Software modeling and design

2025/2026

Project Documentation

prepared by :

Halim Osama 320230144

Supervised by:

Dr. Mustafa Alsayed

Eng. Zeina Sherif

1) introduction:

Project Description:

Rento is a web-based rental platform designed to assist Egyptian university students in finding suitable accommodation near their universities. In many regions across Egypt, students face serious difficulties when searching for housing, including misleading listings, lack of reliable information, and rental fraud. These challenges make the process of finding a safe and affordable place to live stressful and time-consuming during the study period.

Project Purpose:

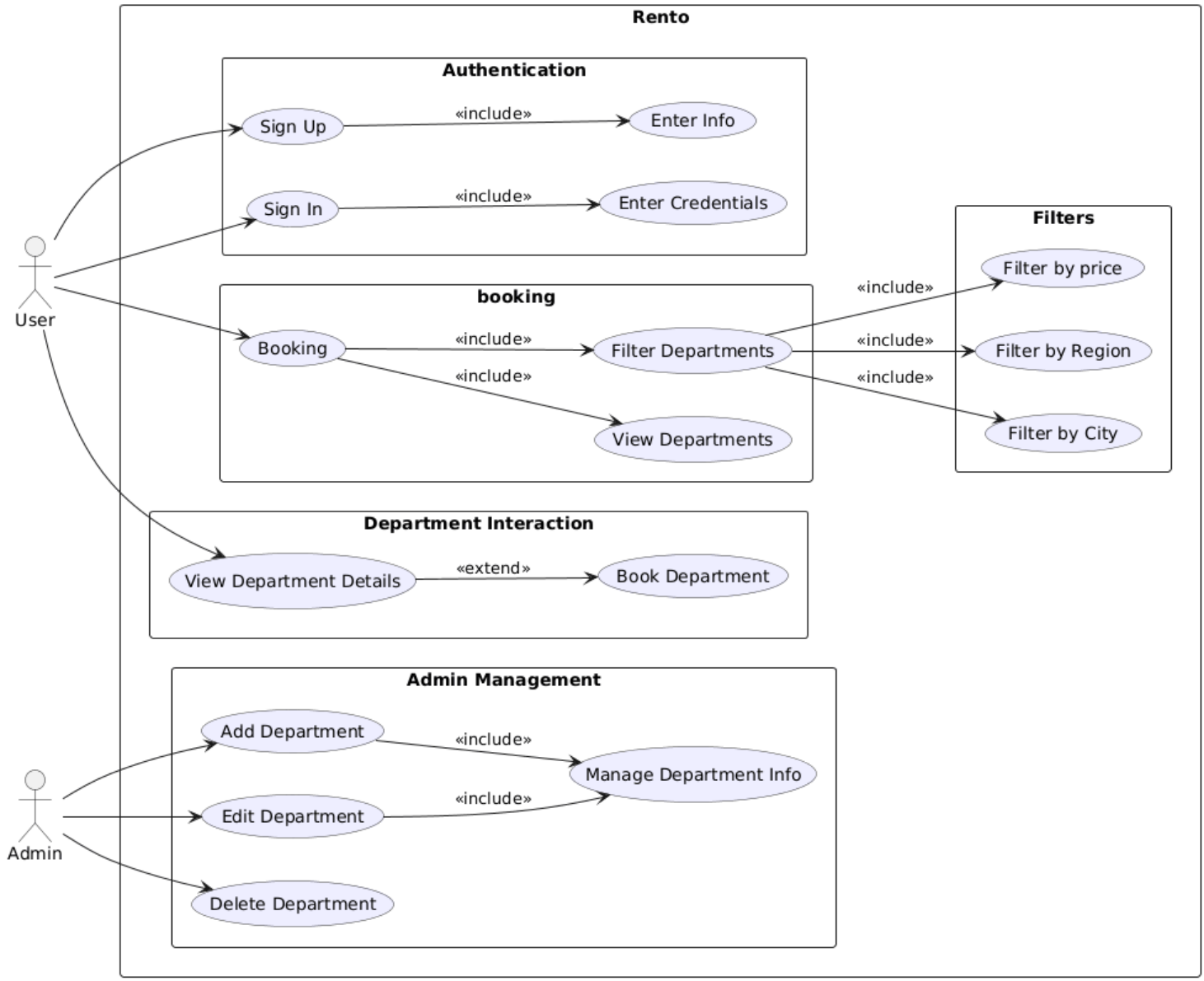
The main purpose of Rento is to connect renters (apartment owners) with student renter through a trusted and well-organized platform. The system enables students to browse available apartments, view detailed property information, and apply filters based on budget and required features such as location, apartment size, and available facilities. This helps students make informed decisions that better match their needs during the university period.

Innovative Aspects:

The innovative aspect of Rento lies in its focus on solving a real and widespread problem faced by Egyptian students. Unlike generic rental platforms, Rento is specifically tailored to student requirements, with an emphasis on budget constraints and feature-based matching. By addressing these student-specific challenges, the platform provides a practical solution that reduces housing-related risks and improves the overall student living experience.

2) Modeling Section:

1. Use case:



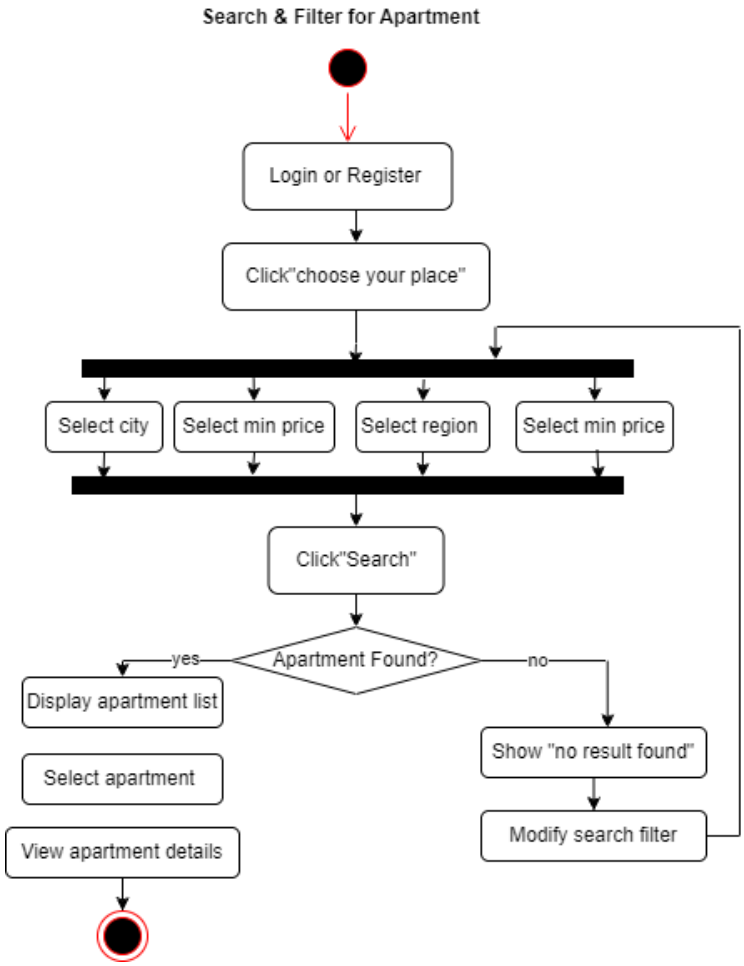
- The **Rento** Use Case Diagram is organized into functional subsystems to maintain a clear separation of concerns between client-side interactions and backend management. We utilized **<<include>> relationships** for mandatory procedures, such as "Enter Credentials" during Sign In, to signify that these steps are essential for the completion of the base use case. Similarly, the **Filters** subsystem is included within the "Filter Departments" process to demonstrate that price, region, and city parameters are integral, non-optional components of the refined search functionality.
- We opted for an **<<extend>> relationship** for "View Department Details" to indicate that this is a conditional, supplemental action that occurs only if a user chooses to explore specific property data during the booking flow. For the **Admin Management** module, we centralized common logic into the "Manage Department Info" use case via inclusion; this design choice was made to reduce redundancy, as both adding and editing departments rely on the same core data-handling mechanisms. This structure ensures that the system's requirements are modular and that actor permissions are strictly partitioned.

2. Activity Diagram:

2.1 Search & Filter:

- This Diagram for the **Search & Filter** process utilizes **thick bar** to model apartment filtering as a set of parallel actions. This design choice indicates that users can define criteria—such as city, region, and price—in any order or simultaneously, reflecting a flexible and non-linear user interface.

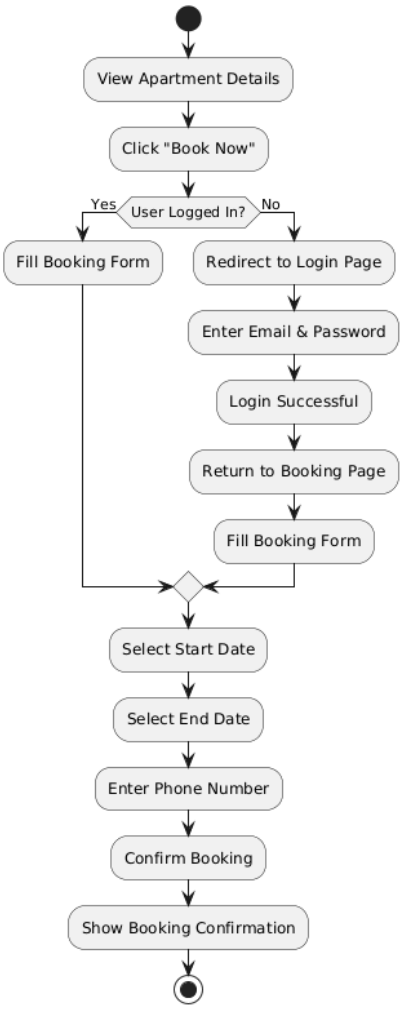
- A **decision node** is placed after the search trigger to handle system outcomes based on property availability. If no results are found, the flow incorporates a **loop-back mechanism** to the filtering stage, allowing the user to "Modify search filter" rather than forcing them to restart the entire process. This ensures a user-centric workflow that facilitates iterative searching until the final state of viewing apartment details is achieved



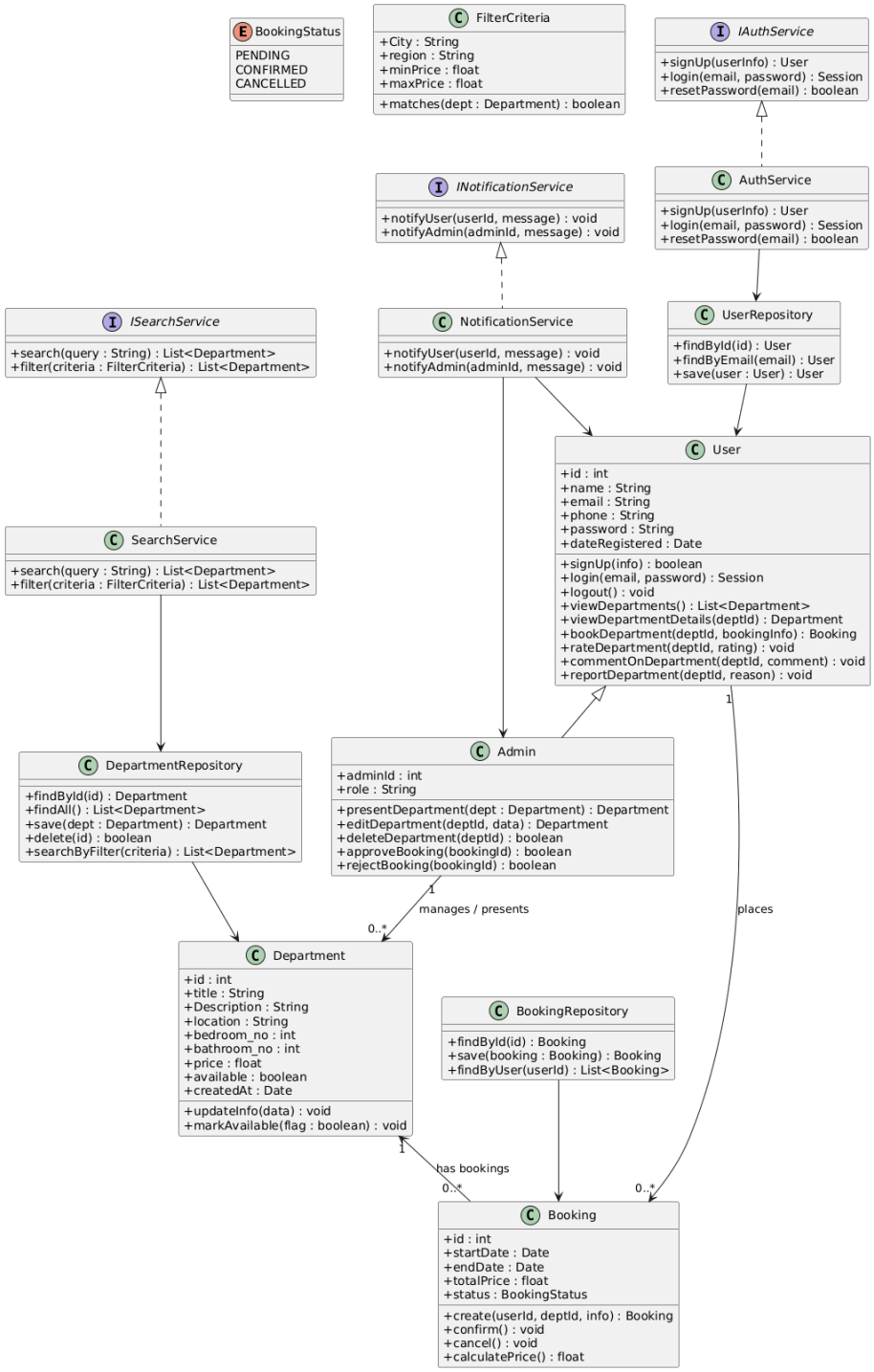
2.2 Booking stage:

- This diagram for **Booking Apartment** focuses on enforcing data integrity and security through a structured workflow. We implemented a **decision node** immediately after the "Book Now" trigger to verify the user's session status. This design choice ensures that authentication is a prerequisite for the booking transaction, automatically redirecting unauthenticated users to a login sub-flow before they can access the booking form.

- the diagram follows a strictly sequential progression for capturing booking details. We organized the selection of dates and contact information into a linear funnel to reduce user cognitive load and ensure that all mandatory fields are completed in a logical order. This culminates in a final confirmation state, providing a clear endpoint that validates the transaction's success for the user.



3. Class Diagram:

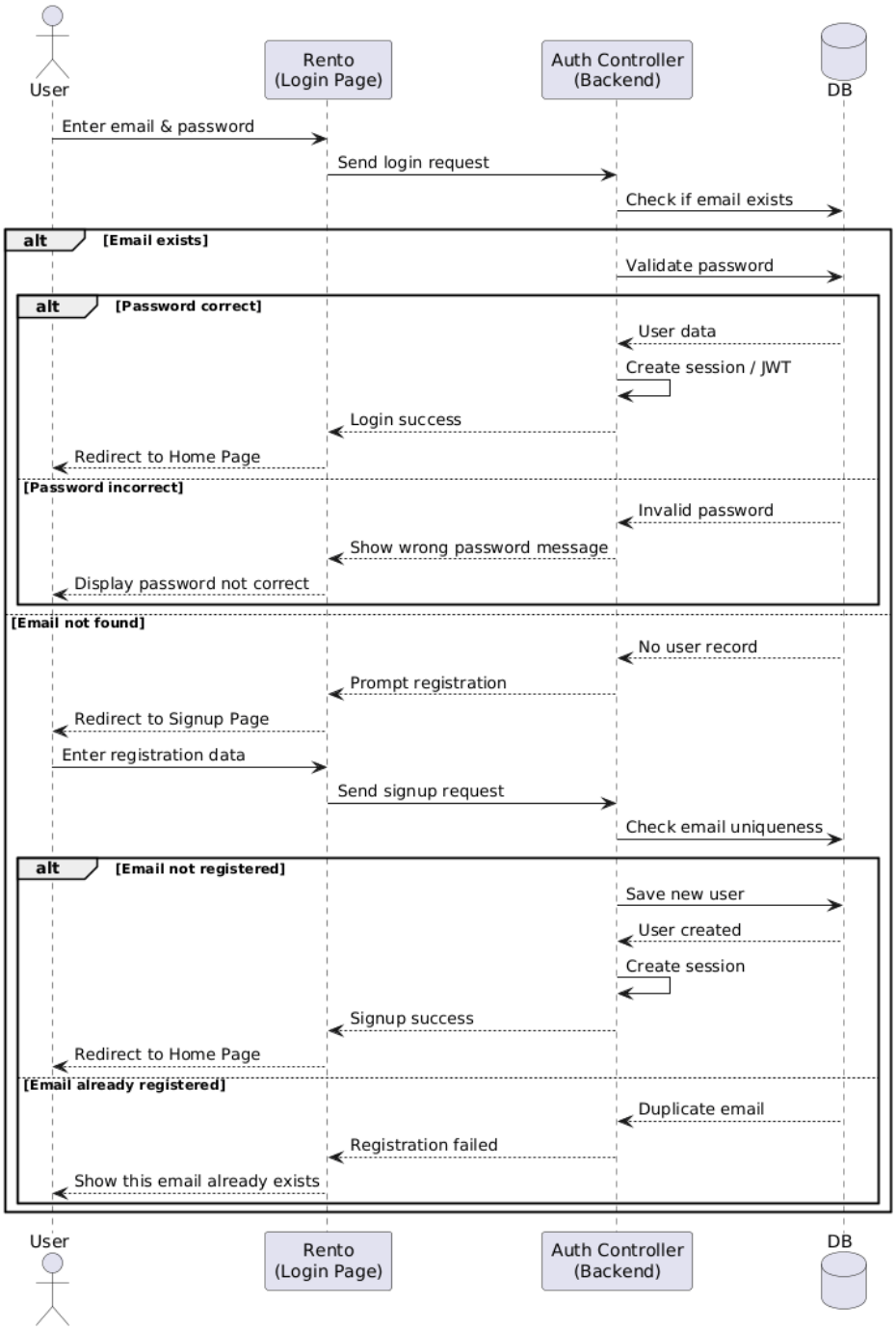


- The Rento Class Diagram utilizes a Repository pattern (e.g., DepartmentRepository, BookingRepository) to decouple core business logic from data persistence, ensuring the system remains maintainable and database-agnostic. We implemented an inheritance relationship where the Admin class extends the User class; this choice allows administrators to inherit foundational profile and authentication attributes while gaining exclusive functional permissions, such as managing department listings.

- To enhance modularity and testability, we employed an interface-driven design for core services like ISearchService, IAuthService, and INotificationService. This allows the system to swap out specific implementations—such as search algorithms or notification providers—without affecting dependent classes. Furthermore, the associations between User, Booking, and Department define a clear transactional flow: a single user can place multiple bookings, while each booking is strictly linked to a specific department to ensure data integrity during the reservation process.

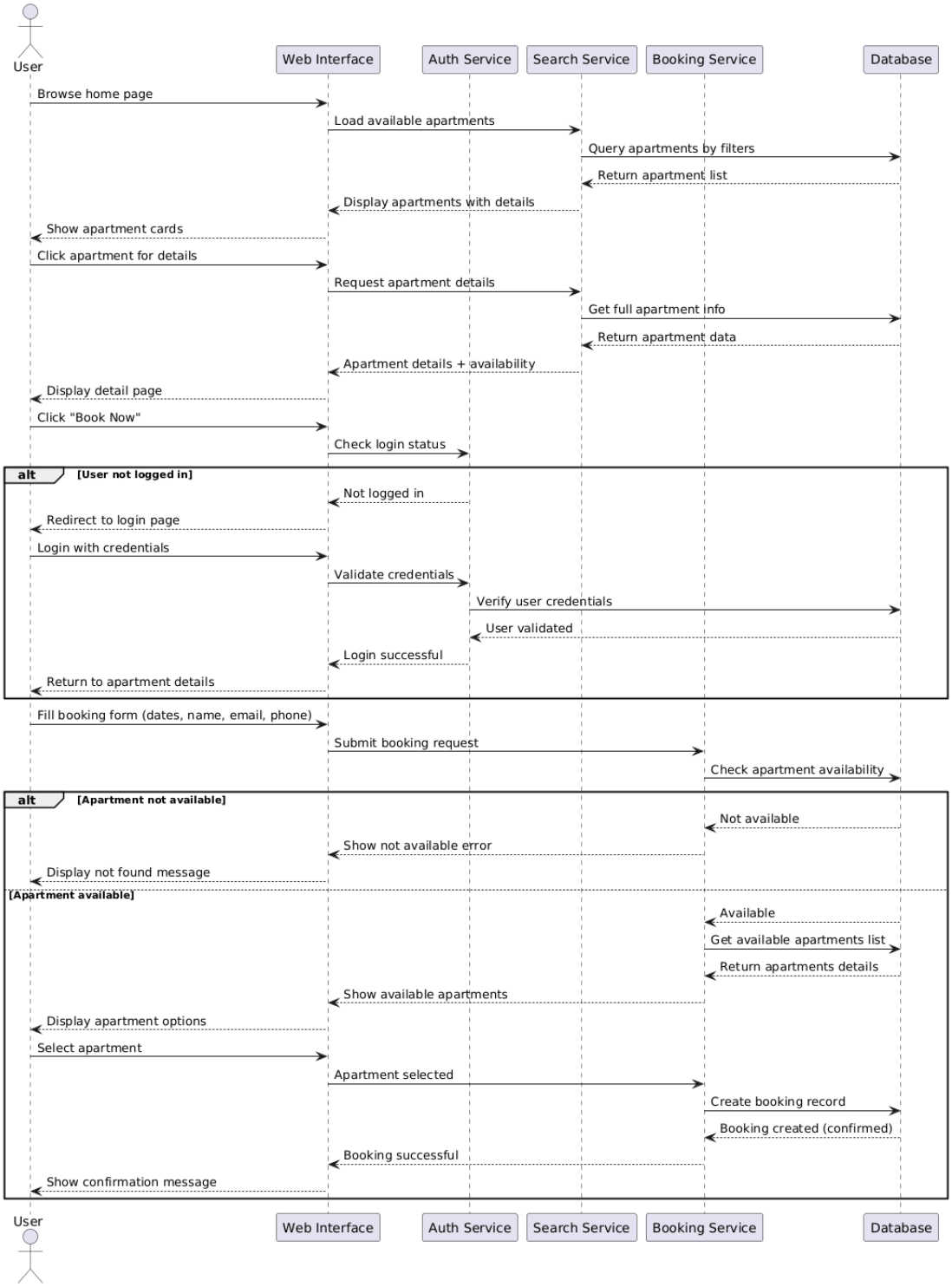
4. Sequence Diagram:

4.1 Login/signup:



- The **Login/Signup Sequence Diagram** utilizes nested **alt (alternative) fragments** to map out the conditional logic required for secure authentication and user onboarding. We chose this design to clearly distinguish between primary success paths and error-handling scenarios, such as incorrect passwords or non-existent accounts, within a single cohesive flow. This ensures that every potential response from the **Auth Controller**—from successful validation to displaying specific error messages—is explicitly accounted for.
- A key design choice was the inclusion of **Session/JWT creation** directly following successful verification; this signifies the transition from an unauthenticated state to a secure, authorized session. Additionally, we integrated a branching path that redirects users to a registration flow if no existing record is found in the **Database**. This approach minimizes user friction by automatically guiding them through the signup process while maintaining strict data integrity checks, such as verifying email uniqueness before saving a new user record.

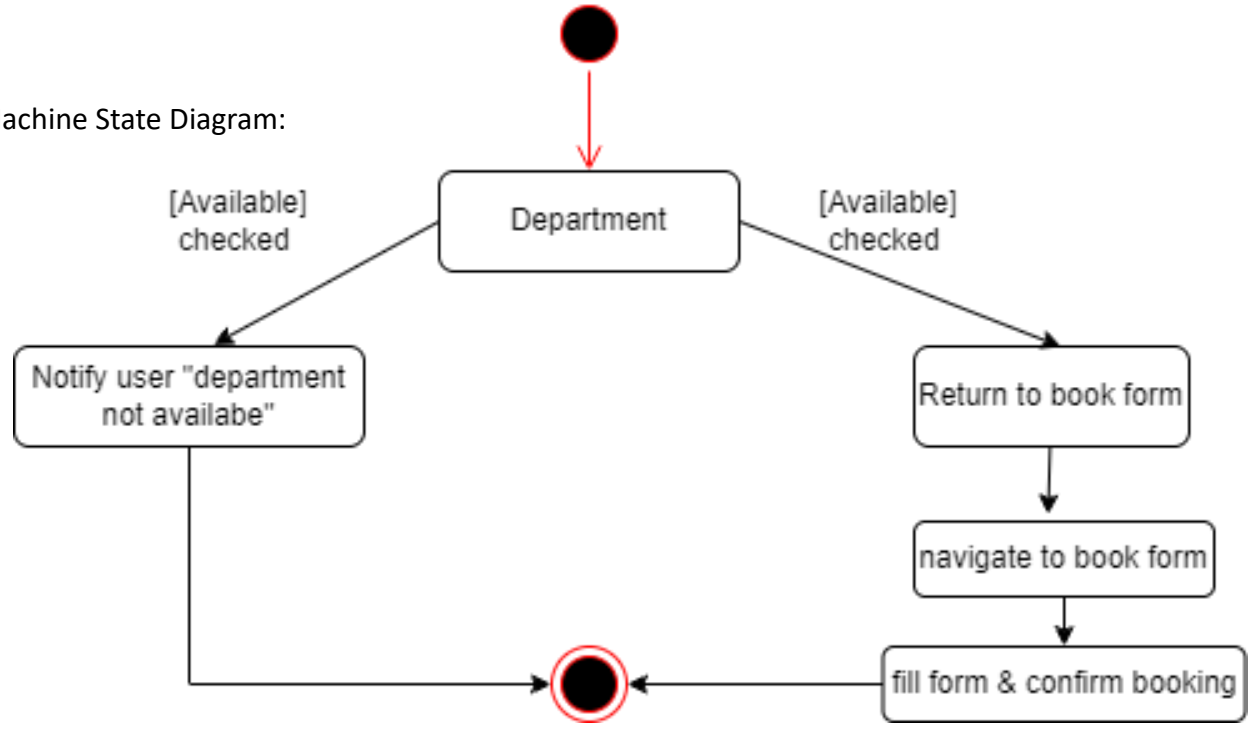
4.2 Search & Booking:



- The Booking Sequence Diagram follows a service-oriented interaction model involving the Web Interface, Auth Service, Search Service, and Booking Service to ensure modularity. We chose to decouple these services so that specific tasks, such as querying apartment details or validating user credentials, are handled by specialized components before interacting with the Database. This structure supports system scalability and ensures that the core booking logic remains independent from the initial search and discovery phase.

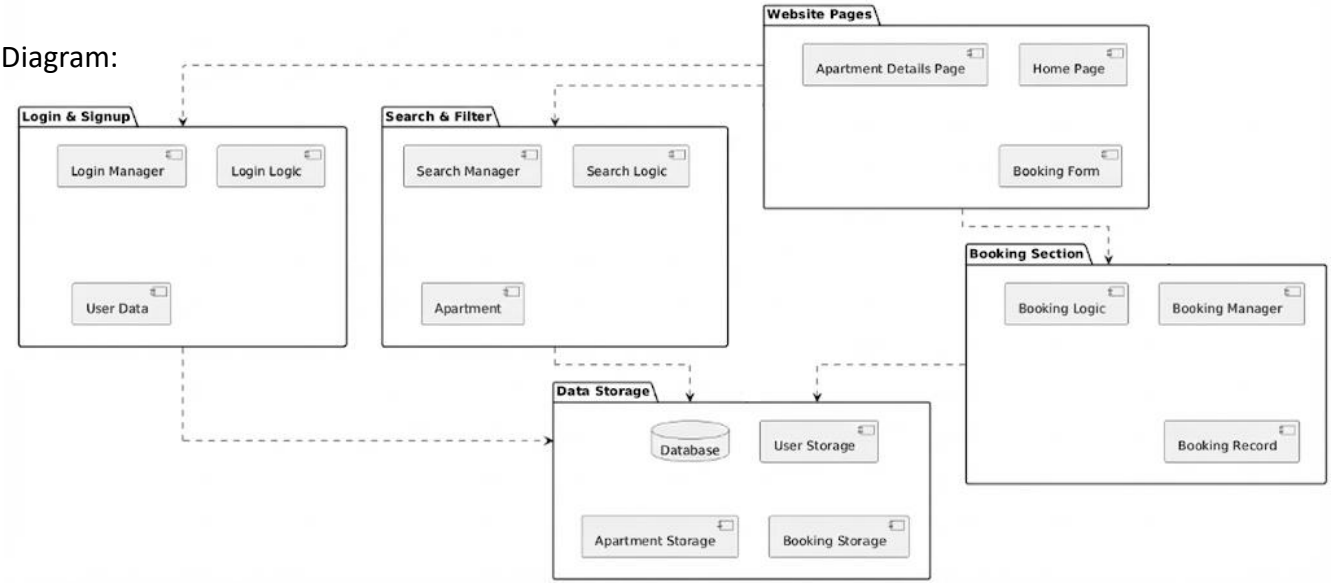
- To manage the transaction's complexity, we utilized alternative (alt) fragments to handle critical conditional logic regarding authentication and property availability. The alt [User not logged in] block creates a secure checkpoint, ensuring only authenticated users can proceed to submit a booking form. Similarly, the alt [Apartment not available] fragment provides a robust error-handling mechanism that prevents overbooking by verifying the real-time status in the Database before a final booking record is created.

5. Machine State Diagram:



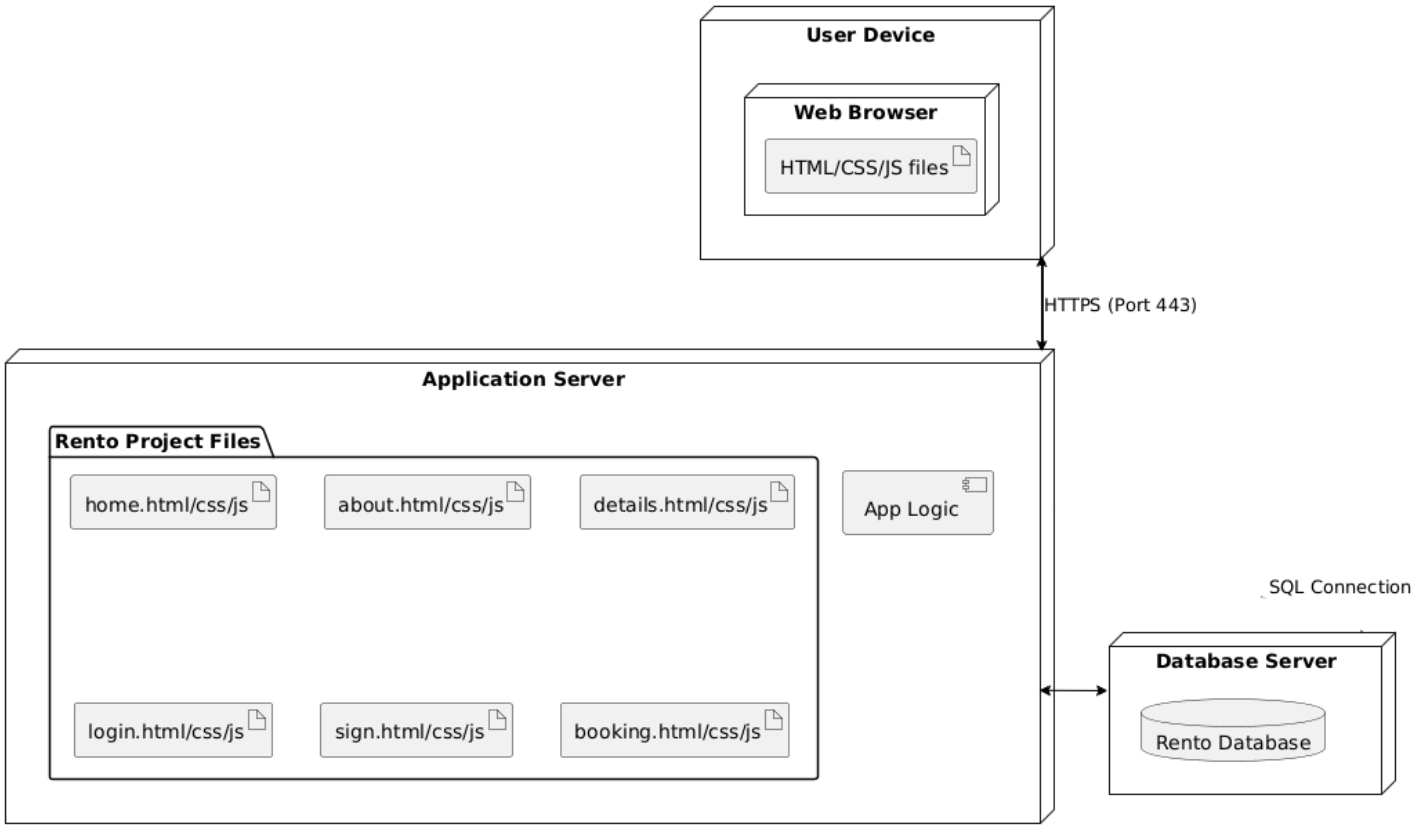
- The **Department Availability State Machine** uses a conditional branching mechanism to manage the transition from property selection to final booking based on real-time status. We chose to implement the availability check as the primary gateway at the "Department" state to ensure the system validates the listing before the user invests time in the booking process. This design choice allows for an immediate notification path if the unit is unavailable, preventing dead-end interactions and maintaining data consistency. For the successful path, we designed a linear sequence that guides the user through the book form and final confirmation, ensuring that the "confirm booking" state is only reachable once all availability prerequisites and form requirements are strictly met.

6. Packet Diagram:



- The **Rento Package Diagram** is designed with a layered architecture to separate user-facing pages from core business logic and data persistence. We chose to organize functions into specific modules like **Login & Signup**, **Search & Filter**, and **Booking Section** to ensure high cohesion and simplify future maintenance. By directing all functional dependencies toward the **Data Storage** package, we created a single point of truth for the database and repositories. This design choice abstracts the storage complexities from the **Website Pages**, ensuring that changes to the database do not require a rewrite of the entire user interface.

7. Deployment Diagram:



- The **Rento Deployment Diagram** establishes a three-tier physical architecture that organizes the system's components into three distinct nodes: the **User Device**, **Application Server**, and **Database Server**. We chose this distributed setup to physically separate the front-end user interface from the back-end business logic and data storage, ensuring enhanced security and system scalability. By deploying project files like home.html, login.html, and booking.html as artifacts on the application server, we maintain a direct relationship with our actual project file structure while using **bi-directional HTTPS and SQL connections** to facilitate secure, two-way data exchange between the browser and the core database. This physical design choice effectively supports the modular architecture defined in our **Package Diagram**, allowing individual tiers to be updated or migrated independently without disrupting the overall system flow.

3) Architectural Justification:

Chosen Architecture: 3-Tier Client/Server:

- The Rento Web Application is structured into three distinct physical and logical layers: the Presentation Tier (User Device), the Application Tier (Application Server), and the Data Tier (Database Server).

We chose the 3-Tier Client/Server Architecture for the following reasons:

1. **Scalability:**
   By separating the system into three tiers, we can scale each part independently based on demand. For example, if the number of users searching for apartments increases, we can upgrade the Application Server without needing to change the Database Server. Furthermore, the modular design of the Search & Filter and Booking packages allows us to expand specific features—like adding a map-based search—without restructuring the entire application.

2. **Maintainability:**
   The project is organized into functional modules such as Login & Signup, Search, and Booking, which follows a Component-Based logical approach within the layers. This separation ensures that developers can update the "Booking Record" logic without accidentally breaking the "Login Manager". Additionally, the physical file structure (folders for about, booking, home_page, etc.) makes the project easy to navigate and manage as it grows.

3. **Security:**
   This architecture significantly improves security by ensuring the Database Server is not directly accessible to the user. All data requests from the Web Browser must first pass through the Application Server via secure HTTPS connections. The Application Server acts as a gatekeeper, validating user credentials through the Login & Signup module before allowing any SQL queries to reach the Data Storage.

4. **Decoupling (Independence):**
   The use of a dedicated Data Storage package abstracts the database complexities from the UI. This means that if we decide to switch from a local MySQL database to a cloud-based service like Supabase, we only need to modify the Database Server connection and the Data Storage code, leaving the Website Pages completely untouched.

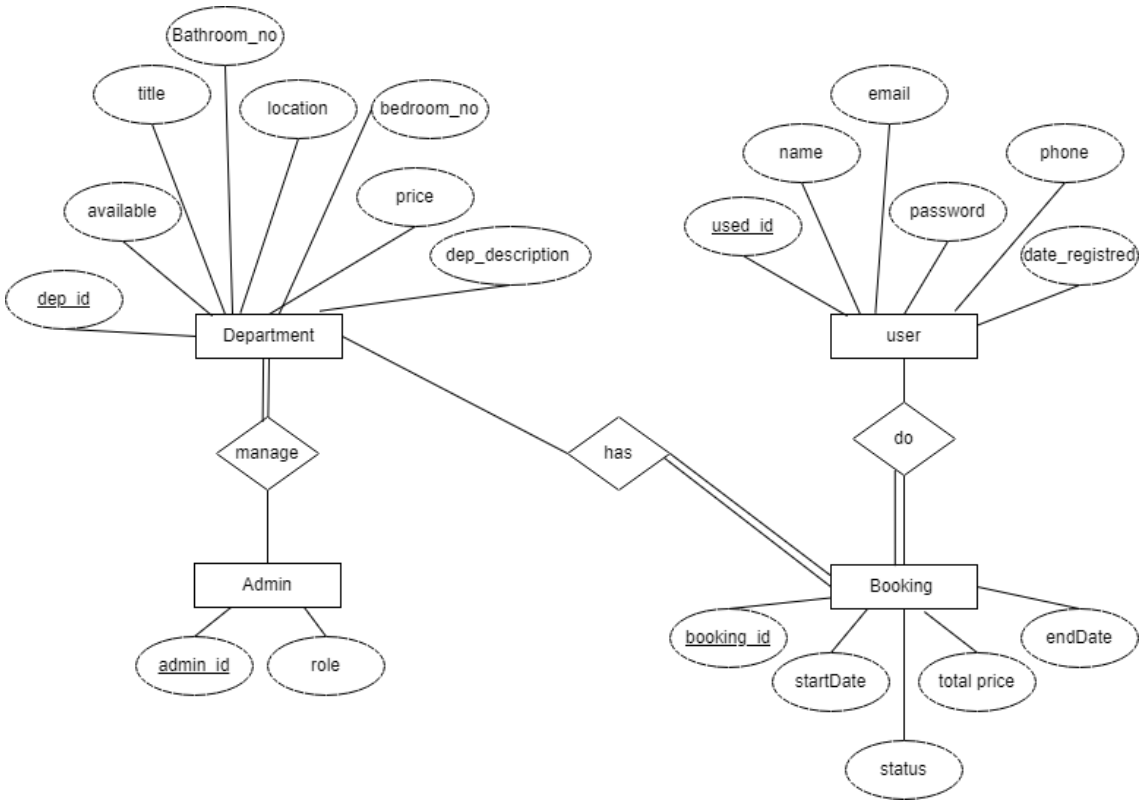
4) Database Design:

1. Entity Relation Diagram(ERD):

- The database is designed to support an apartment/department booking system
- The main entities are User, Admin, Department, and Booking, with clearly defined relationships.

Entities and Attributes:

- **User**
  - ❖ user_id (PK) – unique identifier
  - ❖ name
  - ❖ email (unique)
  - ❖ phone
  - ❖ password
  - ❖ date_registered



- **Admin**
  - ❖ admin_id (PK)
  - ❖ role
  - ❖ user_id (FK)

- **Department**
  - ❖ department_id (PK)
  - ❖ title
  - ❖ description
  - ❖ location
  - ❖ **bedroom_no**
  - ❖ **bathroom_no**
  - ❖ **price**
  - ❖ **available**
  - ❖ **created_at**
  - ❖ **admin_id (FK)**

- **Booking**
  - ❖ booking_id (PK)
  - ❖ start_date
  - ❖ end_date
  - ❖ total_price
  - ❖ status
  - ❖ user_id (FK)
  - ❖ department_id (FK)

Relationships

- **User → Booking**:
  One user can place many bookings, but each booking belongs to exactly one user (1:M).

- **Department → Booking**:
  One department can have many bookings, but each booking is for one department (1:M).

- **Admin → Department:**
  One admin can manage multiple departments, while each department is managed by one admin (1:M).

2. Normal Forms Explanation:

The database schema is normalized to Third Normal Form (3NF) to eliminate redundancy and ensure data integrity.

1) First Normal Form (1NF):

✓ Atomic (indivisible) attributes
✓ No repeating groups or multi-valued attributes
✓ Primary keys uniquely identifying each record

Example: The users table stores one email per user, not multiple emails in one field.

2) Second Normal Form (2NF)

✓ All non-key attributes are fully dependent on the entire primary key.
✓ Each table has a single-column primary key
✓ No partial dependency exists

Example: In the bookings table, attributes like start_date, end_date, and status depend only on booking_id.

3) Third Normal Form (3NF)

✓ There are no transitive dependencies.
✓ Non-key attributes depend only on the primary key
✓ Related data is separated into appropriate tables

Examples:

User details are stored only in the users table, not repeated in bookings.
Department information (price, location) is stored in departments, not duplicated in bookings.