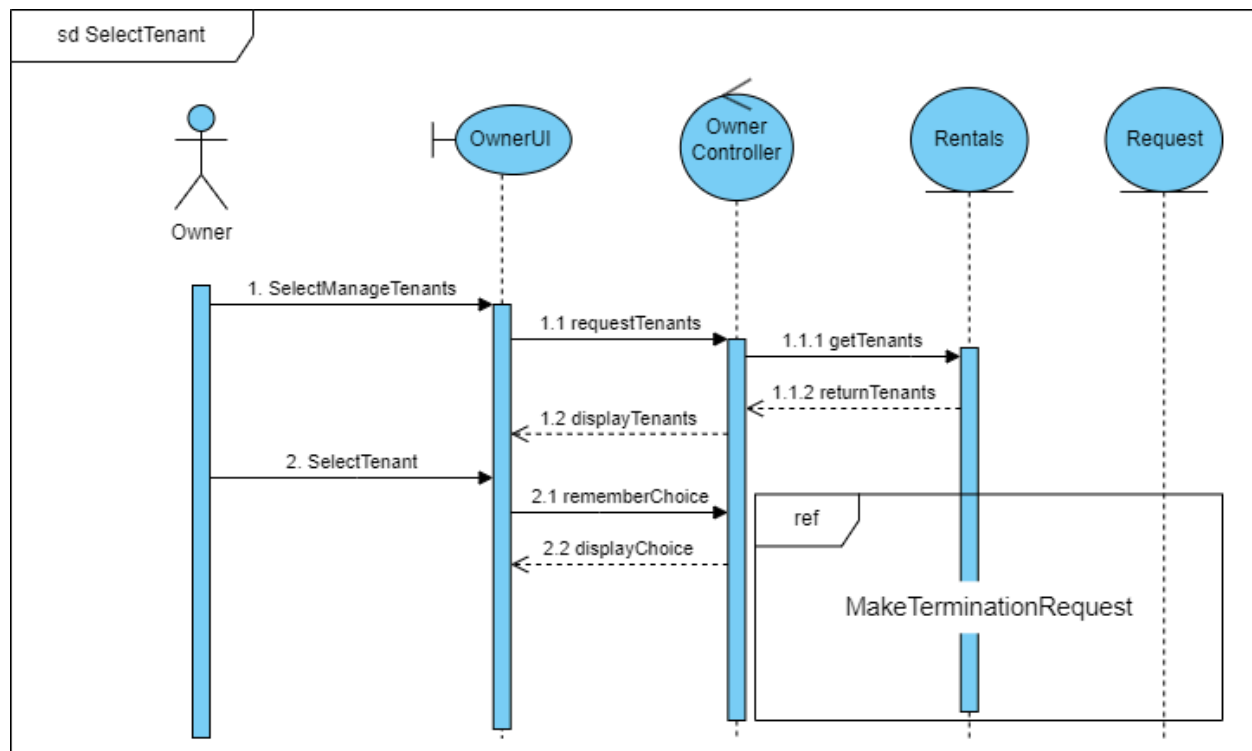
II.III EditListing



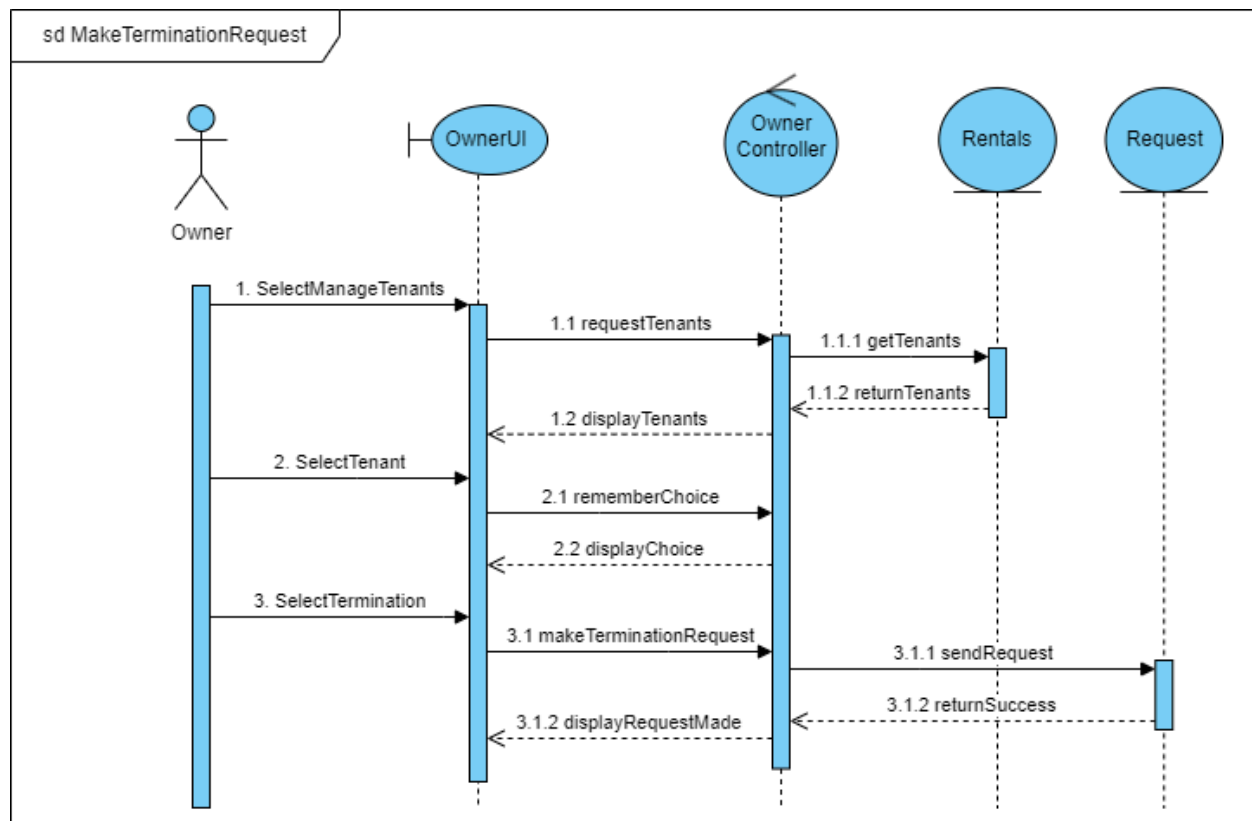
II.IV ManageTenants



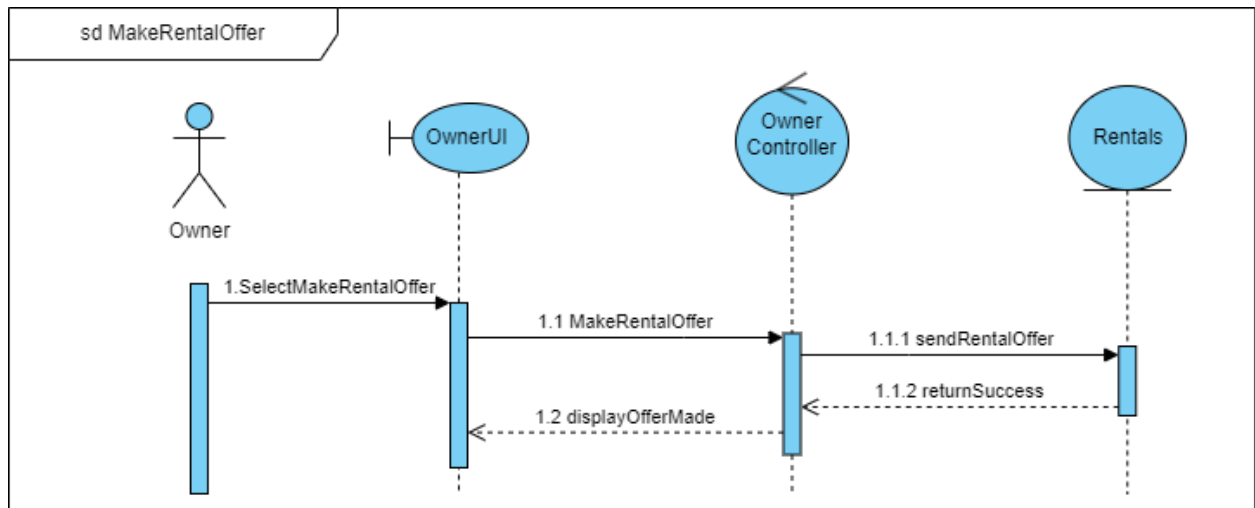
II.V SelectTenant



II.VI MakeTerminationRequest

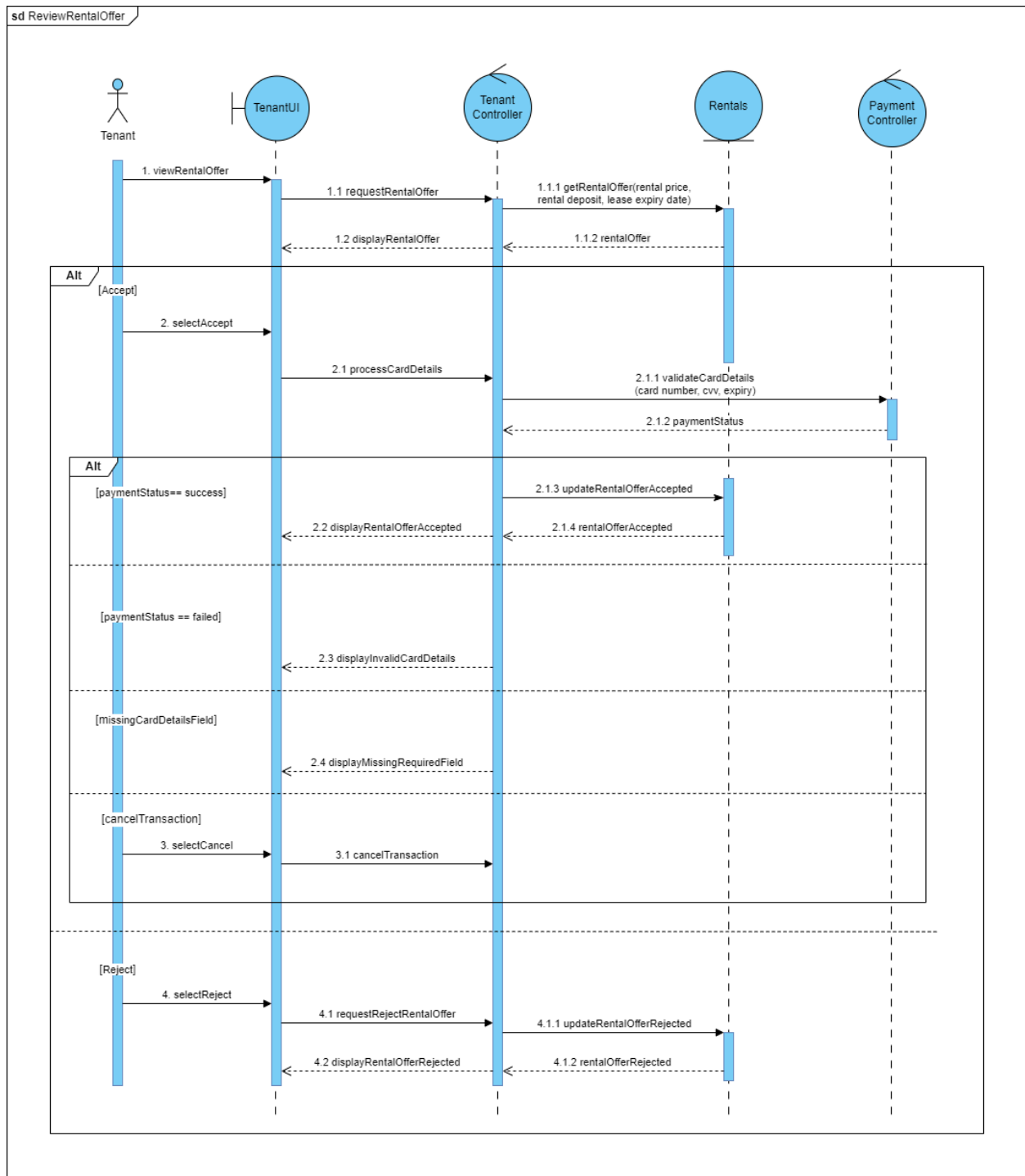


II.VII MakeRentalOffer

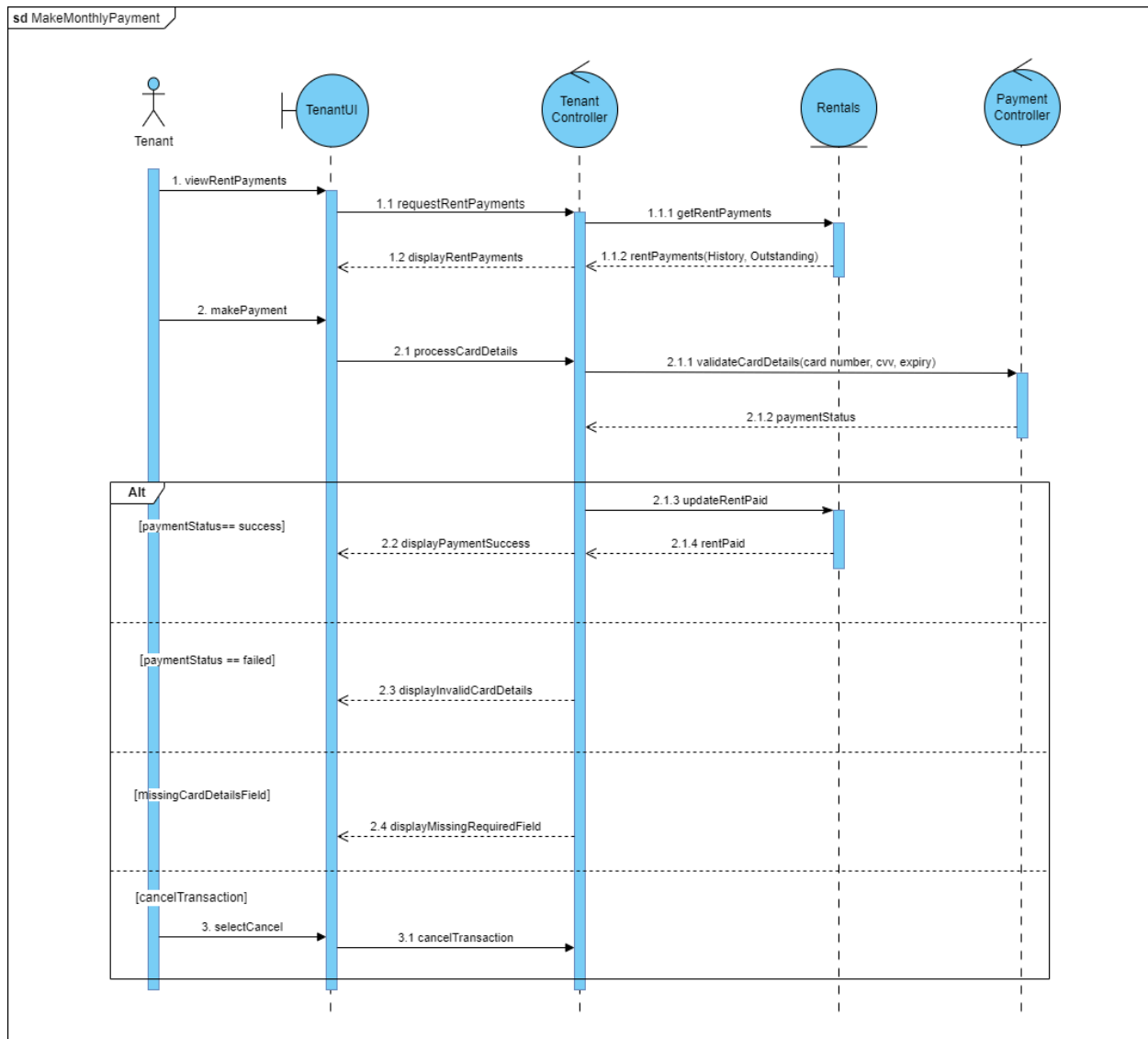


III. For Use Cases under III

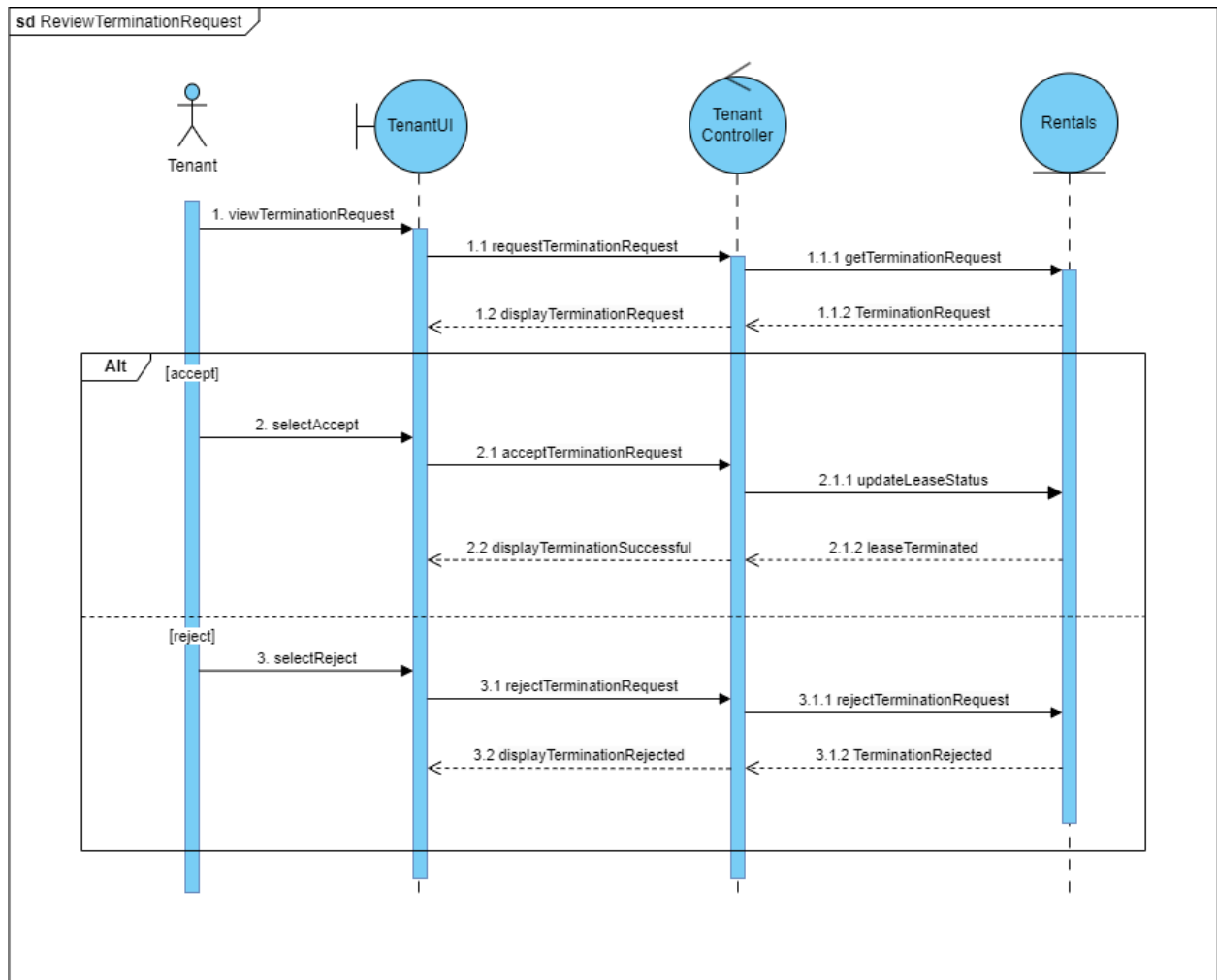
III.I ReviewRentalOffer



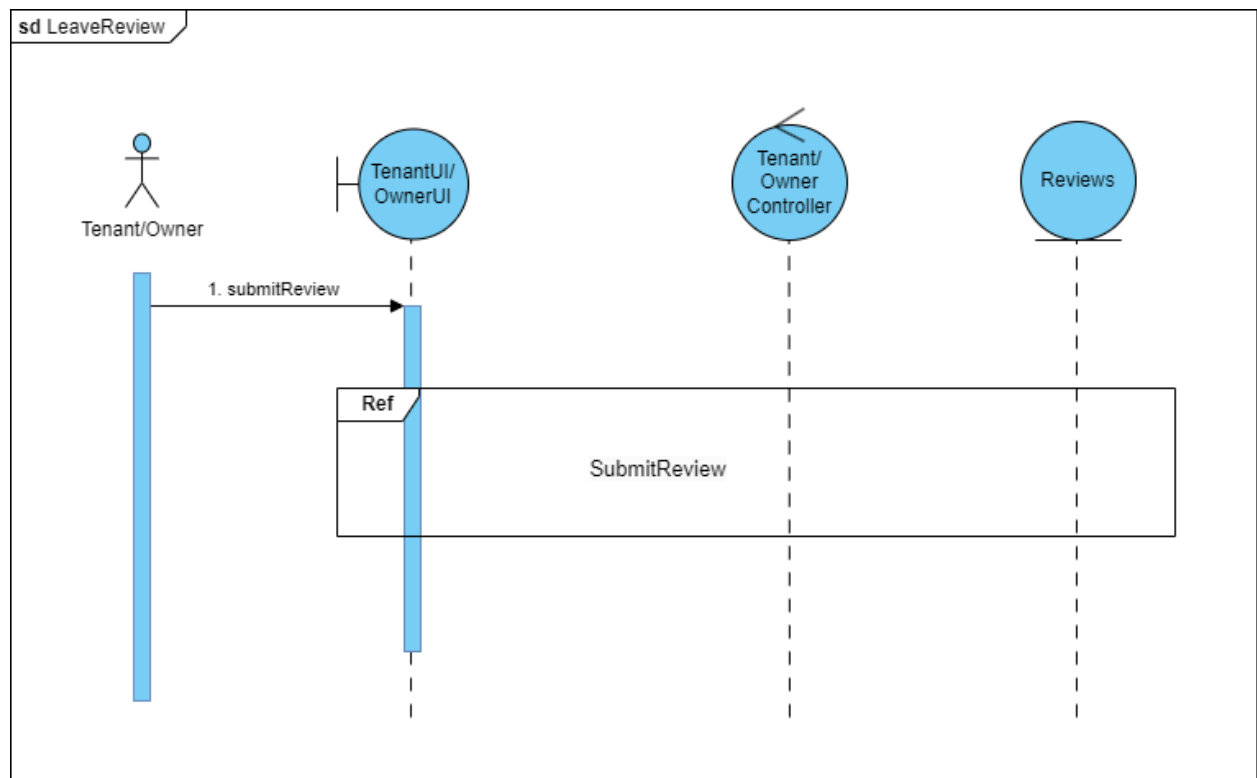
III.II MakeMonthlyPayment



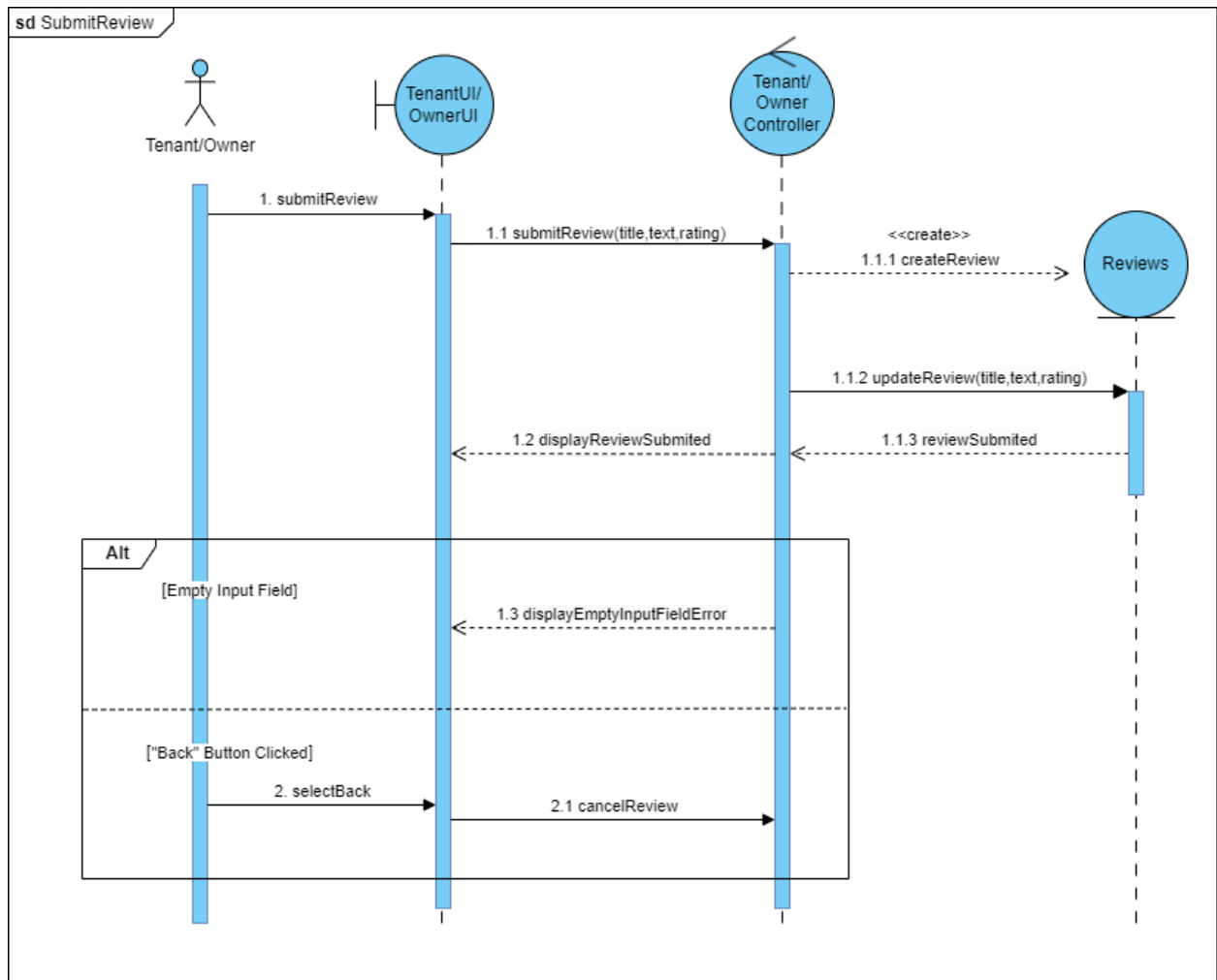
III.III ReviewTerminationRequest



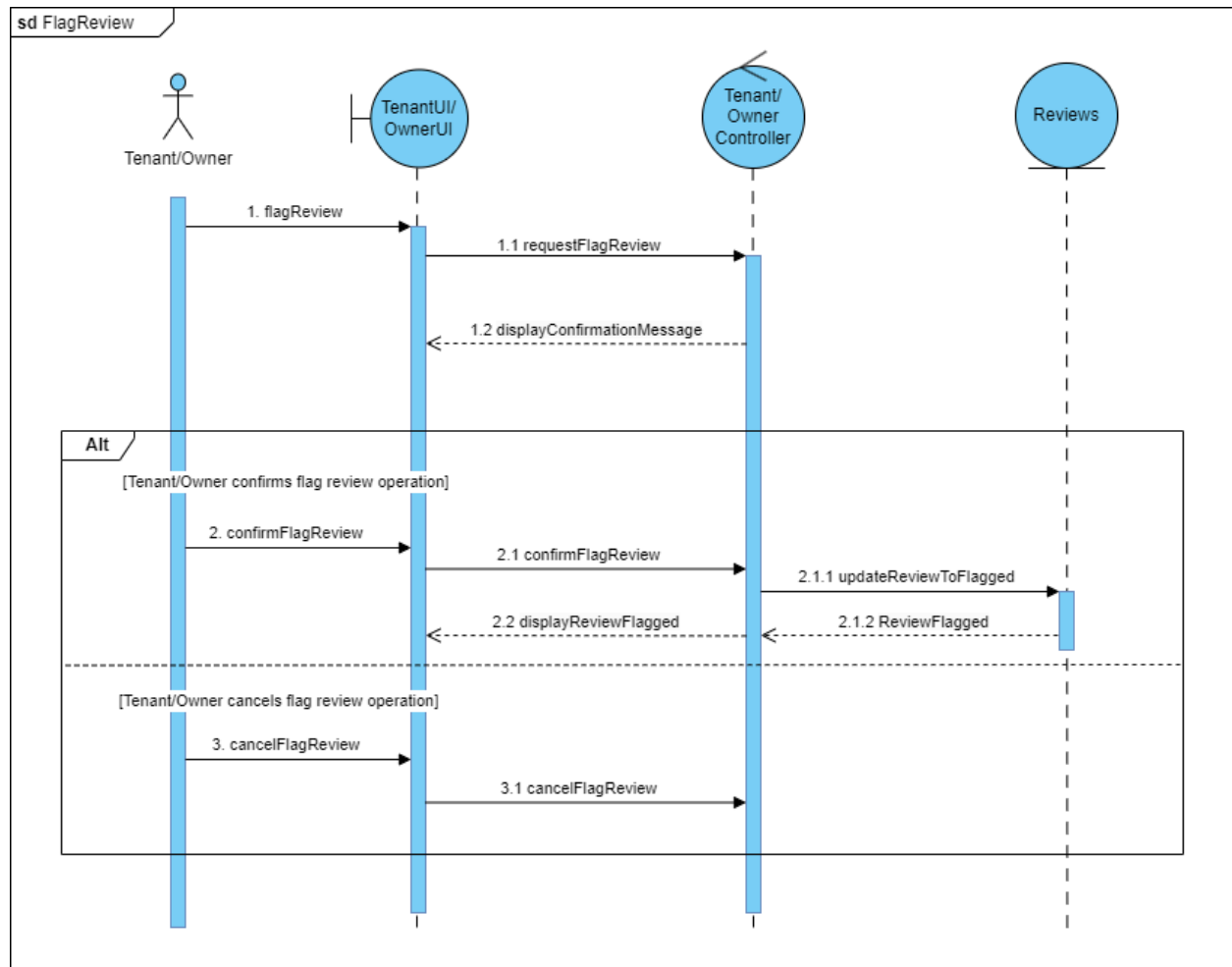
III.IV LeaveReview



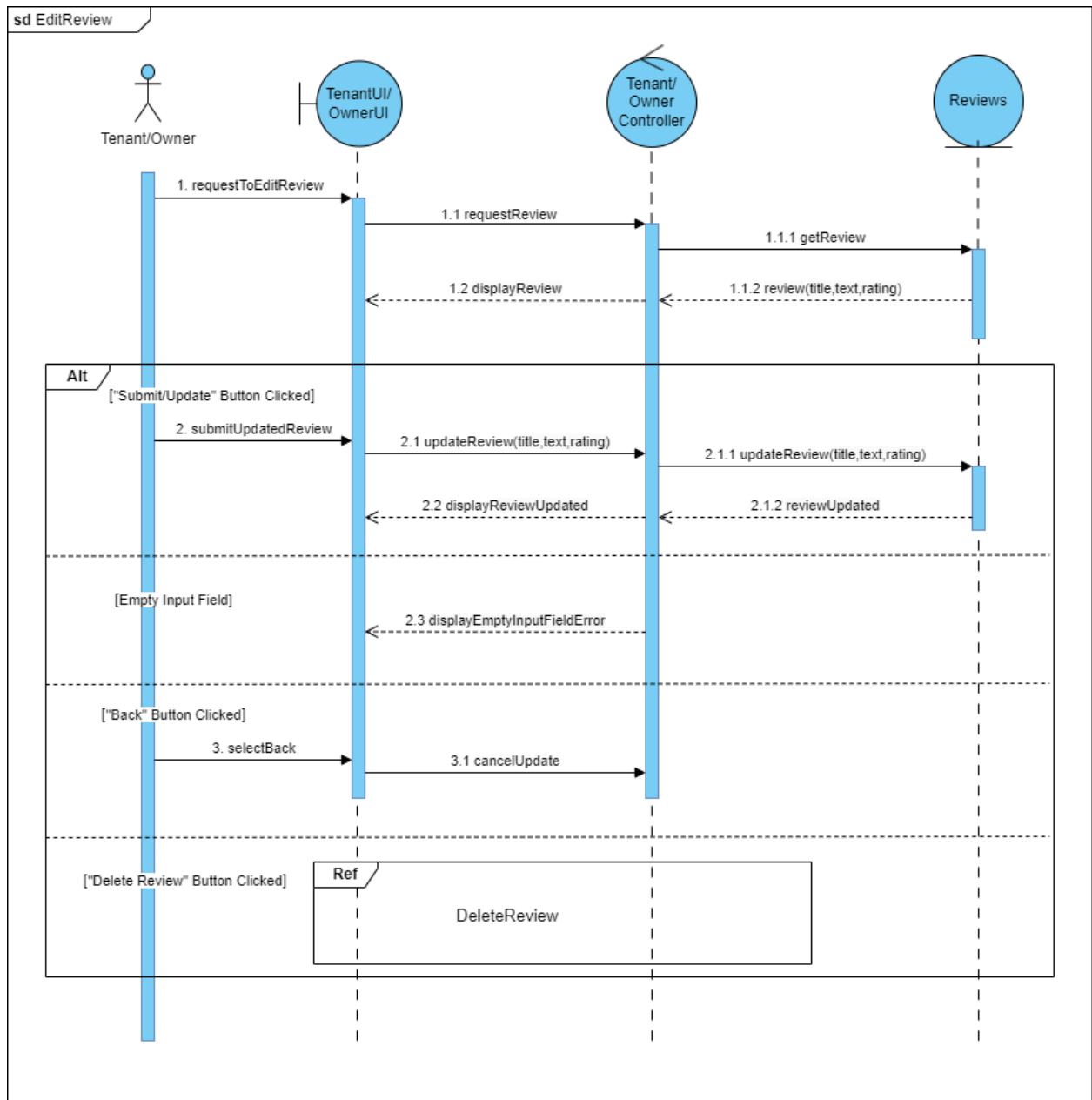
III.V SubmitReview



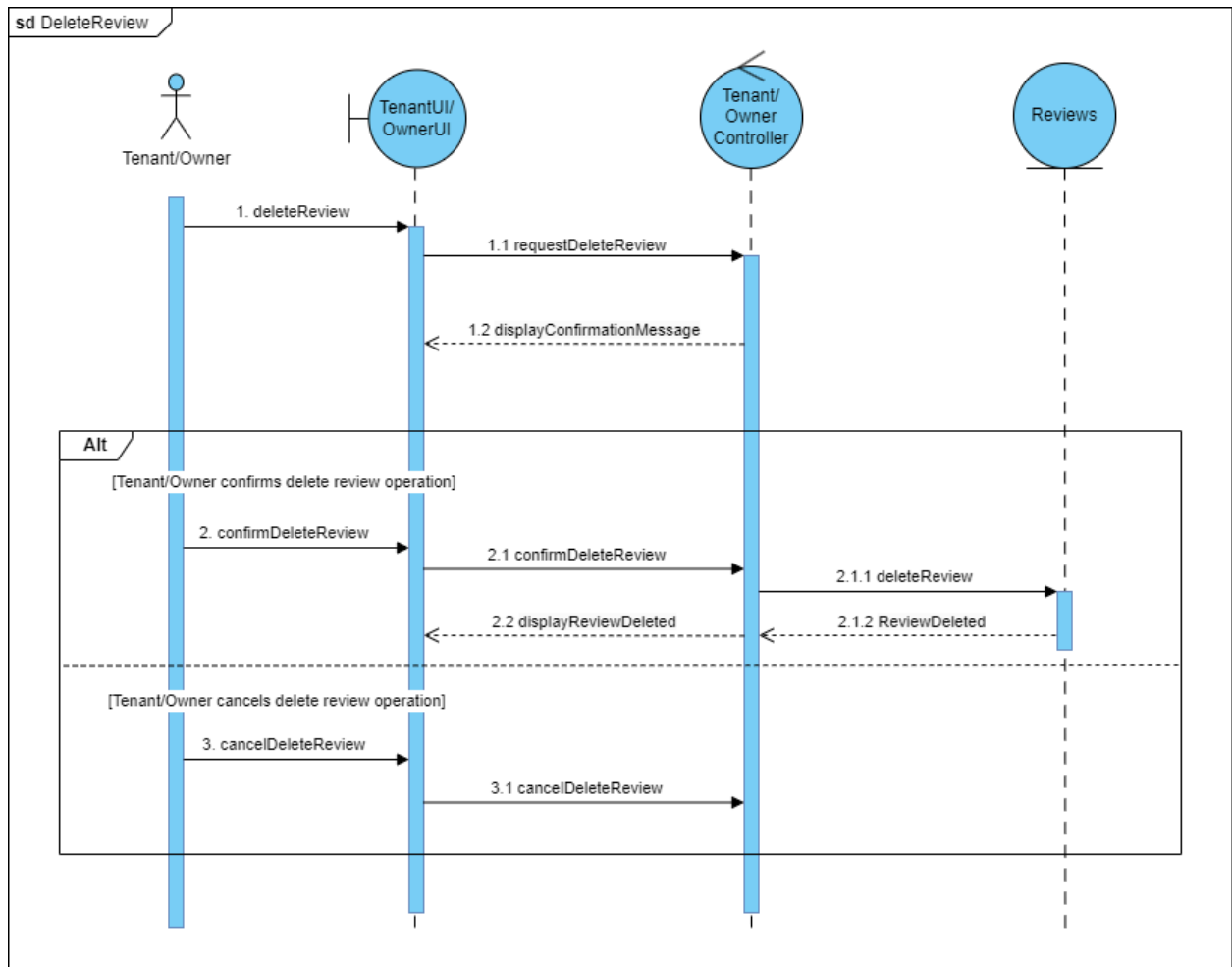
III.VI FlagReview



III.VII EditReview

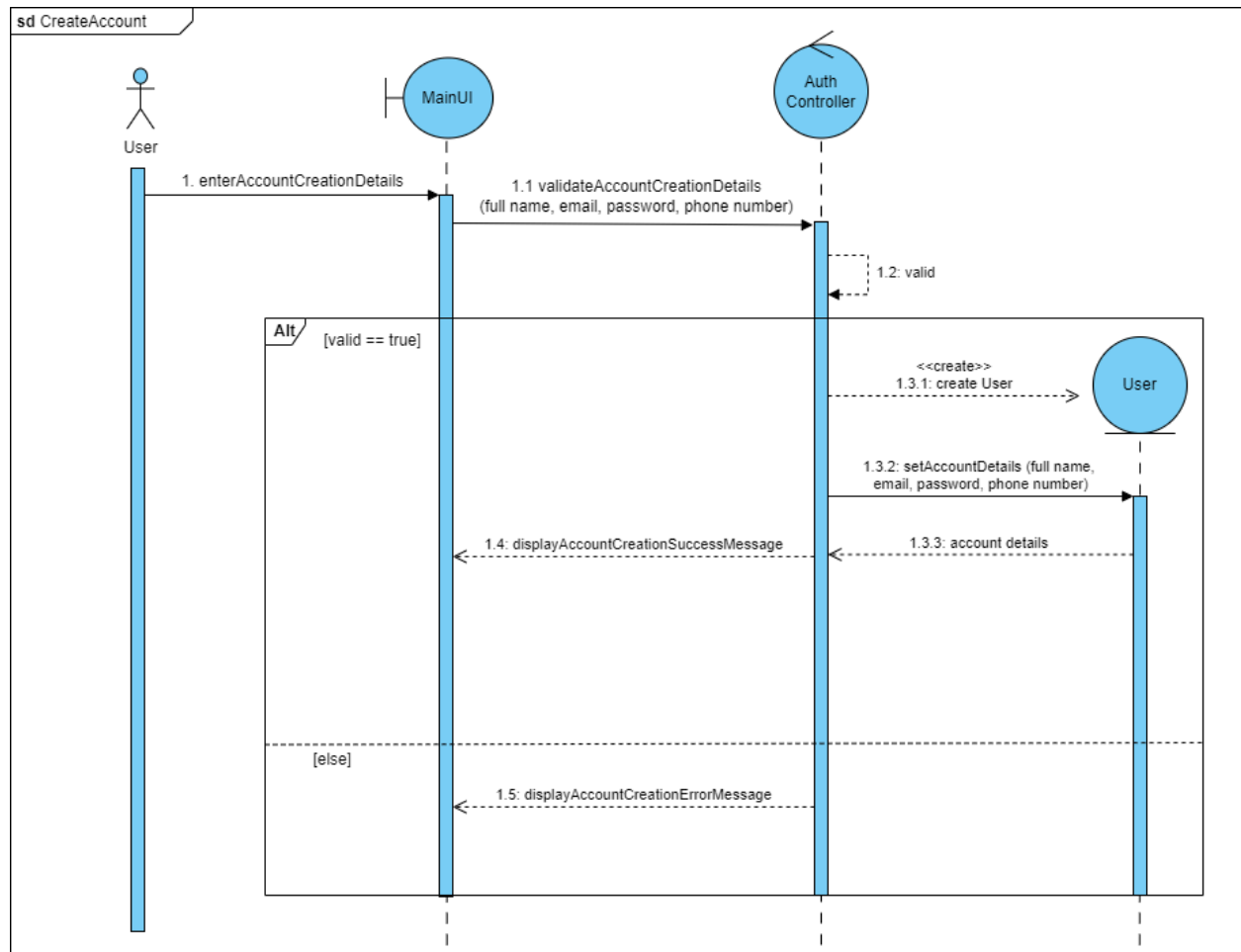


III.VIII DeleteReview

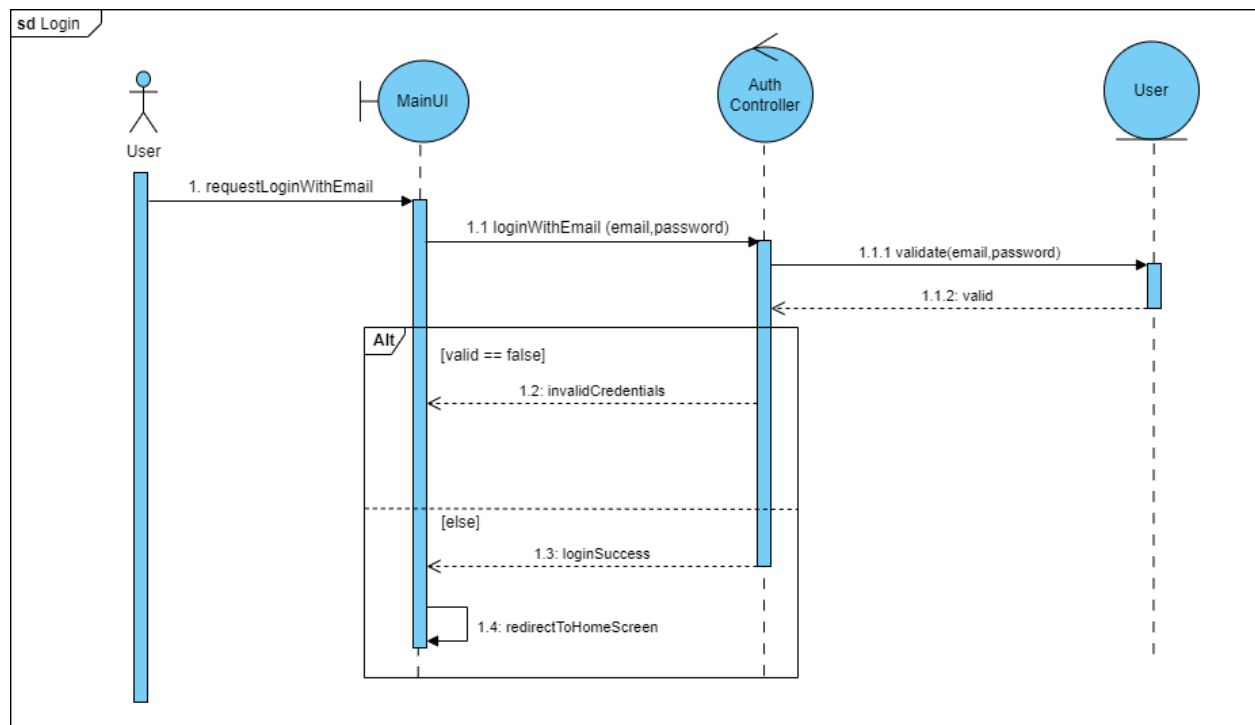


IV. For Use Cases under IV

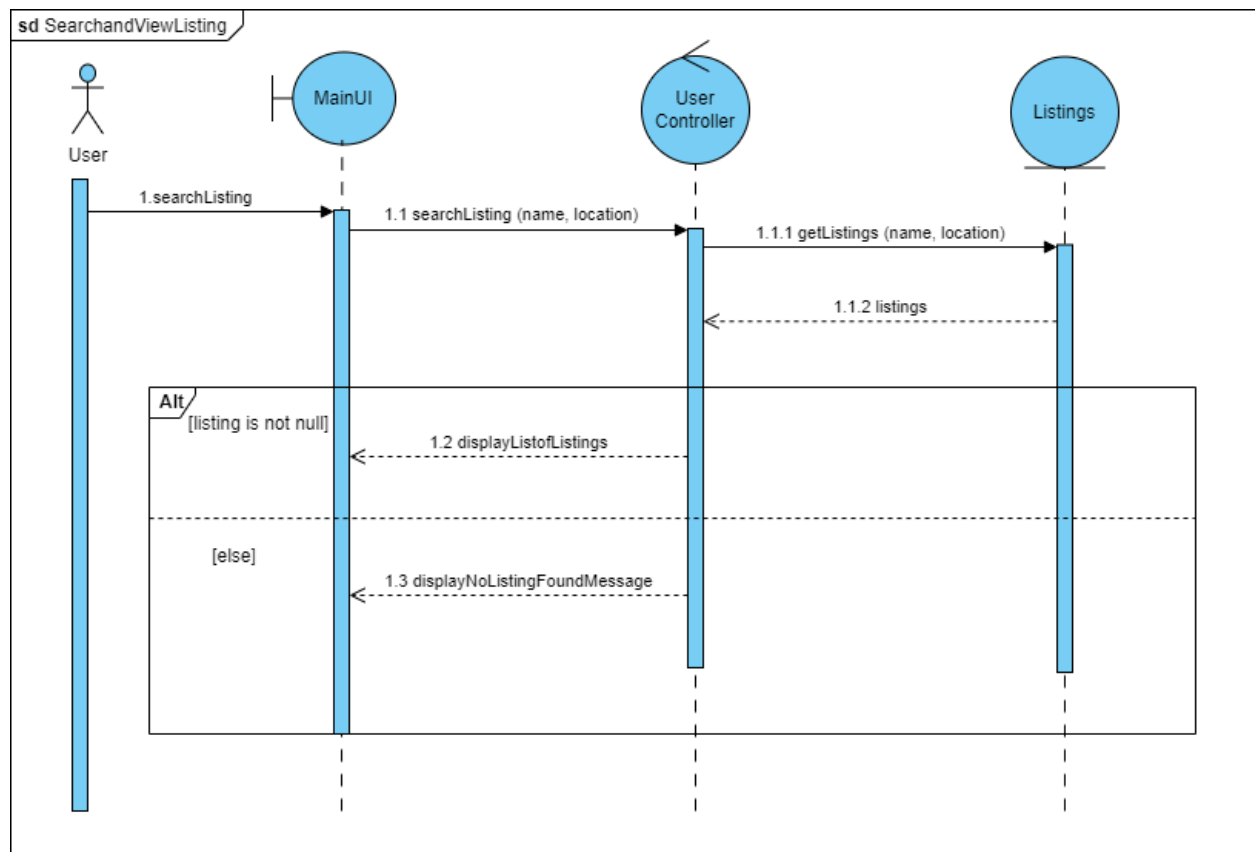
IV.I Create Account



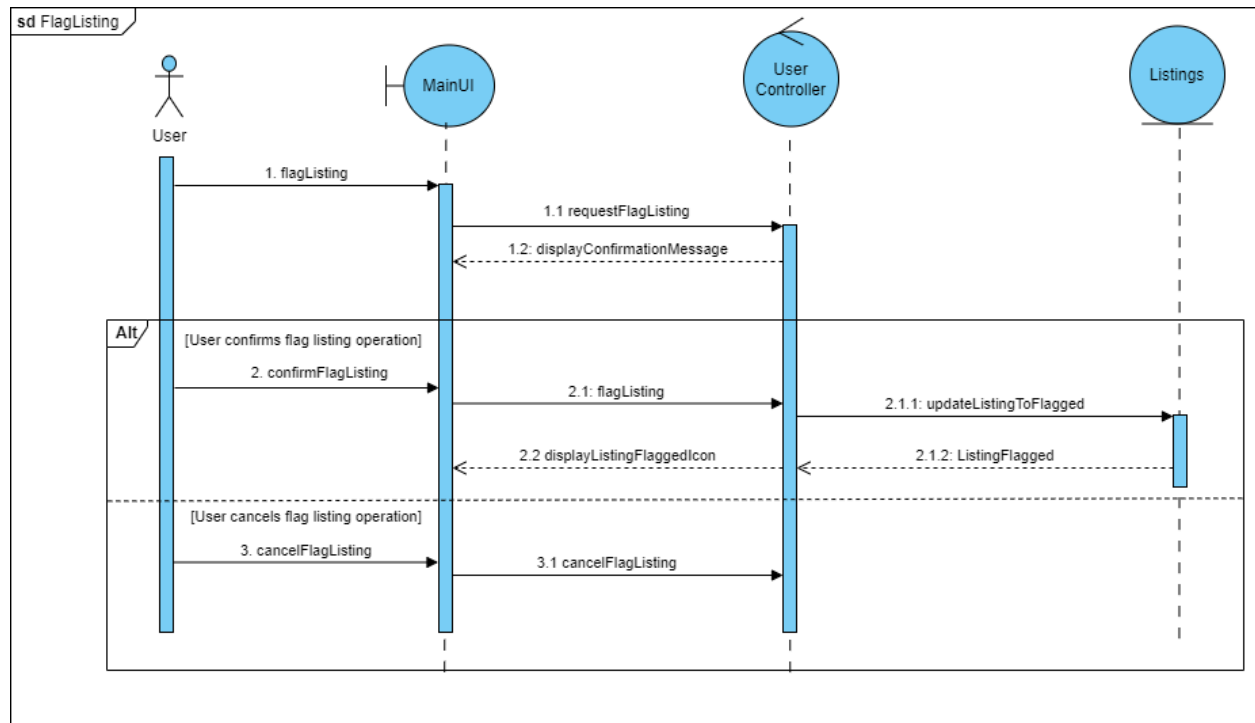
IV.II Login



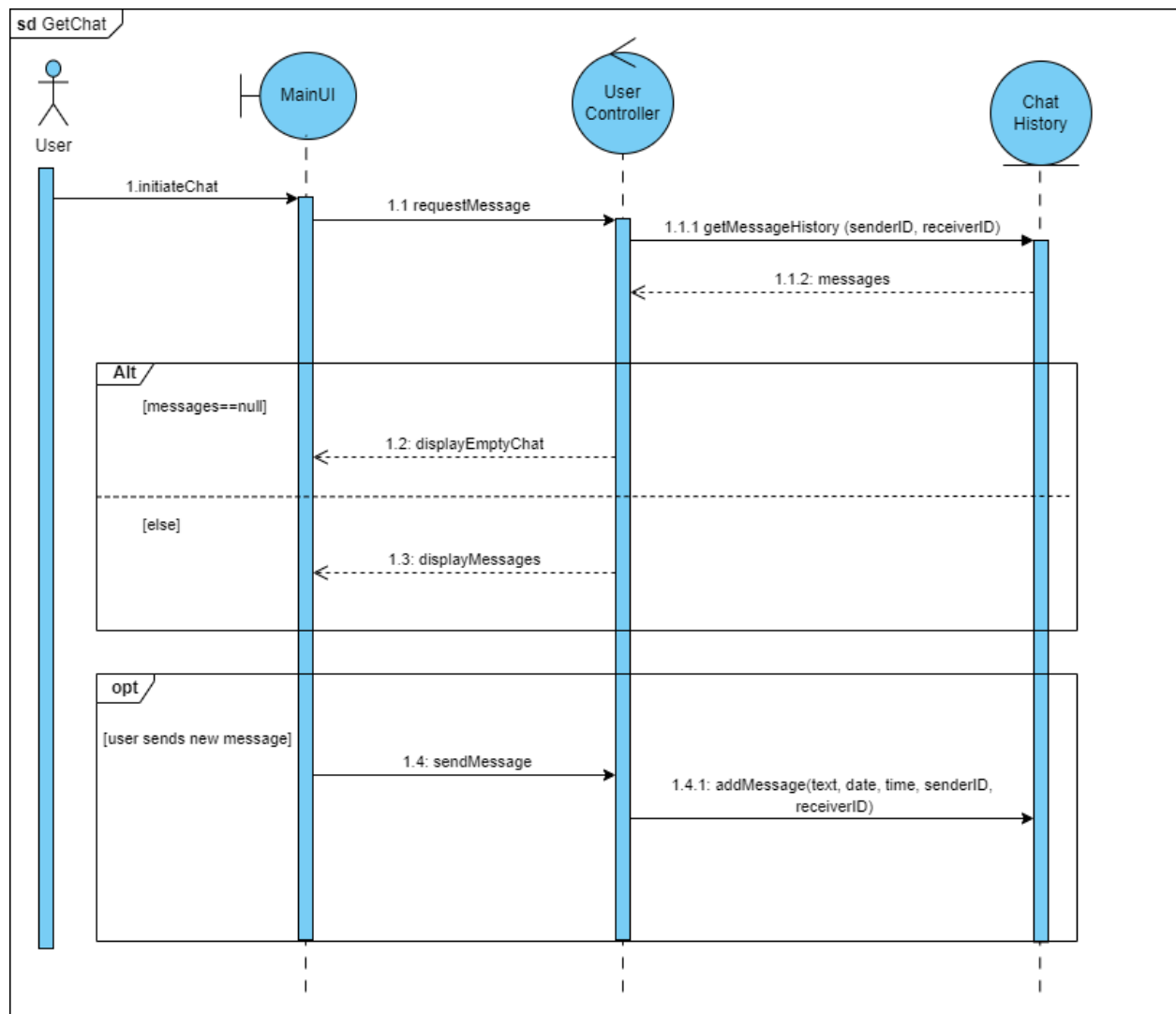
IV.III SearchAndViewListing



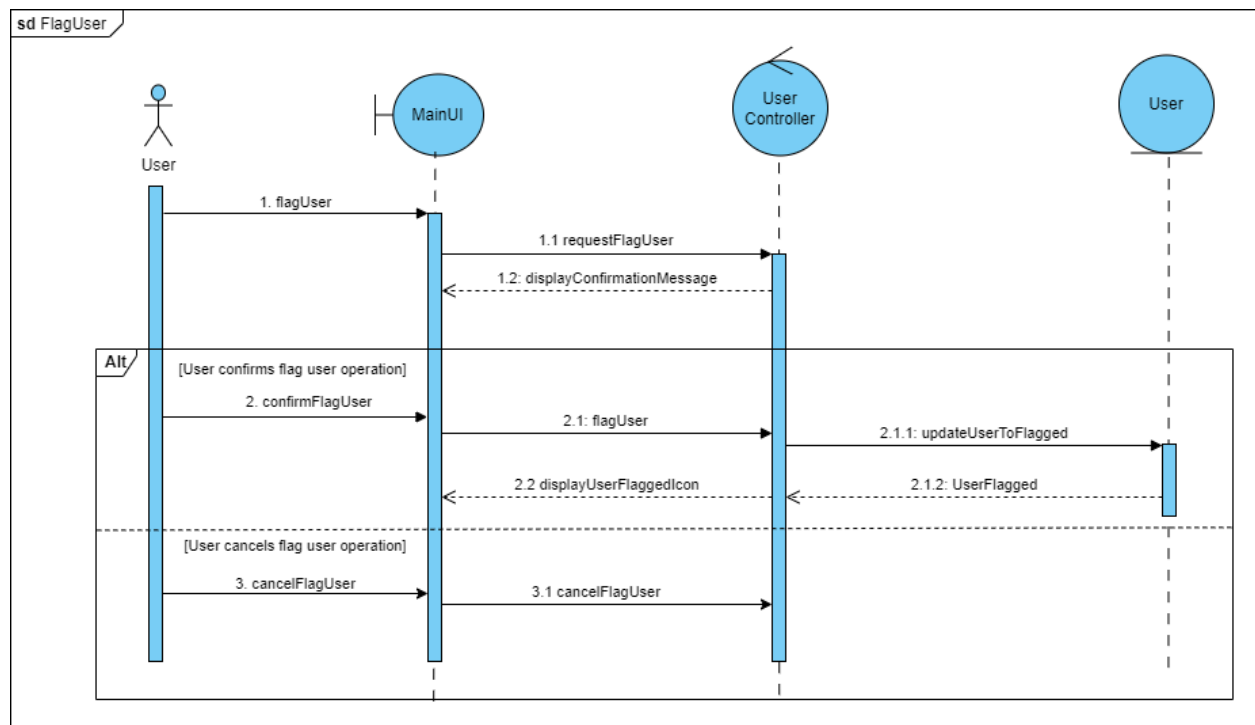
IV.IV FlagListing



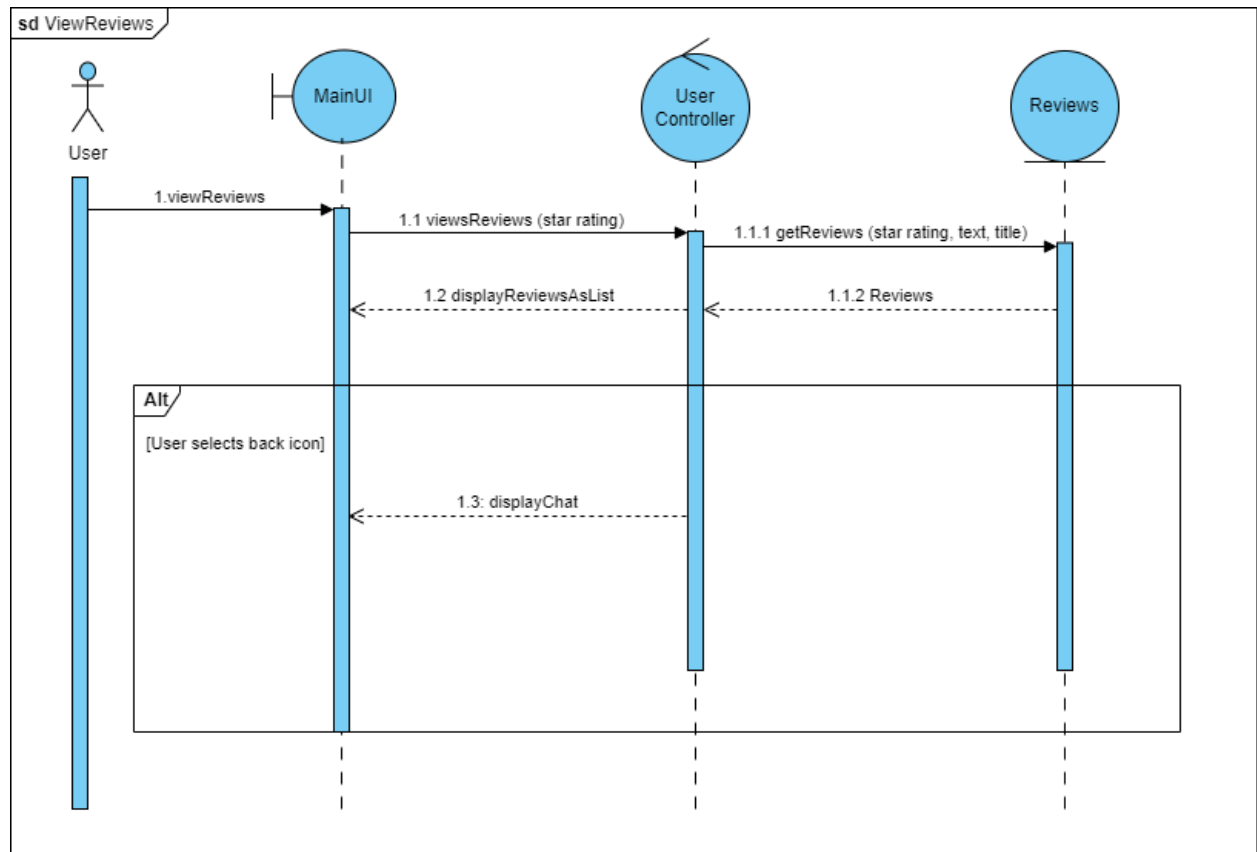
IV.V GetChat



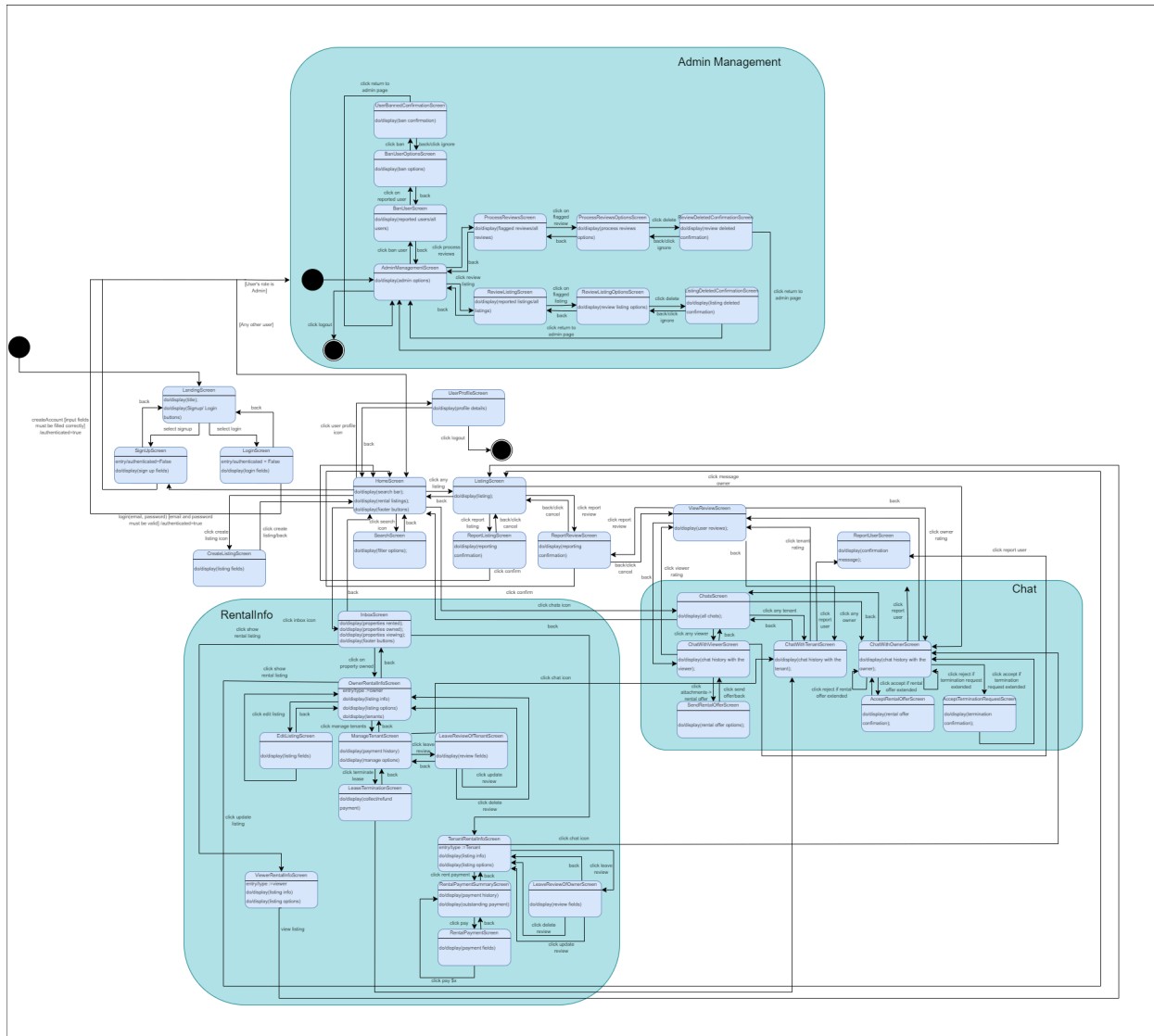
IV.VI FlagUser



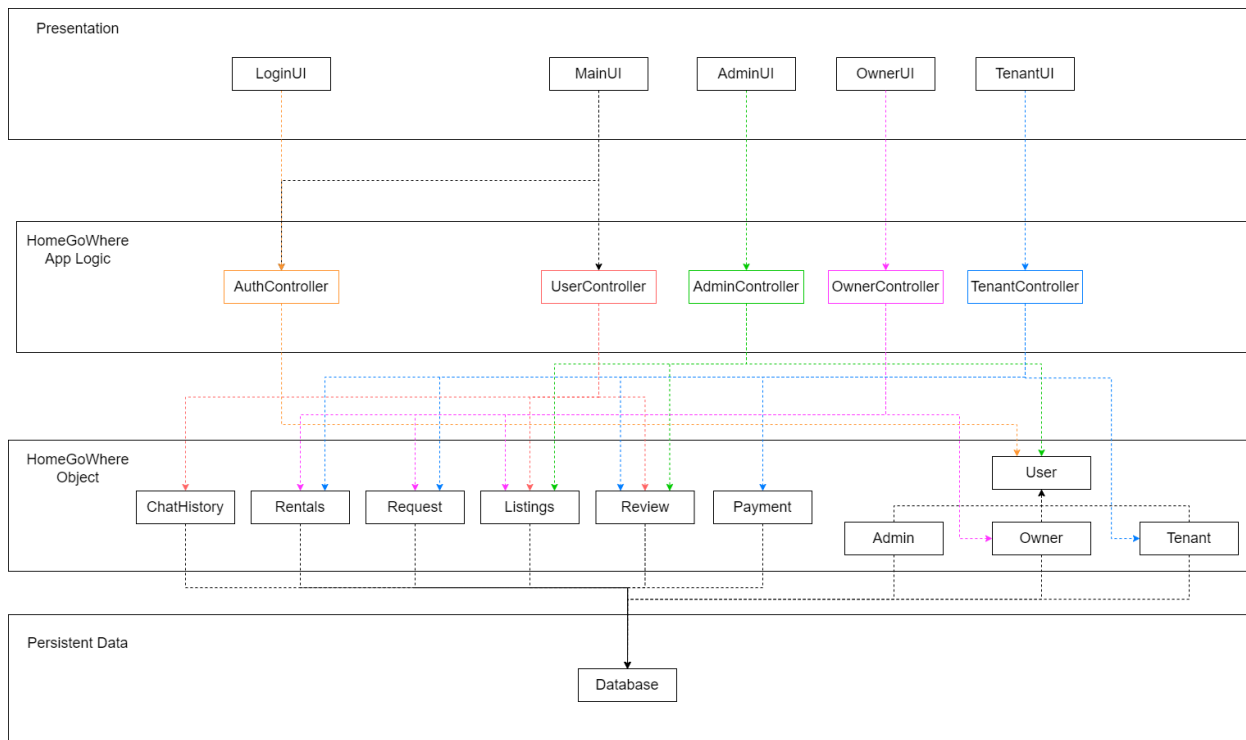
IV.VII ViewReviews



C. Initial Dialog map



3. System Architecture



Presentation Layer

This layer is largely responsible for Users to interact with HomeGoWhere. The different User Interfaces (UI) will then call the corresponding controllers to run the Application logic. This layer consists of:

1. LoginUI

Allow Users to log into the app through a basic username, email and password.

2. MainUI

Will use the services of UserController and AuthController.

3. AdminUI

AdminUI is part of MainUI, allowing admins to do their job through AdminController.

4. OwnerUI

OwnerUI is part of MainUI, allowing owners to do owner taskings through OwnerController.

5. TenantUI

TenantUI is part of MainUI, allowing tenants to do tenant taskings through TenantController.

App Logic Layer

This layer contains all the controller classes that will provide the presentation layer with its corresponding services. The controller classes will request for entities from the Object Layer when required in order to run its logic. This layer consists of:

1. AuthController

AuthController is called by MainUI for Users to log in to the application. This controller includes the log in and sign up methods enclosed within its logic.

2. UserController

UserController is called by MainUI to handle the search functions to return listing results and its corresponding information.

3. AdminController

AdminController contains the logic for administrators. This includes verifying flagged users, banning users and processing reviews.

4. OwnerController

OwnerController contains the logic for owners. The logic includes submitting a listing.

5. TenantController

TenantController contains the logic for tenants. The logic includes making monthly payments, submitting reviews and editing reviews.

App Object Layer

This layer contains the entity classes that will be called by the App Logic Layer to implement its logic. The entity classes are stored in the database of the persistent layer. This layer consists of:

1. ChatHistory

ChatHistory contains the messages and rental offer logs between Users and Owners or Tenants and Owners.

2. Rentals

Rentals contain the listingID, tenantID, status, leaseExpiry, depositPrice, paymentHistory, rentalPrice.

3. Request

Request contains the status, rentalID and refundAmount.

4. Listings

Listings contain the ownerID, tenantID, name, type, floor, unitNumber, location, postal, price, size, beds, bathroom, description, flagged.

5. Review

Review contains userID, rating, title, text for the review, reviewerID and flagged.

6. Payment

Payment contains rentalID, amount, date.

7. Admin

Admin contains adminID.

8. Owner

Owner contains ownerID and ownerName.

9. Tenant

Tenant contains tenantID and tenantName.

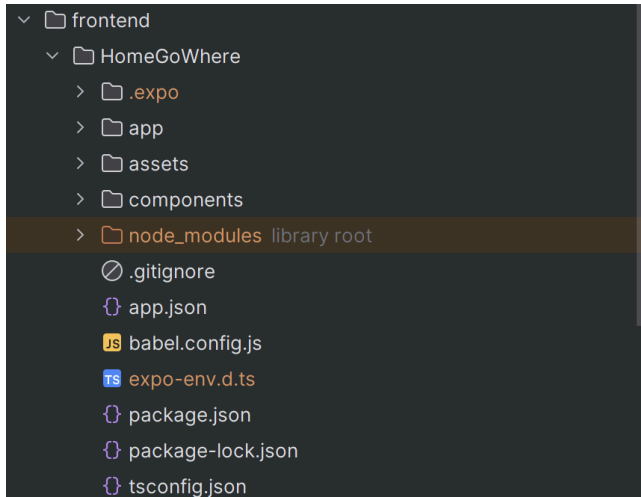
Persistent Data Layer

This layer contains the database that will store all of the entities.

4. Application Skeleton

A. Frontend

1. Built with React Native framework



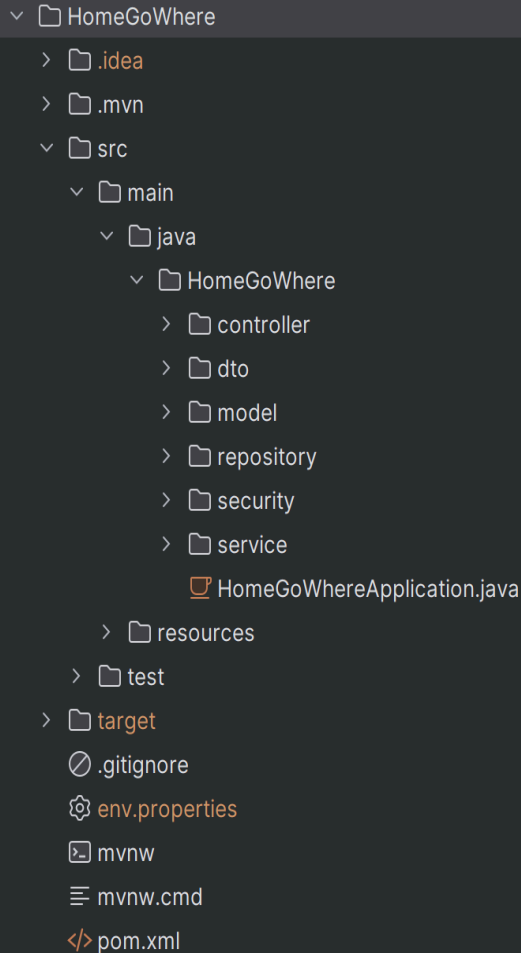
Frontend (React Native) mainly consists of the different User Interfaces (app), which are structured and categorised into different UI.

Assets is a folder which contains all of the images used in the app.

Components is a folder where we have our more commonly used items such as buttons.

B. Backend

1. Built with Spring Boot framework
2. Database used: Postgresql



```

  HomeGoWhere
  ├── .idea
  ├── .mvn
  ├── src
  │   ├── main
  │   │   ├── java
  │   │   │   ├── HomeGoWhere
  │   │   │   │   ├── controller
  │   │   │   │   ├── dto
  │   │   │   │   ├── model
  │   │   │   │   ├── repository
  │   │   │   │   ├── security
  │   │   │   │   └── service
  │   │   │   └── HomeGoWhereApplication.java
  │   │   └── resources
  │   └── test
  ├── target
  ├── .gitignore
  ├── env.properties
  ├── mvnw
  ├── mvnw.cmd
  └── pom.xml

```

Models: Contains the Business Objects.

Service: Contains the methods to access/modify the Business Objects.

Repository: Contains the specific Create, Read, Update, Delete (CRUD) by inheriting from JPA repository.

Controllers: Contains all the controllers that provide the detailed implementation of the various services. It also contains REST API endpoints for data communication between the frontend and backend.

Security: Contains all the security files in charge of ensuring logging in is secure with the use of JWT Authentication.

DTO: Contains all the Data Transferable objects of each model in order to allow communication between the API and our database server without exposing too much information

Target: Contains all classes built

Resources: Contains the file with all used live API keys

5. Appendix

Key Design Issues

A. Identifying and Storing Persistent Data

- Relational database
 - Users
 - **userid** (long), name (String), email (String), contact (String), photourl (String), flagged (int), password (String, encrypted)
 - Listings
 - **listingid** (long), owneruserid (= userid), tenantuserid (= userid), name (String), type (String), floor (int), unitnumber (String), location (String), postal (int), price (int), size (int), beds (int), bathroom (int), description (String), flagged (boolean)
 - Rentals
 - **rentalid** (long), listingid (= listingid), tenantuserid (= userid), rentalprice (long), depositprice (long), rentaldate (Date), leaseexpiry (Date), paymenthistory (String), status (String)
 - Payment
 - **paymentid** (long), rentalid (= rentalid), amount (Long), date (Date)
 - Request
 - **requestid** (long), rentalid (= rentalid), status (String), refundamount (long)
 - Reviews
 - **reviewid** (long), userid (= userid), reviewerid (= userid), rating (int), title (String), text (String), flagged (boolean)
 - ChatHistory
 - **messageid** (long), receiverid (= userid), senderid (= userid), message (String), date (Date), rentalid (= rentalid), requestid (=requestid)

B. Providing Access Control

- Encryption
 - Encrypted password to be stored inside the database

- Access Control

Actors	Users	Listings	Rentals
Owner	createUser() updateUser() flagUser()	getAllListings() flagListing() createListing() updateListing()	createRental() updateRental() getRental()
Tenant	createUser() updateUser() flagUser()	getAllListings() flagListing()	updateRental() getRental()
Admin	createUser() updateUser() getAllUsers() getFlaggedUsers() banUser()	getAllListings() createListing() updateListing() getFlaggedListings() deleteListing()	createRental() updateRental() getRental() getAllRentals()

Actors	Payment	Request	Reviews
Owner	getPayments()	createRequest() getRequest()	createReview() updateReview() getReview() flagReview()
Tenant	getPayments() createPayment()	getRequest() acceptRequest() denyRequest()	createReview() updateReview() getReview() flagReview()
Admin	getPayments() getAllPayments() createPayment()	createRequest() getRequest() getAllRequests()	createReview() updateReview() getReview() getAllReviews() getFlaggedReviews() deleteReview()

Actors	ChatHistory
Owner	makeChatHistory() getChatHistory()
Tenant	makeChatHistory() getChatHistory()
Admin	makeChatHistory() getChatHistory() getAllChatHistory()

C. Tech Stack

- Frontend
 - React Native
 - Expo
 - JavaScript
- Backend
 - Spring Boot (Java)
- Database
 - Postgresql