Course Code/Course Title

Group Number

Project Topic

Link to GitHub Repository

Instructors Name

Date of submission

|  |  |  |  |
| --- | --- | --- | --- |
| SN | Member’s Name | Registration | Team Role |
|  |  |  |  |
|  |  |  |  |

Introduction  
Brief Overview of the Project  
Studhome is a modern web-based platform designed to streamline the process of discovering, reserving, and managing student housing. Built as a RESTful API using Django and Django REST Framework (DRF), it serves as a backend for potential frontend applications (e.g., mobile apps or websites) that cater to students seeking affordable accommodations near universities. The system allows users to register, browse house listings filtered by room type (single, double, or apartment), location, and price, and interact with properties through features like saving favorites, reservation /physical tours, and securing reservations via integrated mobile money payments.

Problem Statement  
Finding suitable, affordable, and safe housing near universities is a major and stressful challenge for students. The current process is often fragmented and inefficient, characterized by:

* **Information Overload and Scattered Listings:** Students must scour multiple unreliable websites, social media groups, and physical bulletin boards, making it difficult to find and compare all available options in one place.
* **Lack of Trust and Verification:** It is challenging to verify the legitimacy of listings and landlords, leading to potential scams or misrepresented properties.
* **Inconvenient and Inefficient Processes:** Traditional methods of contacting landlords, scheduling physical tours, and securing payments (often via cash or bank transfers) are time-consuming, lack transparency, and are not secure.
* **Poor Financial Management for Students:** High upfront costs and a lack of integrated, student-friendly payment solutions like mobile money create significant barriers.

Studhome aims to solve these problems by providing a centralized, trusted, and digital platform that simplifies the entire student housing lifecycle.

**Objective and Purpose of your Platform**  
The primary objective of Studhome is to become the leading digital platform that seamlessly connects students with ideal housing options, eliminating the friction and anxiety associated with the search and reservation process.

**Purpose:**  
To create a secure, efficient, and user-centric ecosystem that:

1. **Centralizes Information:** Aggregates verified housing listings in a single, easy-to-navigate platform.
2. **Builds Trust:** Implements verification processes for both properties and users to ensure a safe and reliable community.
3. **Streamlines Operations:** Digitizes and automates key processes like search filtering, tour scheduling, and reservation management.
4. **Facilitates Secure Transactions:** Integrates a familiar and secure mobile money payment gateway to simplify financial transactions for students.

Key Features and Innovations

Target Audience  
Studhome’s platform is designed for two primary user groups:

1. **Primary Users (Tenants):**
   * **University and College Students:** This is the core audience. Specifically, both new (freshmen/international students) and returning students who are seeking affordable, convenient, and safe housing near their campus.
   * **Their Pain Points:** Lack of time, limited budget, need for a secure and simple process, and desire for a trustworthy source of information.

Project Management (Scrum)

Team Roles and Responsibilities

Overview of the Scrum Process

o Sprint Planning

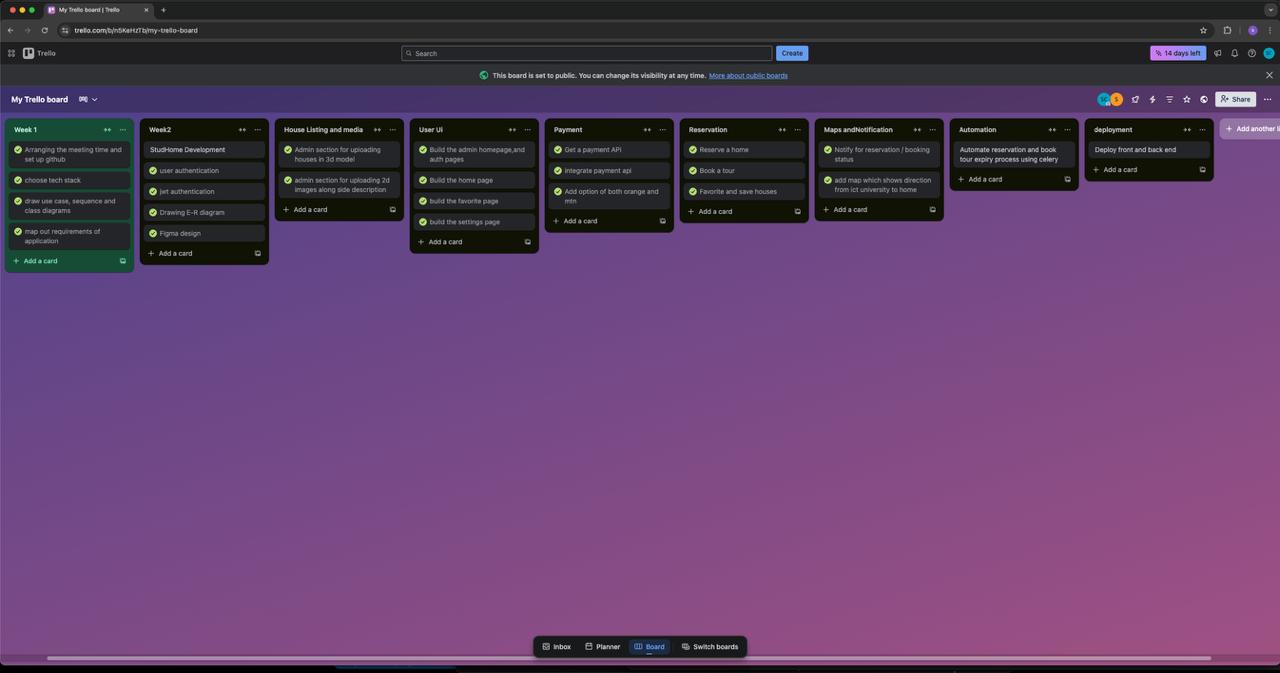
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| User Story | Task Description | Estimated Story Points | |  | **Assignee** | | --- | --- | | Status (To Do, In Progress, Review, Done) |
| **US1: Registration** | |  |  |  |  | | --- | --- | --- | --- | | 1.Design User database schema (name, phone, hashed password)  2. Implement /api/auth/register endpoint (phone as username) 3. Implement JWT token generation upon registration/login |  |  |  | |  |  |  |  |  | |  |  |  |  |  | |  |  |  |  |  | | 3 | Dev A | To Do |
| **US2: Profile Picture** | 1. Integrate Cloudinary SDK into backend 2. Design Profile schema to link user to image URL | 5 | Dev B | To Do |
| **US15: Deploy Backend** | 1. Set up Render account and connect GitHub repo 2. Configure environment variables (DB URL, JWT secret, Cloudinary keys) on Render 3.Test deployed API endpoints with Postman | 2 | Dev A | To Do |
| |  |  | | --- | --- | | **NFR: Database & Setup** |  | | 1. Set up production PostgreSQL database on Render 2. Create basic error handling middleware | 1 | DevA | To Do |
| **US3: Upload Property** | 1. Design Property database schema (location, price, desc, landlord FK) 2. Implement /api/properties POST endpoint (protected, for landlords) 3. Add server-side validation for property data | 5 | Dev B | To Do |
| **US4: Media to Cloudinary** | 1. Extend Property schema to store an array of image URLs 2. Modify property upload endpoint to handle multiple image files 3. Implement logic to upload images to Cloudinary and save URLs 4. Write tests for the property creation flow | 8 | Dev A | To Do |
| **NFR: Error Handling** | 1. Enhance error handling for file uploads and database operations | 2 | Dev B | To Do |

o Daily Stand-ups

o Sprint Reviews and Retrospectives

o Product backlog and sprint backlog

Tools Used for Workflow Management



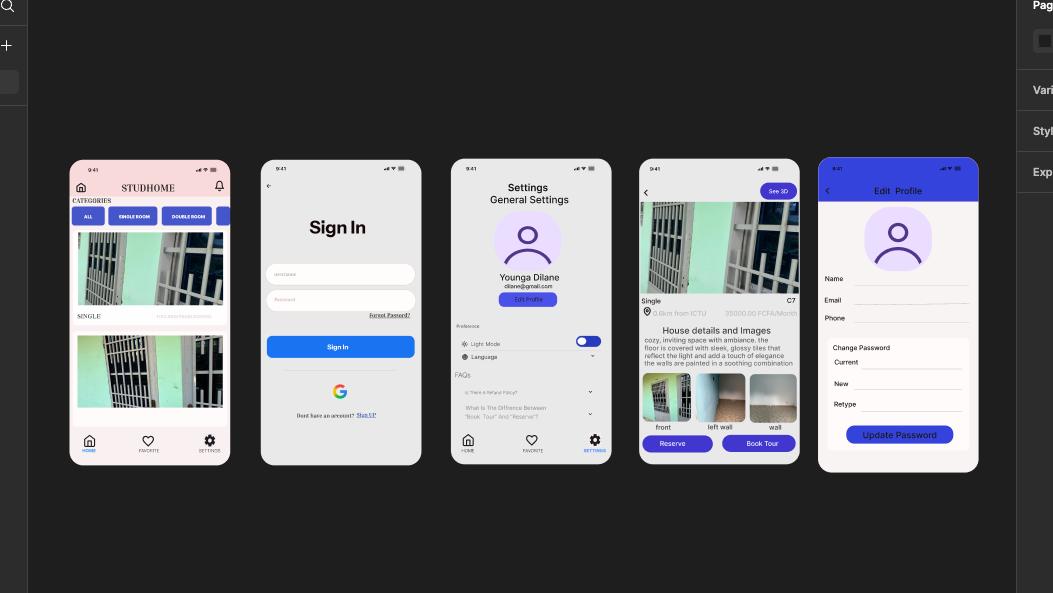
Challenges Faced and How They Were Addressed  
  
1. Onboarding a New Team Member with Limited Technical Experience  
A new developer joined the team with enthusiasm but lacked the technical experience and context of the project. There was a significant risk of this developer becoming overwhelmed, disengaged, or "lost," which would slow down the team's overall velocity and impact morale.  
**Solution**  
**Enhanced Daily Stand-ups:** The Scrum Master ensured the daily stand-up was a safe space for the new developer to be transparent about blockers. Questions like "I'm stuck on X" were met with immediate support ("I can help you with that right after this meeting") rather than judgment.

### 2. Landlord Refusal to 3D Scanning solution: Contract negotiation that is an agreement with the landlords to bring more tenants to available rooms

Evidence of Scrum Process

Application Design and Architecture

 Overview of Front-end Design

o UI/UX Design Principles  


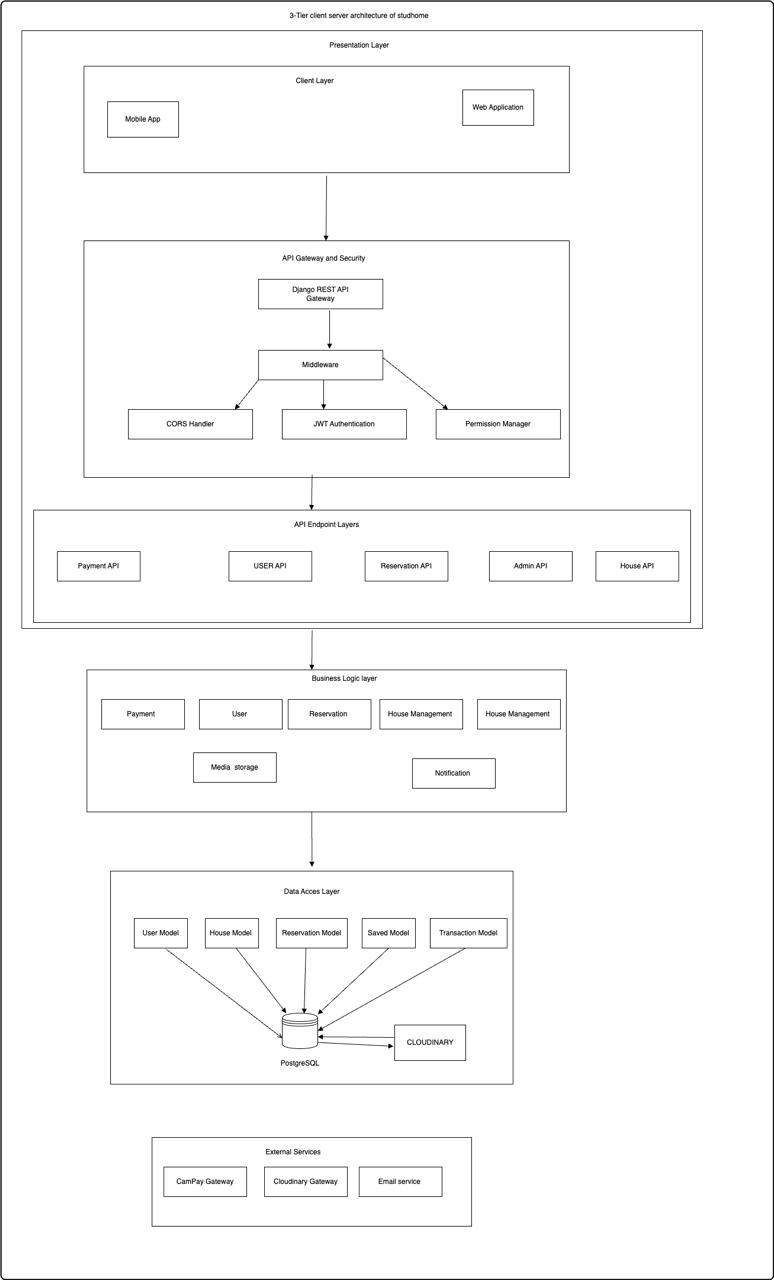
System Architectural design process  
Description

Functional Requirements

1. User Management

\* User Registration: Allow new users to create accounts with username, email, phone number, and password  
\* User Authentication: Enable users to login and obtain JWT tokens for session management  
\* User Profile Management: Users can view and update their profile information (username, phone number)  
\* Password Management: Users can change their passwords with proper validation  
2. House Management  
\* House Listing: Display available houses with filtering by room type (single, double, apartment)  
\* House Details: Provide detailed information about specific houses including media, price, location  
\* House Creation: Admin users can create new house listings with media uploads  
\* House Updates: Admin users can modify existing house information  
\* House Media Management: Support for uploading images (up to 6) and 3D models (up to 1) per house  
\* House Removal: Soft delete functionality to remove houses from frontend display  
3. Reservation System  
\* House Reservation: Users can reserve houses for a specified period (7 days default)  
\* Reservation Tracking: Users can view their active reservations  
\* Reservation Expiry: Automatic expiry handling for reservations  
\* Reservation Status: Track whether houses are reserved and by whom  
4. Tour Booking System  
\* Tour Booking: Users can book tours for available houses  
\* Tour Restrictions: Prevent tour bookings for houses reserved by other users  
\* Tour Tracking: Users can view their booked tours  
5. Payment Processing  
\* Payment Initiation: Integration with CamPay for mobile money payments  
\* Payment Verification: Real-time payment status verification  
\* Payment Webhooks: Handle payment status updates from payment provider  
\* Transaction History: Track all user transactions with detailed information  
6. Saved Homes Feature  
\* Save Houses: Users can save houses to their favorites list  
\* Unsave Houses: Remove houses from saved list  
\* Saved Homes Listing: View all saved houses  
7. Notification System  
\* Email Notifications: Send confirmation emails for successful payments and reservations  
\* Payment Status Updates: Notify users of payment confirmations

Component to architecture design  
1. User Management Component.  
2. Reservation System Component.  
3. Tour Booking System Component.  
4. Payment Processing Component.  
5. Saved Homes Feature Component.  
6. Notification System Component.  
  
Description of architecture  
Architectural diagram



### Description of each component and its role 1. User Management Component **Authentication: Handles user login and registration Authorization: Determines what the user can do Profile management:** 2. Reservation System Component Handles the logic for a student to reserve an available home (US6). It checks if the property is not already reserved or booked. 3.Tour Booking System Component This component handles the process of scheduling and managing physical property tours, which involves a financial transaction. 4. Payment Processing Component This is a critical, standalone service dedicated to handling all financial transactions securely. 5. Saved Homes Feature Component This component manages a user's personal list of preferred properties. 6. Notification System Component The Payment component receives a "success" webhook and triggers the Notification System to send Alice a confirmation SMS

### 

Non-Functional Requirements and Their Implementation

1. Security

Requirements:  
\* Secure authentication and authorization  
\* Protection against unauthorized access  
\* Secure payment processing  
\* Data validation and sanitization

Implementation:  
\* JWT Authentication: Uses rest\_framework\_simplejwt for stateless authentication  
\* Permission Classes: Implements role-based access control (IsAuthenticated, IsAdminUser, AllowAny)  
\* Password Security: Built-in Django password validation and hashing  
\* Input Validation: Django REST Framework serializers with field validation  
\* CORS Configuration: Controlled cross-origin resource sharing  
\* Environment Variables: Sensitive data like API keys stored in environment variables  
# Security implementation examples  
REST\_FRAMEWORK = {  
 'DEFAULT\_AUTHENTICATION\_CLASSES': [  
 'rest\_framework\_simplejwt.authentication.JWTAuthentication',  
 ],  
 'DEFAULT\_PERMISSION\_CLASSES': [  
 'rest\_framework.permissions.IsAuthenticated',  
 ],  
}

# Environment variable usage  
  
CAMPAY\_USERNAME = config('CAMPAY\_USERNAME')

2. Scalability  
Requirements:  
\* Handle multiple concurrent users  
\* Efficient database operations  
\* Media storage scalability  
Implementation:  
\* Database Indexing: Strategic indexes on frequently queried fields  
\* UUID Primary Keys: Distributed system-friendly identifiers  
\* Cloud Media Storage: Cloudinary integration for scalable media storage  
\* Pagination Support: Built-in DRF pagination capabilities  
# Database indexing examples  
class Meta:  
 indexes = [  
 models.Index(fields=['username', 'email']),  
 models.Index(fields=['house\_name', 'room\_type']),  
 models.Index(fields=['payment\_date']),  
 ]

3. Performance  
Requirements:  
\* Fast response times  
\* Efficient media handling  
\* Optimized database queries  
Implementation:  
\* Database Optimization: Proper indexing and field constraints  
\* Media CDN: Cloudinary CDN for fast media delivery  
\* Selective Field Loading: Serializers only include necessary fields  
\* Query Optimization: Foreign key relationships properly handled  
# Performance optimization examples  
class HouseSerializer(serializers.ModelSerializer):  
 class Meta:  
 model = House  
 fields = ['house\_id', 'house\_name', 'room\_type', 'lat', 'lng', 'media', 'is\_reserved', 'price',  
'description', 'reservation\_status']

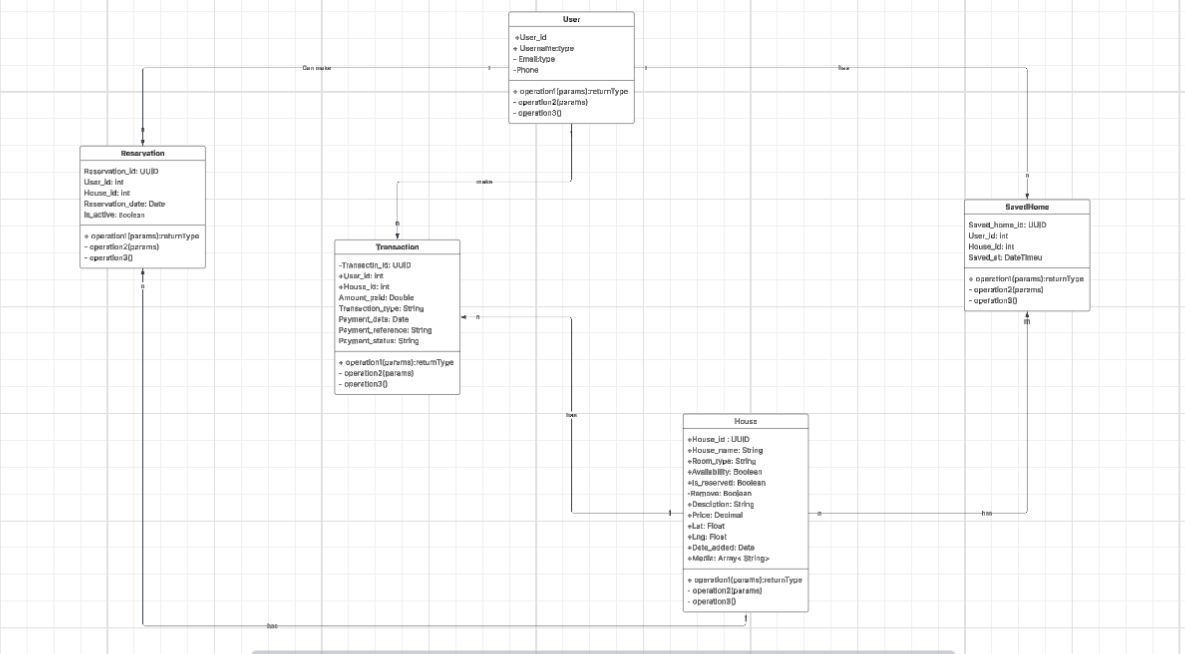
4. Reliability  
Requirements:  
\* Data consistency  
\* Error handling  
\* Transaction integrity  
Implementation:  
\* Database Constraints: Unique constraints, foreign key relationships  
\* Error Handling: Try-catch blocks with proper error responses  
\* Data Validation: Multiple levels of validation (model, serializer, view)  
\* Atomic Transactions: Proper database transaction handling  
# Reliability implementation examples  
try:  
 payment\_response = campay.initCollect({...})  
except Exception as e:  
 return Response({'error': f'Failed to initiate payment: {str(e)}'},   
 status=status.HTTP\_400\_BAD\_REQUEST)

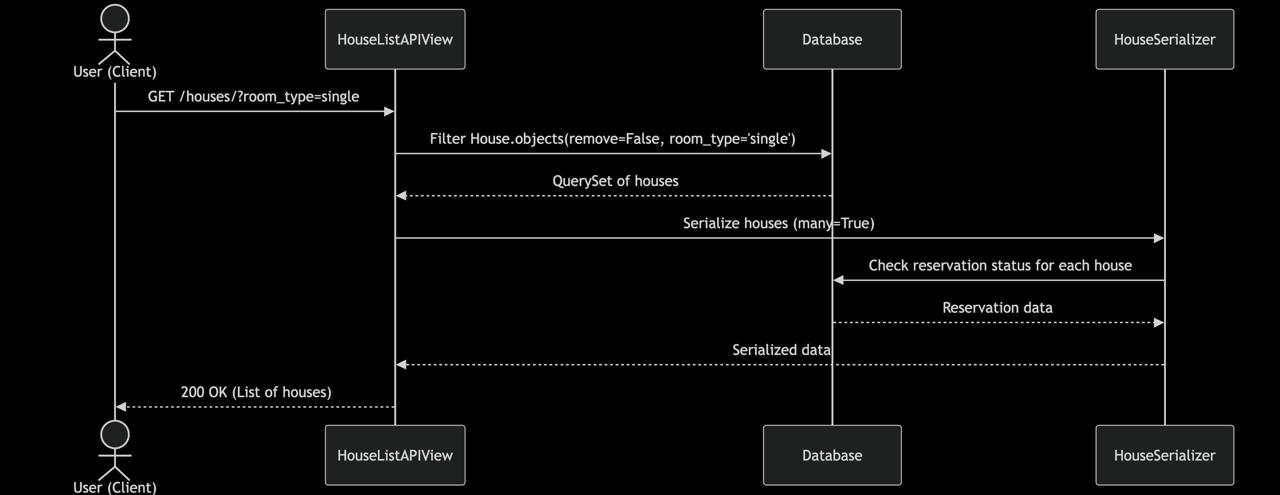
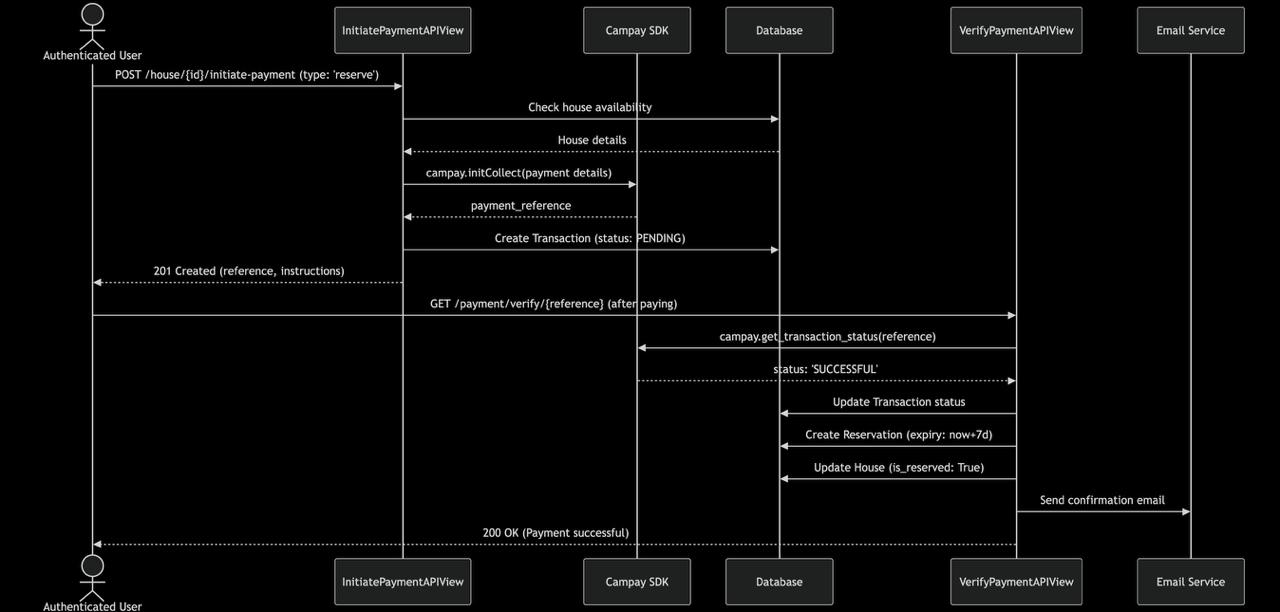
5. Maintainability  
Requirements:  
\* Clean code architecture  
\* Separation of concerns  
\* Documentation  
\* Modular design  
Implementation:  
\* Django Best Practices: Proper MVC pattern with models, views, serializers  
\* Code Organization: Separate files for models, views, serializers, admin  
\* Logging: Integrated logging system for debugging and monitoring  
\* Configuration Management: Environment-based settings  
# Maintainability examples  
logger = logging.getLogger(\_name\_)  
logger.info(f"User registration response: {response\_data}")

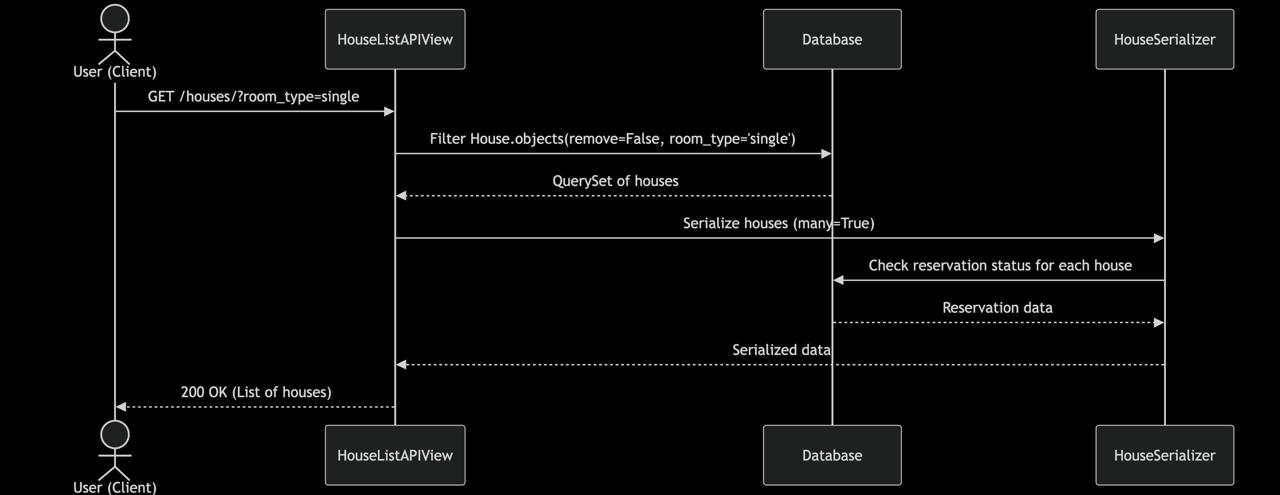
6. Usability  
Requirements:  
\* Clear API responses  
\* Consistent data formats  
\* User-friendly error messages  
Implementation:  
\* RESTful API Design: Standard HTTP methods and status codes  
\* Consistent Response Format: Standardized JSON responses  
\* Descriptive Error Messages: Clear error descriptions for users  
\* API Documentation: Well-structured URL patterns

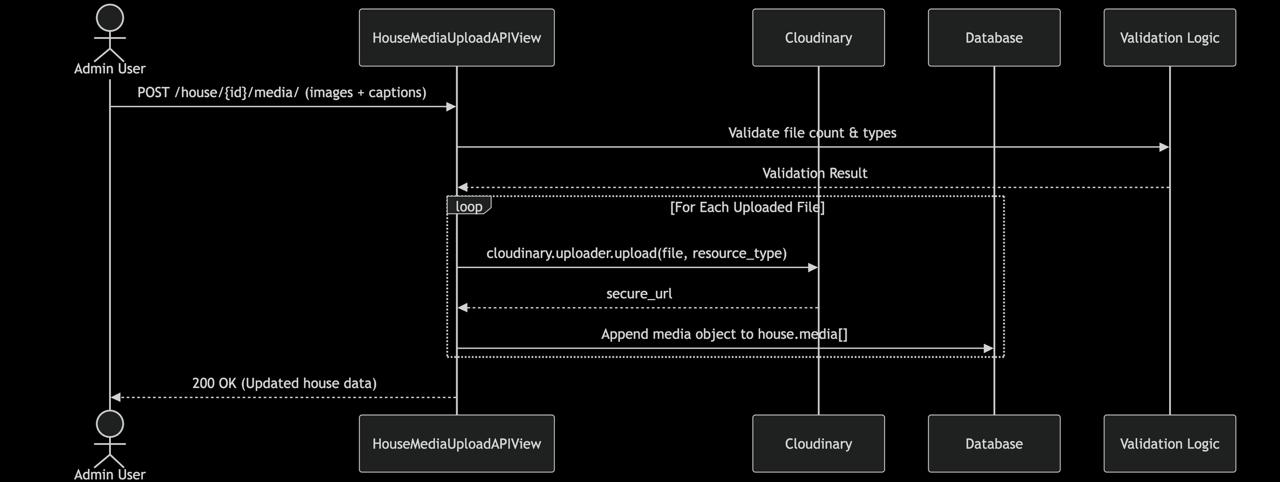
7. Compatibility  
Requirements:  
\* Cross-platform support  
\* API versioning  
\* Mobile compatibility  
Implementation:  
\* REST API: Platform-agnostic HTTP-based API  
\* CORS Support: Cross-origin request handling for web clients  
\* Mobile-Friendly: Phone number field support, appropriate data formats  
\* Standard Formats: JSON for all data exchange  
  
Areas for Improvement  
1. Caching: No caching implementation visible for frequently accessed data  
2. Rate Limiting: No API rate limiting implemented  
3. Monitoring: Limited monitoring and alerting capabilities  
4. Testing: No visible test implementation  
5. API Versioning: No explicit API versioning strategy  
6. Documentation: Could benefit from automated API documentation (e.g., Swagger)  
7. Backup Strategy: No explicit backup and recovery procedures defined

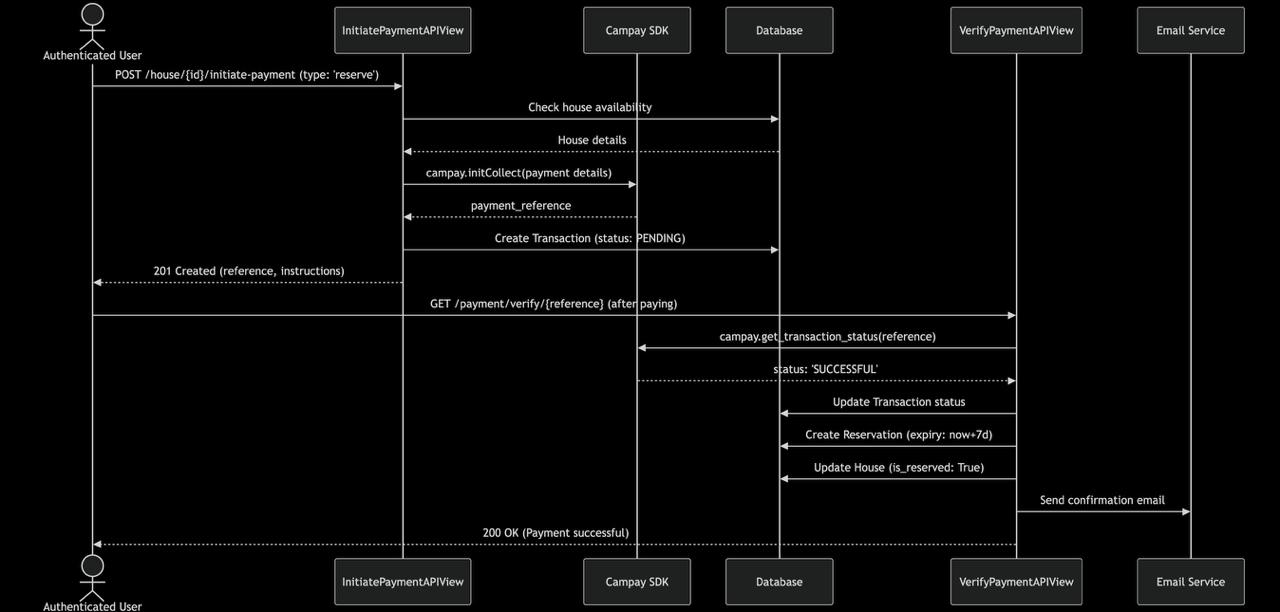
Pros and cons of architecture  
  
LLD- UML Design of system  
A Use Case Diagram of system

Class Diagram  


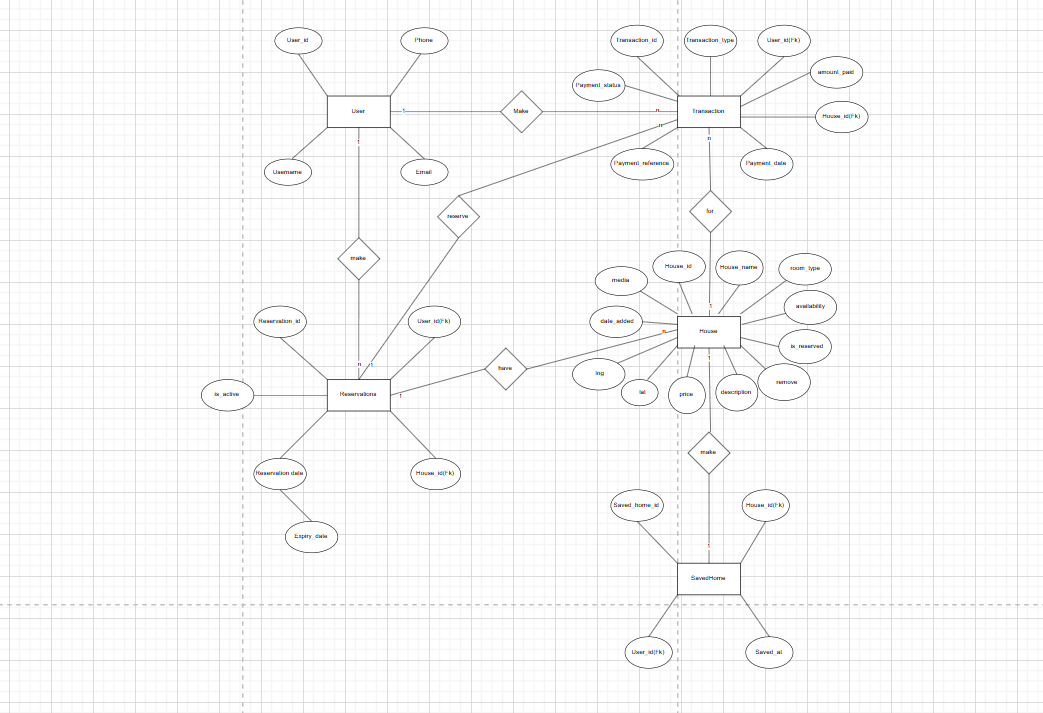
Sequence diagram  
  
1. User registration ans jwt authentication  
  
  
2. User browsing and filtering house listing  
  


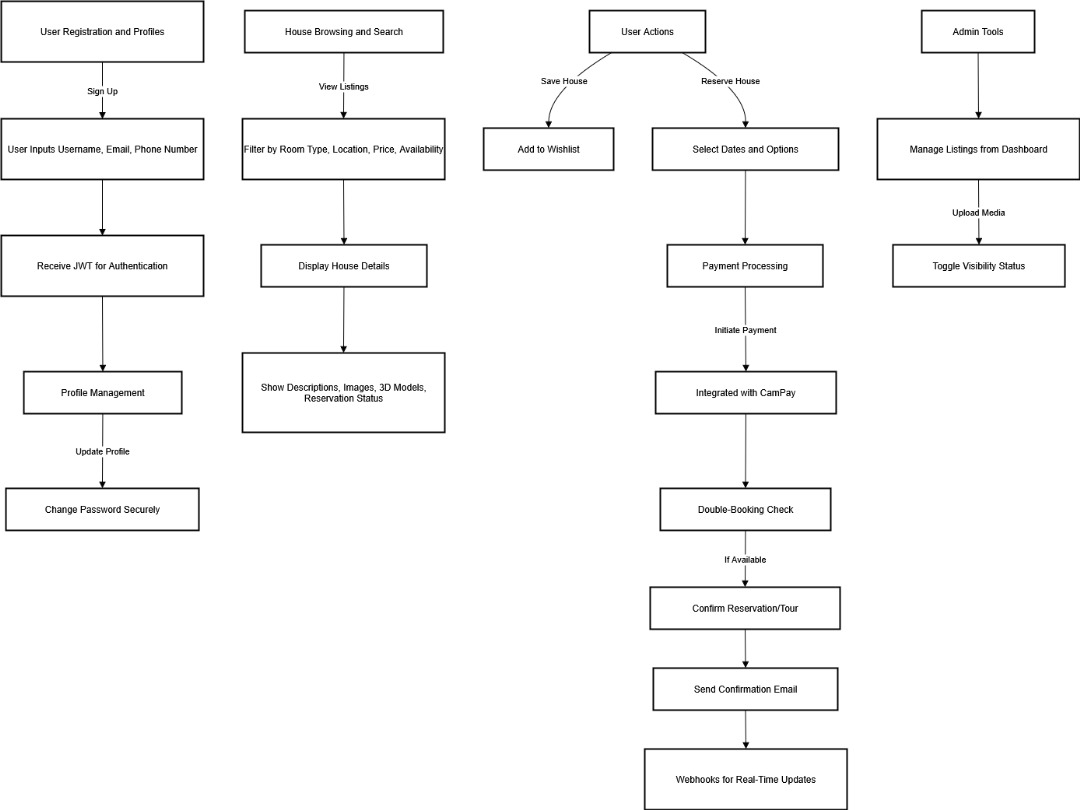
3. House reservation with payment  


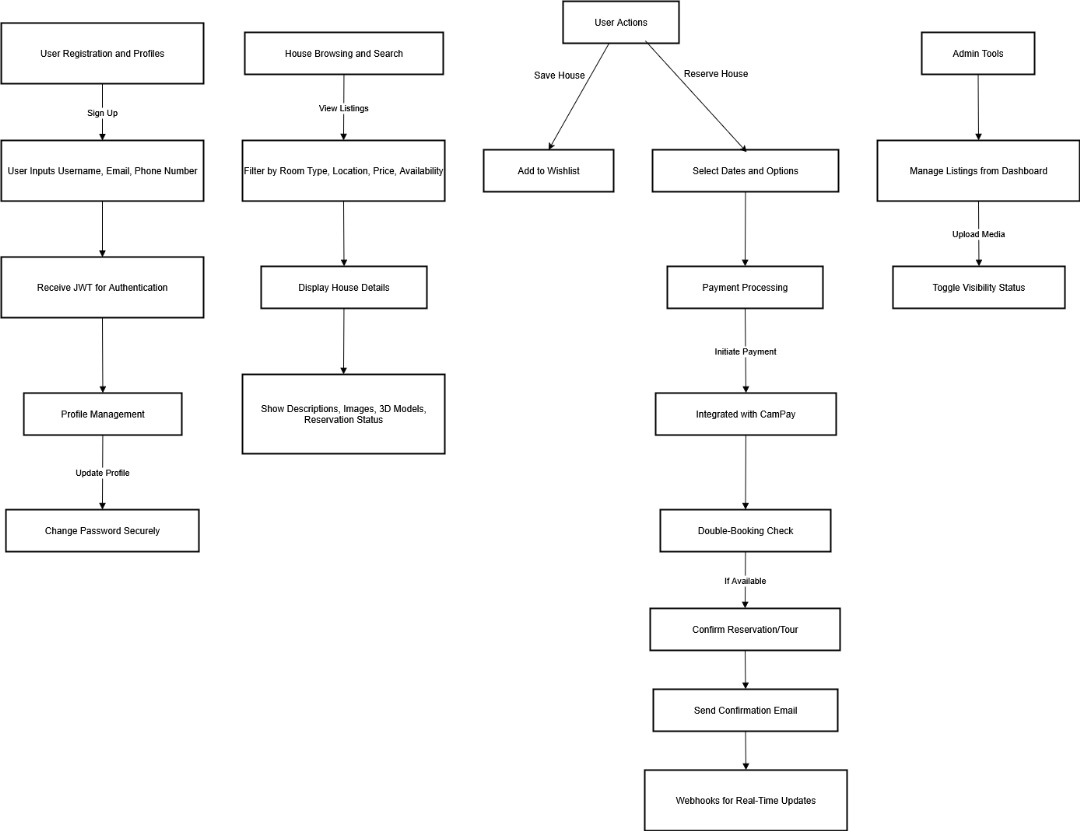
4. Admin uploading House media  


5. Payment Webhook Notification  


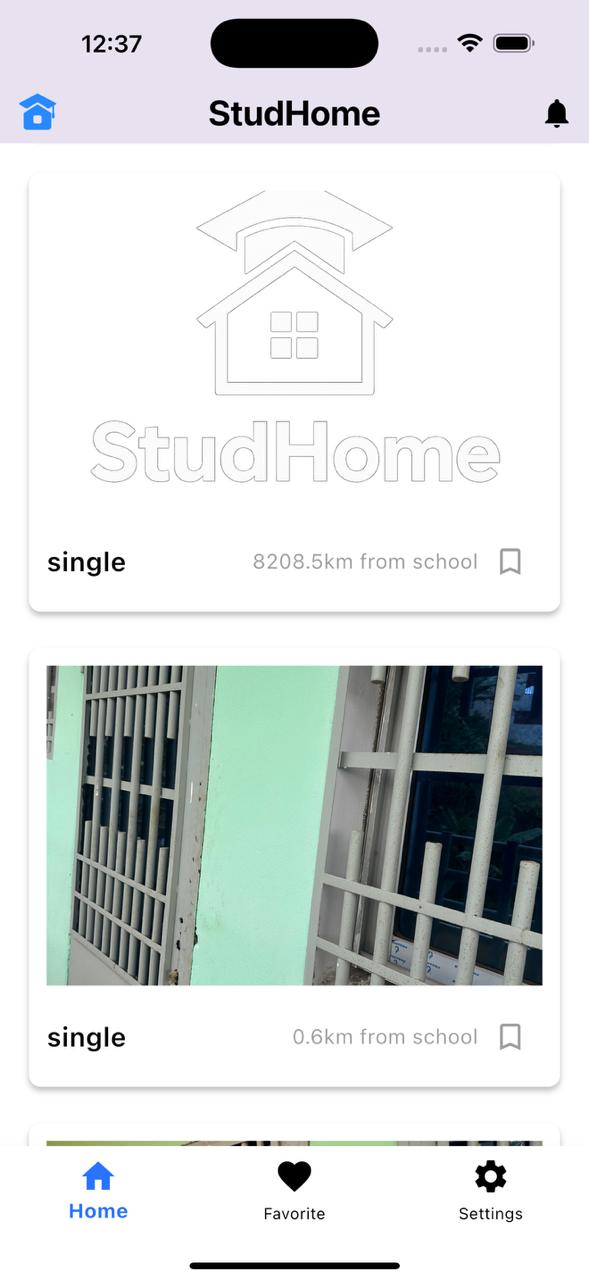
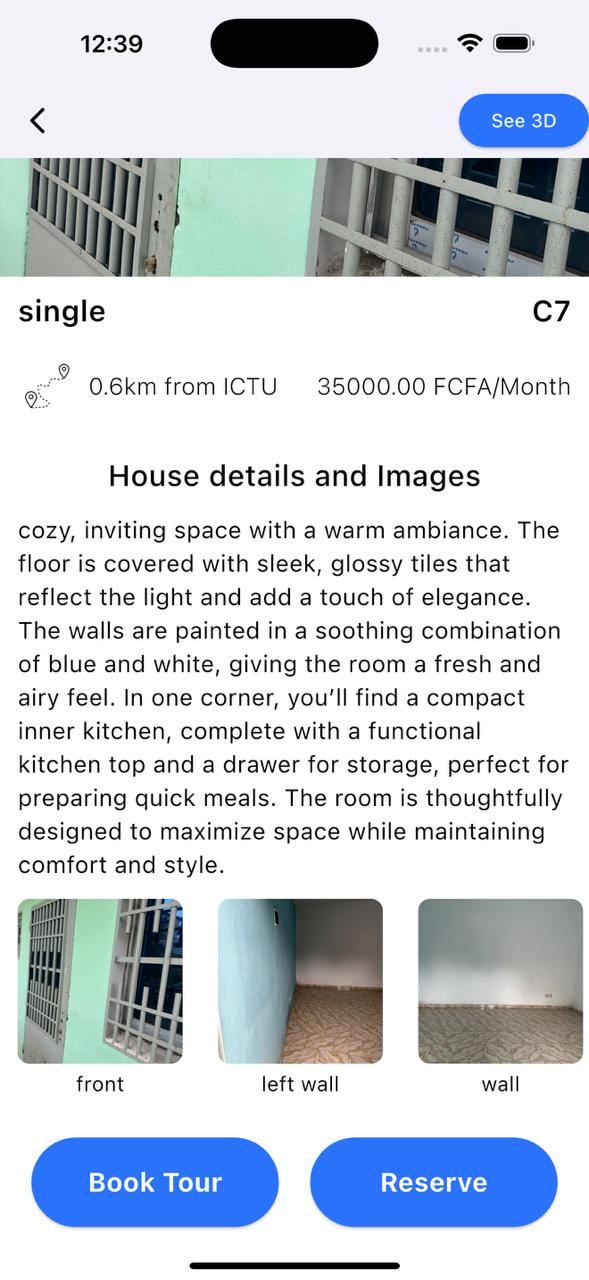
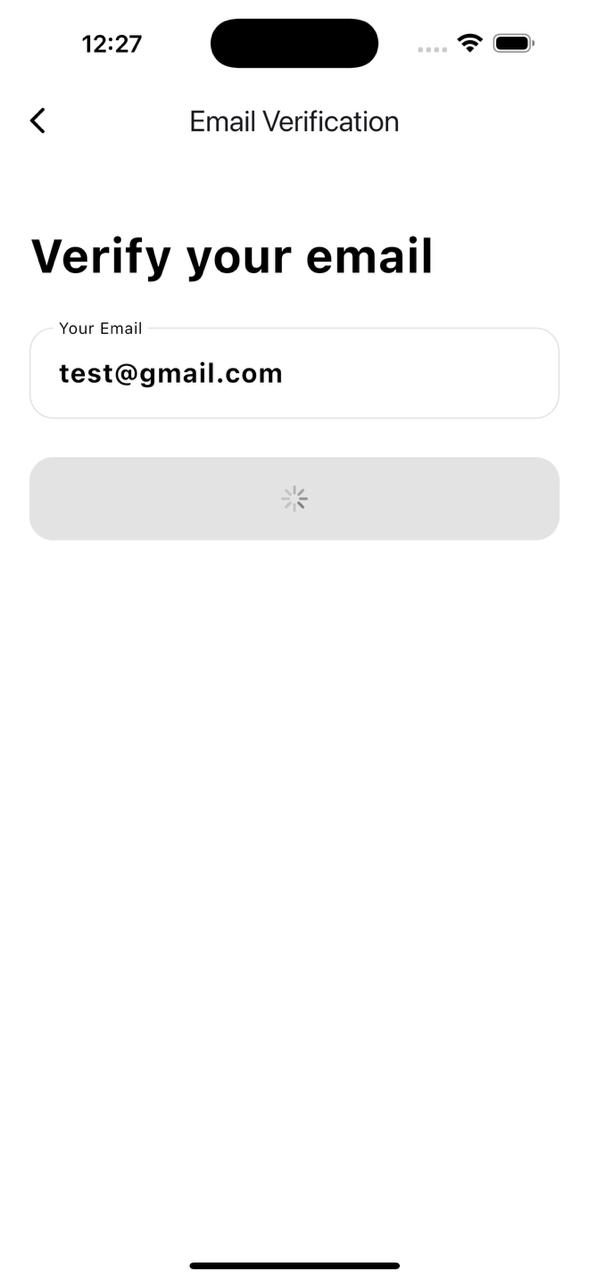
ER Diagram

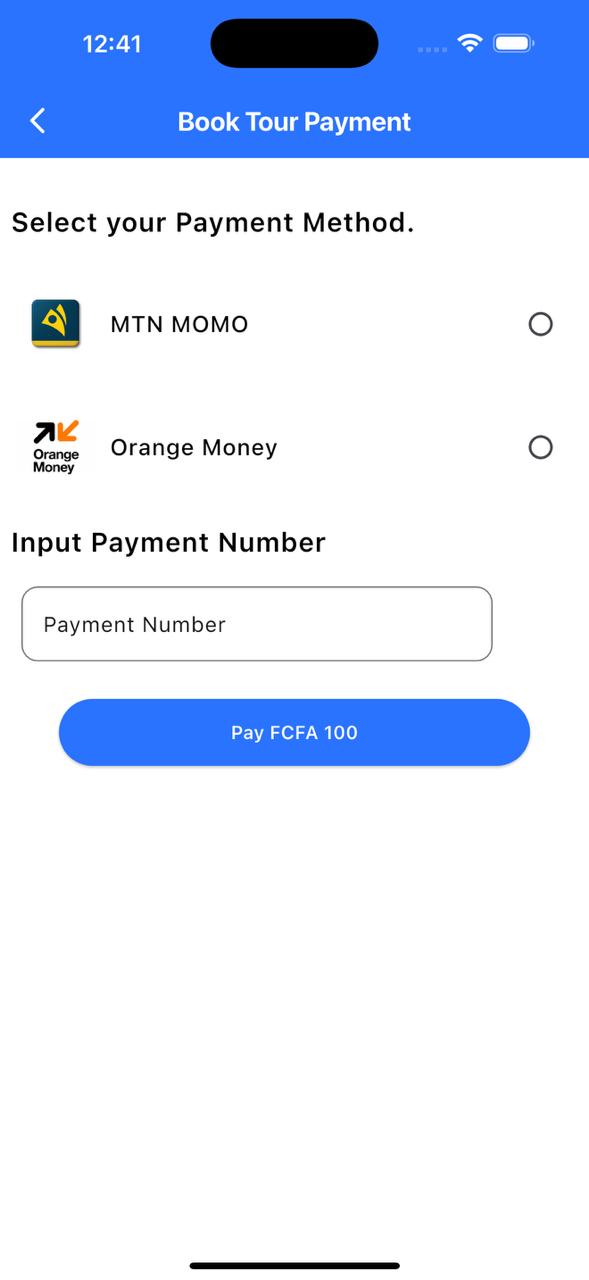


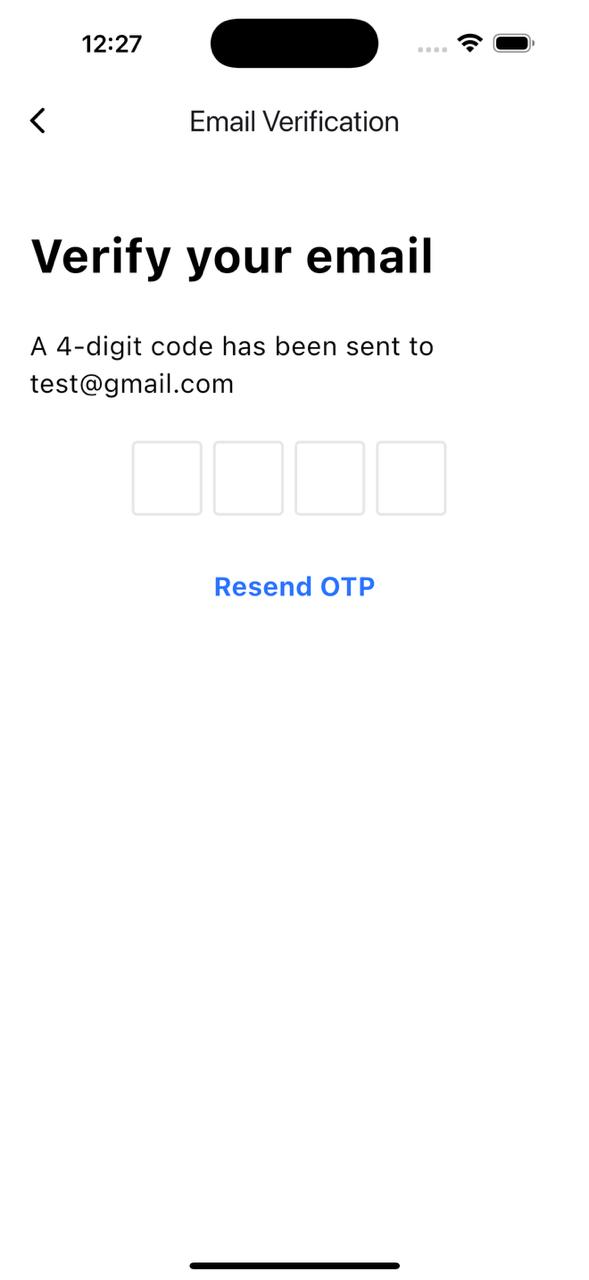
Activity Diagram  
1. 

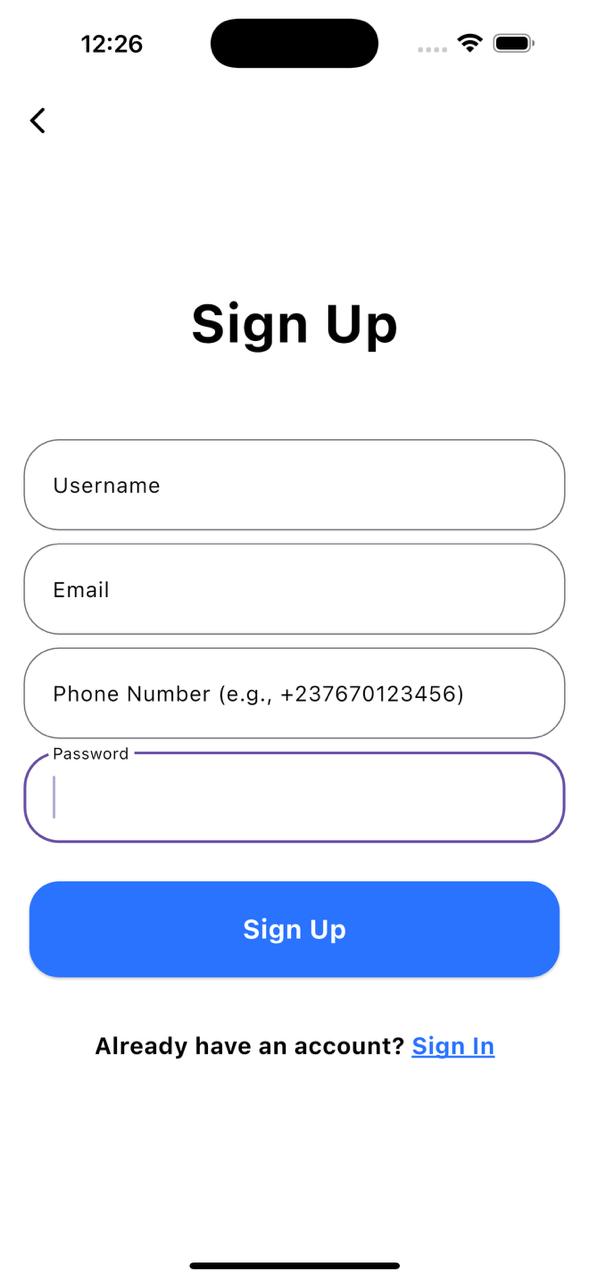
2.  


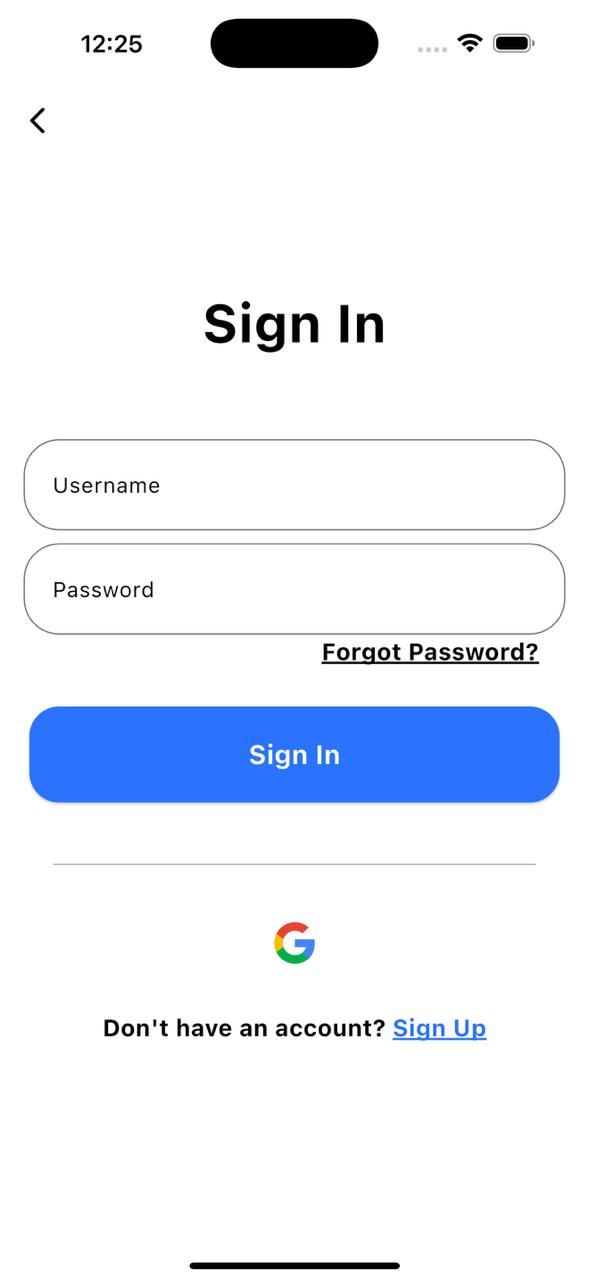
Design Patterns  
Design principles  
Technology Stack

Results and Discussion  
Screenshots of various application scenarios  
  










Screenshots of various API Request/Response